

Enabling High Quality Executable Domain Specific Language Specification

A thesis submitted to the University of Manchester for the degree of
Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2014

Qinan Lai
School of Computer Science

Table of Contents

Table of Contents	2
List of Figures	7
List of Tables	9
Abstract	10
Declaration	11
Copyright Statement	12
Acknowledgement	13
Chapter 1. Introduction	14
1.1 Motivation	16
1.2 Research objectives	20
1.3 Contributions	21
1.4 Research methodology	22
1.5 Thesis structure	25
Chapter 2. Domain Specific Language Foundations	26
2.1 Programming language specification	26
2.1.1 Concrete syntax	27
2.1.2 Abstract syntax	28
2.1.3 Behavioural semantics	28
2.2 Domain Specific Language development	29
2.2.1 DSL language specification	30
2.2.2 Traditional ways of DSL development	31
2.2.3 Developing DSL by a model-driven approach	33
2.3 Methods for defining DSLs	34
2.3.1 Review of DSL abstract syntax definition	35
2.3.2 Survey of semantic description approaches of DSLs	35
2.4 Quality of DSL specifications	44

2.4.1	Model quality goals.....	44
2.4.2	Practice to improve quality of models.....	45
2.4.3	Requirements of high quality language specifications.....	47
2.5	Summary.....	51
Chapter 3. Model-driven foundations		52
3.1	Model-Driven Engineering	52
3.1.1	Meta-modelling.....	53
3.1.2	Model-driven work flow	54
3.2	Semantics of UML.....	55
3.2.1	Previous work to formalise the semantics of UML	55
3.2.2	Foundational subset of UML.....	56
3.2.3	Summary.....	56
3.3	Action Language for fUML (ALF)	57
3.3.1	Features of the ALF notation	57
3.3.2	A tutorial of ALF	58
3.4	Summary.....	61
Chapter 4. A Framework for Quality Language Specification (FQLS)		63
4.1	DSL syntax and semantics definition	63
4.2	Architecture of FQLS	65
4.2.1	Definition layer.....	65
4.2.2	Analysis layer	67
4.2.3	Execution layer.....	69
4.2.4	Summary.....	71
4.3	A development process for DSL	72
4.3.1	Software & Systems Process Engineering Meta-Model.....	72
4.3.2	Method content	74
4.4	Summary.....	83

Chapter 5. Defining DSLs using FQLS.....	84
5.1 Defining DSL via ALF	84
5.1.1 Representing abstract syntax by ALF	85
5.1.2 Representing static semantics by ALF	85
5.1.3 Representing behavioural semantics via FQLS.....	87
5.2 Defining a Petri net language	90
5.3 Defining Petri net language via ALF	93
5.4 Discussion	96
5.5 Summary.....	99
Chapter 6. Static analysis of DSL specifications using FQLS	100
6.1 Extended static checking	100
6.2 Building static code analysers	107
6.3 Bridging FQLS specification with fUML	110
6.3.1 Atlas Transformation Language	111
6.3.2 Mapping ALF to fUML	112
6.4 Summary.....	117
Chapter 7. Executing DSL specifications using FQLS.....	118
7.1 Architecture of the code generation project	119
7.2 From ALF structural aspects to Emfatic.....	122
7.3 From ALF's behavioural aspects to Emfatic	125
7.3.1 Generating statements.....	126
7.3.2 Generate expressions.....	128
7.4 Discussion	132
7.5 Summary.....	134
Chapter 8. Case study:	
Formalising Business Process Execution Language	135
8.1 Introduction to WS-BPEL	135

8.1.1	Compositing web services with BPEL	135
8.1.2	Structure of a BPEL process.....	137
8.1.3	Execution of a BPEL process	141
8.2	Scope of the case study	142
8.3	Defining abstract syntax	145
8.4	Defining behavioural semantics	147
8.4.1	Execution model overview.....	147
8.4.2	Variables	149
8.4.3	Communication	149
8.4.4	Semantics of Basic Execution	151
8.4.5	Semantics of Structured activities.....	156
8.4.6	Semantics of Scope and Process	158
8.4.7	Correlations	160
8.4.8	Abstract activities	163
8.5	Checking and testing the language specification	164
8.6	Summary.....	165
Chapter 9.	Evaluation.....	166
9.1	Evaluating the ALF-based BPEL specification	166
9.1.1	Syntactic correctness/consistency evaluation.....	167
9.1.2	Evaluating semantic correctness by testing	167
9.1.3	Evaluating model quality by software metrics	170
9.2	Evaluating static checkers of the ALF Language Specification Framework.....	179
9.3	Limitations.....	181
9.4	Summary.....	182
Chapter 10.	Conclusion and further work.....	183
10.1	Conclusions.....	183

10.1.1	Summary of contributions	183
10.1.2	Summary of evaluation	184
10.2	Further work	185
Appendix A.	Complete specification of BPEL.....	187
Appendix B.	Guidelines for the FQLS process.....	224
	Decide the semantic definition strategy	224
	Guideline: Behavioural semantics development.....	225
	Guideline for developing behaviours	226
	Guideline for creating semantics definition architecture.....	227
	Guidelines for implements an external checking strategy.....	227
Appendix C.	List of errors sourced from static checkers of other languages.	229
Bibliography	232

Word Count: 48307.

List of Figures

Figure 1: Process of design research.....	23
Figure 2: Components of language specification.	27
Figure 3: An example of a meta-model.....	28
Figure 4: Roles and products in the DSL development process.....	30
Figure 5: Components of translational approach.	36
Figure 6: Components of operational semantics.....	39
Figure 7: Languages used in rewriting approach.	40
Figure 8: Languages used in weaving approach.....	42
Figure 9: The context of model-driven workflow.....	54
Figure 10: MDA workflow.	55
Figure 11: Architecture of FQLS.	65
Figure 12: Workflow of analysis layer.....	69
Figure 13: Overview of SPEM.	74
Figure 14: Phases of DSL development.	75
Figure 15: Products and process phases.....	75
Figure 16: Developing abstract syntax.....	77
Figure 17: Develop behavioural semantics.	78
Figure 18: Checking language specification.	79
Figure 19: Testing language specification.....	81
Figure 20: Attaching behaviours directly to meta-model operations.	87
Figure 21: Attaching behaviours as separate operation rules.....	88
Figure 22: Variable and Statements meta-model.....	89
Figure 23: Variable and Statements run-time meta-model.....	89
Figure 24: Attaching behaviours to a run-time meta-model.....	90
Figure 25: The meta-model of a PNTD.....	91
Figure 26: Executable meta-model of PNTD.	94
Figure 27: Defining semantics as activities.	96
Figure 28: Checking name typos.	97
Figure 29: Checking non-existent properties.	98
Figure 30: Validation package.....	109
Figure 31: ALF text to model transformation.....	112
Figure 32: An example in the ALF grammar meta-model.....	113

Figure 33: Arithmetic expression models in the ALF meta-model.....	114
Figure 34: Compiled fUML model of <code>getActiveTransition</code>	117
Figure 35: Generating Java code.....	121
Figure 36: Project structure of the executor generator.....	121
Figure 37: Graphical syntax of BPEL.	137
Figure 38: BPEL and WSDL.	138
Figure 39: Abstract syntax of WSDL.	145
Figure 40: Abstract syntax of BPEL.....	146
Figure 41: Runtime meta-model of BPEL.....	148
Figure 42: State machine of basic execution.	151
Figure 43: A flow activity that contains pick activity and links.	152
Figure 44: Basic executions from runtime meta-models.	153
Figure 45: Structured executions.	156
Figure 46: State machine of <code>ScopeExecution</code>	158
Figure 47: example of correlations.....	161
Figure 48: Illustration of correlations.	162

List of Tables

Table 1: ALF syntax that is similar to that of Java.....	59
Table 2: Errors identified from established checker.....	101
Table 3: Errors identified in the BPEL case study.....	102
Table 4: fUML analysis approaches	111
Table 5: Transformation languages.....	112
Table 6: Mapping between ALF code and fUML models.....	115
Table 7: Mapping from ALF to Emfatic.....	122
Table 8: Mapping from ALF statements to Java.....	127
Table 9: BPEL handlers.....	139
Table 10: BPEL basic activities.....	140
Table 11: BPEL structured activities.....	141
Table 12: Differences between BPEL 1.1 and 2.0.	143
Table 13: Added concepts in BPEL 2.0.	143
Table 14: testing files	169
Table 15: environments	170
Table 16: LOC comparison.	173
Table 17: Cychomatic complexity comparison.	175
Table 18. Cognitive weights of control structures.....	176
Table 19: Cognitive complexity comparison.....	179

Abstract

Domain Specific Languages (DSL) are becoming a common practice for describing models at a higher abstraction, using a notation that domain experts understand. Designing a DSL usually starts from creating a language specification, and the other tools of the DSLs are derived from the specification. Hence, the quality of the language specification can crucially impact the quality of the complete DSL tool chain.

Although many methods for defining a language specification have been proposed, the quality of the language specification they produced is not emphasised. This thesis explores the quality of language specifications, and proposes consistency, correctness, executability, understandability, and interoperability as the key features that a high quality language specification processes.

Given the importance of these features, this thesis designs a new language definition approach that is based on the newly published OMG standards, namely: the semantics of the foundational subset of UML (fUML), and the Action Language for fUML (ALF). This approach enables the creation of a language specification with the proposed criteria. Moreover, a software framework that simplifies the production of high quality language specifications is built. Finally, a software development process is developed, which analyses the roles, products, and activities in DSL specification development.

The framework is demonstrated by defining the language specification of Business Process Execution Language (BPEL) as a case study. The BPEL specification is further evaluated, which confirms the desired quality features are processed.

Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

- The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.
- Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.
- The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.
- Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see <http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487>), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see <http://www.manchester.ac.uk/library/aboutus/regulations>) and in The University’s policy on Presentation of Theses.

Acknowledgement

I would like to express my greatest gratitude to my supervisor, Dr. Andy Carpenter, for his invaluable feedback, patient guidance, support, and encouragement.

I would like to offer my special thanks to my external examiner, my internal examiner, and my advisor Dr. Renate Schmidt for their precious feedback.

This thesis could not have been accomplished without my family and friends for the support they offered through these years. I would like to thanks my parents for their supports and help. My deepest thanks are to my wife, Ying Jing, for the support, help and love.

Finally, I would like to thank the School of Computer Science of the University of Manchester for the financial support via the Overseas Research Scholarship.

Chapter 1.

Introduction

The history of programming languages shows them evolving from machine languages to assembly languages, then to third generation languages like Java or C++. An alternative way to see this is as enabling the use of higher levels of abstraction for specifying implementations. That is, specifications that contain less low-level detail and which are less dependent on a single platform. By using a higher-level language, a problem can be solved more quickly and the solution usually contains fewer errors. For example, consider using an assembly language or Java to solve the same problem. The assembly code requires developers to remember many different instruction words and memory locations, and the process is error-prone. On the other hand, the Java software would contain fewer code lines and would be easier to write and debug.

Third generation languages are not the end of the programming language evolution. Language developers are seeking to raise the abstraction level further by creating new languages that have more concise and powerful syntaxes. For example, new languages, such as Scala, Groovy, Xpand¹ and Google's Go language include concepts like closure, higher-order functions and sequence expressions. Mature languages are also enhanced as a result of the development of newer programming languages. This can be seen in the development of Java 7 and Java 8 where support for simplified syntaxes, such as for multiple catch statements, and higher-level concepts, such as lambda expressions, are occurring. Thus, perceived deficiencies in existing languages lead to the creation of new languages, and the creation of new languages leads to the enhancement of existing languages.

The programming paradigm is also changing. While the object-oriented programming paradigm is still dominant, newer languages absorb concepts from Functional Programming, and Aspect Oriented Programming (AOP) can make logic clearer, and produce code that is easier to develop and maintain [78]. Meanwhile, the development of newer frameworks also makes programming easier for certain application areas. For

¹ <http://wiki.eclipse.org/Xpand>

example, a well-supported domain is web application with frameworks such as Spring, JSF and Ruby on Rails widely used.

One important aim of these technologies is to allow developers to focus on the problem that they are trying to solve. Therefore, by utilising these technologies, they can ignore many details that are not relevant to the problem. However, in all of the approaches mentioned, the focus is on the solution domain. In other words, although some high level concepts are supported, they still capture the details of the solution and developers must put effort into creating a solution from the problem. Modelling the solution rather than the problem means they could be trying to solve the wrong problem; moreover, modelling a problem using a language intended for defining solutions always involves unnecessary detail.

A promising way of raising the abstraction of programming languages is to capture the problem directly using Domain Specific Languages. A Domain Specific Language (DSL) is a programming language designed for a particular domain or application area. Deursen et al. [155] defines a DSL as

[a] “programming language or executable specification language that offers, through appropriate notations and abstractions, expressive focus on, and usually restricted to, a particular problem domain”.

Compared to General-Purpose Languages (GPL), DSLs are more expressive for the domain they serve. DSLs are thought to be the next generation of programming languages [100]. However, it is not always possible to distinguish between GPLs and DSLs, because the main differences are the problem domain on which they focus. The term ‘domain-specific’ is a relative concept when compared to general purpose languages [156].

Another way of raising the abstraction level is to use modelling. Models are the abstraction of a real system or problem, and are widely used as a way of expressing design blueprints. The Model-Driven Engineering (MDE) [134, 16] approach suggests focussing on models rather than on computing. It encourages the use of a high-level model to describe the system, and the entire implementation of the system can be automatically derived from the models by model transformations and code generation [148]. DSL programmes and models share certain similarities, since both try to capture the problem domain via a higher-level abstraction language. Hence, the development of DSL and models are related, and they share various theories and tools.

This chapter presents an overview of current DSL development and highlights the problems that motivated the research presented in this thesis. Section 1.1 motivates my

work, and is followed by Section 1.2 that introduces the research objectives and the hypothesis of the thesis. Section 1.3 highlights the contributions of the thesis. Section 1.4 discusses the methodology of this project. Finally, Section 1.5 presents a structural overview of the complete thesis.

1.1 Motivation

DSLs are not new concepts. Examples such as UNIX shell have a longer history than that of most of the current GPLs. On the other hand, programming in DSLs shared many similarities to modelling. Both DSLs and models are ways of precisely and concisely describing a narrow domain. Indeed, DSLs provide a way of expressing models in such a manner that domain experts, who may not know programming, can use the notations they understand for development. To use models in MDE also requires the design of a language (abstract syntax) that is sufficiently expressive for the target domain that the required models can be specified. By first designing a DSL and then using it to solve problems, MDE has demonstrated higher productivity level with fewer errors than traditional software development approaches that separate design and implementation [102].

Developing a useful (domain specific) language entails not only defining its syntax and semantics, but also creating support tools, such as editors, executors (either compilers or interpreters) and, possibly, verifiers. Historically the creation of DSLs has been held back by the high cost of developing this tooling [144]. The characteristic of MDE to define things, including languages, by models and then use these models as inputs to transformations and generators means that with the appropriate transformations and generators, DSL support tooling can be created from a model of the language drastically cutting the cost of creating this tooling. This application of MDE is referred to as Model-Driven Language Engineering (MDLE).

MDLE has provided good support for the development of domain specific programs expressed in a DSL. Code editors (textual and graphical), tools for checking the well-formedness of programs and code generators can be produced by tools developed as part of the Eclipse Modelling Project (EMP). There are also many other commercial or open source projects that can provide some or all of this support [43]. However, an important aspect of a language specification, namely behavioural semantics, is not well supported by MDLE.

Behavioural semantics refers to the rules used to execute a programme. It is noted that some languages are designed purely for structural modelling and do not have execution or evaluation semantics. Although such non-executable languages still have semantics, this thesis concentrates only on the behavioural semantics of executable DSLs.

As an important aspect of a DSL specification, behavioural semantics should be clearly defined. However, in practice, they are often defined by English prose or a reference implementation with, hopefully, consistent prose. Although prose makes the specification accessible to a wider audience than a technology based description, it has many limitations. In particular, as natural languages are subject to readers' interpretations, it can be ambiguous. Also, because the specification cannot be automatically processed, executors, simulators and debuggers cannot be derived from the DSL specification.

The definition of behavioural semantics is shifting from a prose-based one to a formally specified one. Several approaches to formally defining behavioural semantics have been adopted by real-world DSLs (see Subsection 2.2.3 for examples). However, many researchers have identified defects in the published DSL specifications. If MDLE defines behavioural semantics as models, their quality is affected by the same factors as any model. The literature reveals that consistency, correctness, understandability, executability and interoperability are seen as the most important factors affecting the quality of models [101, 140, 103, 137, 19, 23].

Consistency of DSL specification

In DSL development, a common approach is to define the abstract syntax and behavioural semantics via different languages. The syntax domain and the semantic domain are then linked by a manual or automatic (but usually not bi-directional) transformation. Hence, it is possible that inconsistent concepts exist, such as the semantic definition referring to concepts that do not exist in the abstract syntax definition, or abstract syntax elements without semantics. The inconsistency between abstract syntax and the behavioural semantics of DSLs, which is also called the 'fragmentation problem' [137], can lead the two to evolve separately with undetected inconsistencies affecting later developments.

The Specification and Description Language Specification (SDL) [153] is an example of how syntax and semantics can evolve separately. The former version of SDL-2000 defines syntax and semantics as two separated standards. In a major update - SDL-2010,

significant changes were made to the syntax; however, the semantic specification is still that of SDL-2000, in which these changes are not applied to the official language standard.

Correctness of DSL specification

Assume the language specification is consistent, it is still necessary to maintain the *semantic correctness* of the specification. Semantic correctness reflects whether the semantics of the DSL specification are as the language developers intended (see Section 2.4.3 for details). The correctness of the language specification is an important factor that affects the quality of language specifications. An incorrect language specification can confuse its audience and can crucially impact the correctness of other products that are derived from it.

In practice, the correctness of many DSL specifications is questionable. For example, Glaser et al. [42] examined the language specification of SDL-2000 and identified that there are thousands of errors in the specification. Similarly, Wilke and Demuth [160] performed a check on the UML specification, which identified that nearly 80% of its static semantics are incorrect. These examples demonstrate the need of DSL specifications development process to be improved.

Executability of DSL specification

The way of which a DSL is executed is defined as the executability of the language specification, the actual execution is done by an executor. While creating the DSL specification, the developers may develop a reference implementation in parallel. This reference implementation can be used to test the language specification and further maintain its correctness. When releasing the DSL specification, the reference implementation can also be released as a official executor.

Due to the high cost of development, a reference implementation is usually not a compulsory part when releasing a DSL specification. It is possible that the behavioural semantics of a DSL has never been tested until its release, which results the released language specification uncertain quality. Thus, there is a conflicted interest between maintaining the quality of the DSL specification and reducing the development cost.

Understandability of DSL specification

The prime aim of models is communication and the target can be either humans or machines [132]. Traditionally, GPLs are developed by computer scientists who have fundamental programming language knowledge, which means that they may use one of the

many formal frameworks that exist for defining semantics using mathematical notations. Unlike GPLs, DSLs developers are often lack the knowledge to understand formal notations [65, 137].

In contrast to the GPL, the semantics of DSL can be more complex. During the boom of multi-core and parallel computing, DSLs needed semantics that included the concepts of parallel behaviours, threads and signals. For example, reactive programming languages (Esterel [12] and HUME [63]) and process definition languages (BPMN [52], BPEL [147] and xSPEM [10]) contain concurrent concepts. However, existing development approaches fail to obtain both expressiveness and understandability, as will be discussed in the literature review chapters. The result is that the definitions of semantics are either not easy to understand or not easy to compose by normal DSL developers. Both factors result in a complex specification, which affects the understandability of the specification.

Interoperability of DSL specification

Although there are other ways of defining abstract syntax, the MDLE approach usually defines the abstract syntax of a DSL using the Meta Object Facility (MOF)[51] as a meta-model. The MOF standard does not exist in isolation, OMG have other related standards, e.g. the MOF mapping to XMI, that means models can be moved between any tools that support these standards. Thus, abstract syntax descriptions are interoperable across multiple tools. In contrast, there is no widely agreed standard for defining behavioural semantics meaning many approaches have been used (see Section 2.3). Thus, behavioural semantic specifications are only usable by the original tools used to create them; i.e. they are not sharable.

Quality assurance tasks for DSL specification

There are various ways of ensuring the quality of the models. Many researchers proposed that modelling processes, automatic error prevention, simulation and conventions are practices for improving model quality [103]. Although it is desirable to apply these methods to a DSL specification, as the number of DSL users is very small, it is only worth doing so if the cost thereof is small. The current state of defining DSLs uses separates specification languages with limited expressiveness, which makes it difficult and not cost-effective to improve the model quality of DSL specifications. The ways of maintaining model quality are expanded in the literature review chapters.

In summary, developing the semantics of DSLs while retaining the quality thereof is a challenging task. Inconsistent, incorrect and complex semantics affect the quality of DSL

specifications. On the other hand, a high quality DSL specification should be *consistent, correct, executable, interoperable* and *understandable*.

1.2 Research objectives

The literature review identifies that current approaches to defining DSL specifications, especially behavioural semantics, lack quality assurance and that many of the existing semantic specifications for DSLs are of low quality. The main causes of this are:

- The separation of abstract syntax and behavioural semantic definitions which allows inconsistency between the two elements.
- The limited expressiveness of modelling languages causes DSL specifications to be complex and not easily understood.
- The lack of executability makes it difficult to test the specification, thus leading to an incorrect specification .
- The lack of interoperability makes DSL specification not processable by a wide ranges of machines , which results in having less tool support.

The DSL specification is used as the basis for understanding the language, and is the guide for related tool development. Thus, investigating ways of ensuring the quality of DSL specifications is a significant task. There are several properties proposed as the features of a high quality DSL specification, which can be categorised as follows:

- Features can be addressed by a DSL definition approach. Interoperability and understandability are features of the DSL definition approach.
- Features can be addressed by a DSL development methodology. Correctness, particularly semantic correctness, cannot be guaranteed by a specification method because it relies largely on the interpretation of the language engineer. However, by applying a proper development methodology and by using proper tools, the errors in the specification can be identified more easily.
- Features can be addressed by both the DSL definition approach and development methodology. This includes the consistency and the executability of DSL specification. Creating a mechanism to ensure the consistency automatically is one way of dealing with the inconsistency. On the other hand, since the inconsistencies

are caused by defining DSL separately, it is promising to use a method that could define them in the same language.

As indicated above, a DSL specification would be of a high quality if a proper DSL definition approach, development method and tools were to be used. Since defining DSL in a unified manner could eliminate inconsistency errors, this thesis focuses on using a unified definition approach. The hypothesis of this thesis is therefore:

Hypothesis:

Given the importance of the DSL development, if a *highly expressive, interoperable modelling language* were to be applied for defining *both the abstract syntax* and the *behavioural semantics*, and if the development were supported by proper software development method and tools, the DSL specification would be of better quality in terms of consistency, correctness, understandability and interoperability.

Thus, the research objectives are

- To investigate the requirements of a high quality DSL specification.
- To design an approach that could define a DSL in a unified manner, fulfilling the requirements of a high quality DSL specification.
- To develop a framework to support the software development process that assists in the development of DSLs.
- To create a software development process that applies the newly defined approach.

1.3 Contributions

The contributions this thesis made can be categorised as scientific contributions and technical contributions. The scientific contribution can be summarised as:

- A new approach for defining a pure, model-based DSL specification in a unified manner is designed. The new method is based on the newly published OMG standard fUML/ALF, which defines abstract syntax and behavioural semantics in one interoperable modelling language. The unified definition makes the automatic identification of inconsistency errors possible. It also makes specifying concurrent behaviour and concepts of threads possible, without sacrificing understandability.

The thesis also made three technical contributions:

- A software framework for defining DSL and performing various quality assurance tasks is developed. It enables the definition of a DSL specification, performing a static analysis and report errors, bridging the definition domain/analysis domain, building test DSL programmes and testing the behavioural semantics.
- A modelling process for defining the DSL specification, which applies quality assurance of the DSL specification directly rather than leaving quality assurance to the implementation stage, is proposed as a SPEM model. It includes roles, activities, and guidelines.
- The thesis also confirms that, though fUML is designed to give UML semantics, it can also be used to define the semantics of a wider range DSLs. This is demonstrated by defining the Petri net example (see Chapter 5) and the BPEL case study (see Chapter 8).

1.4 Research methodology

In Section 1.1, a set of problems in current model-driven language engineering is identified. To solve these problems, it is necessary to design a solution. Hevner and Chatterjee [67] summarised that “all design begins with the awareness of a problem”. The known frameworks in design science research all started with problems, followed by the design of a conceptual framework, a prototype or a real product, and the evaluation of the design. Examples include [151] and [154]. In this section, the process of design science research followed by this thesis is illustrated.

The Design Science Research Methodology

Hevner and Chatterjee [67] extended Vaishnavi and Kuechler [154]’s work and formulated the activities in design research as a six-step-process, which is illustrated in Figure 1. The same process is followed in this research.

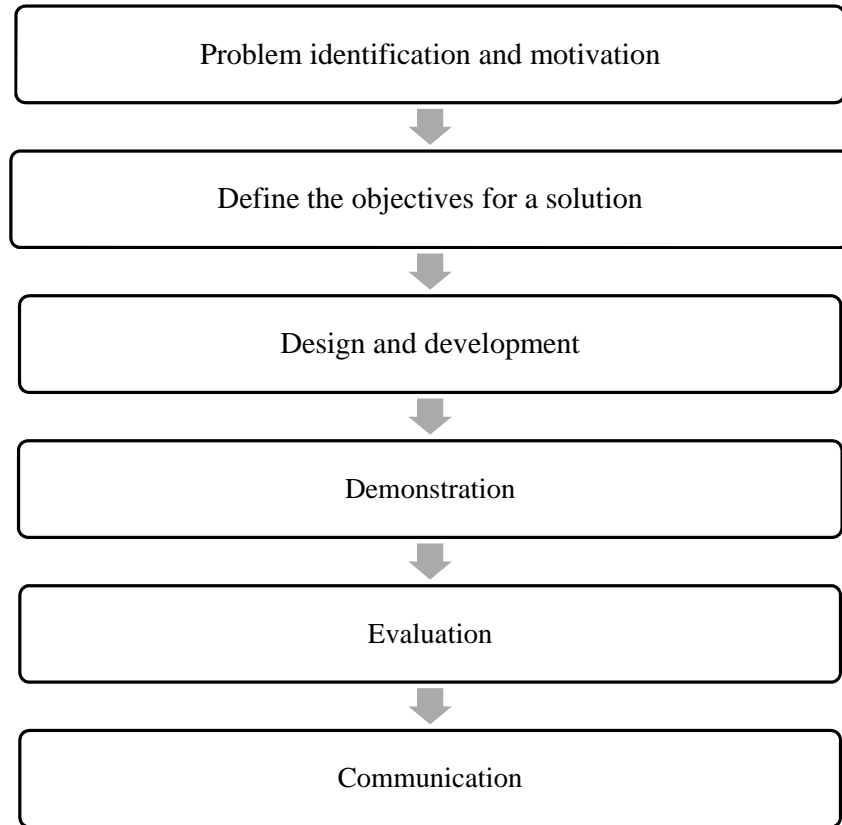


Figure 1: Process of design research.

Problem identification and motivation

This step involves defining the research problems and justifying the value thereof. The problems were previously identified in Section 1.1.

Define the objectives for a solution

By analysing these problems, a set of requirements is established, and the solution to these problems must have these features. In this thesis, the requirements of quality language specification are explored in Section 2.4.

Design and development

This step includes creating and developing the solution. Due to the features listed in Section 1.2, different approaches to defining semantics are explored, according to which the design is constructed. The contributions of the thesis all involve design and development. The framework for defining a DSL based on ALF/fUML is designed, and the supporting tools, including code editors, model transformers and an executor are implemented to perform the defining and quality assurance tasks.

Demonstration

Demonstration is an important step as it shows that the designed artefacts could solve one or more instances of the problem. This could involve experiments, simulation or case studies. In this research, the following features of our solution need to be demonstrated:

- That ALF/fUML can be used for specifying the abstract syntax and behavioural semantics of a DSL.
- The realisation of the software development process. When a software development process for DSL specification is designed, it must be demonstrated that it can be used in a real context, and that it is possible to perform quality assurance tasks, such as testing and static checking.

These demonstration aspects are well suited to a case study. A case study that uses the new software framework to realise a DSL specification could demonstrate the usability of the framework, as long as the selection of the case study is capable of showing the features proposed in Step 2. Using the software development process to implement the case study could demonstrate the usability of the software development process, while simultaneously identifying defects in the process and identifying guidelines that could be reused.

Evaluation

Evaluation is the process of examining how well the design actually addressed the problem. In this thesis, evaluation includes:

- Evaluating the DSL specification created in the case study and seeing whether the specification meets the requirements.
- Evaluating the framework and the software development process to see whether they meet the design requirements.

Communication

Hevner et al. [68] proposed that communication of the researching findings and innovations to other, relevant audiences is an independent step of design science. Aspects of the research solutions have already been published in several academic papers,

[87] Qinan Lai and Andy Carpenter. Defining and verifying behaviour of domain specific language with fuml, BM-FA '12, pages 1:1–1:7, 2012. ACM.

[88] Qinan Lai and Andy Carpenter. Static Analysis and Testing of Executable DSL Specification. In: Proceedings of MODELSWARD2013, Barcelona, Spain, 2013. INSTICC.

1.5 Thesis structure

This thesis is organised in three sections:

Background

Chapter 2 provides an introduction to the foundations of domain specific language development. It also includes the literature review of the current DSL development approaches. Finally, the model quality of DSL specifications are discussed and the characteristics of quality DSL specifications are analysed.

Chapter 3 provides an introduction to the model-based technologies that are used to build the language specification framework.

Design and development

Chapter 4 gives an overview of a Framework for Quality Language Specification Framework (FQLS), which includes the software framework and the software development process.

Chapter 5 introduces the language definition approach of FQLS and the definition layer of the FQLS by guiding the readers through a running example.

Chapter 6 illustrates the analysis layer of the FQLS. It summarises the static checks that FQLS can perform and introduces the development of the checkers. It then explains how the FQLS enables the reuse of fUML checking approaches by a model-transformation.

Chapter 7 introduces the execution layer of the FQLS and discusses the way in which the language specification is translated into Java code.

Demonstration and evaluation

Chapter 8 demonstrates the contributions of the thesis by applying it to defining a language specification for BPEL.

Chapter 9 evaluates the FQLS by using the BPEL case study, conducting experiments on evaluating different parts of the framework.

Finally, Chapter 10 concludes the thesis and proposes further work.

Chapter 2.

Domain Specific Language Foundations

This chapter introduces the foundations of domain specific language development. Section 2.1 explains the basic elements of any computer language specification. Thereafter, Section 2.2 introduces the process of domain specific language development. Section 2.3 reviews the technologies used, especially those for defining behavioural semantics. Section 2.4 considers the characteristics of a DSL specification and summaries the desirable features of a high quality language specification.

2.1 Programming language specification

A widely supported programming language may have many code editors and compilers that have been developed by various parties. However, all these tools support the same language syntax and semantics, because all these tools are implementations of the same language specification, as illustrated in the bottom of Figure 2. A language specification is an abstract definition of a programming language that covers one-or-more *concrete syntax* (what the user writes), *abstract syntax* (what the user expresses), *behavioural semantics* (how it is interpreted) and possibly the mappings thereof.

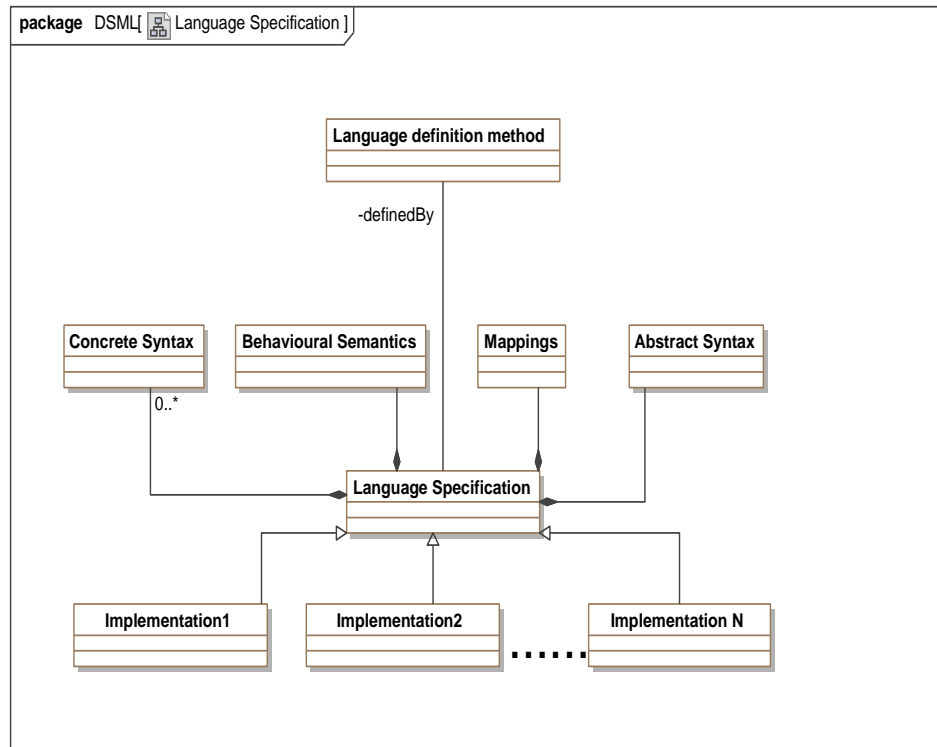


Figure 2: Components of language specification.

Figure 2 also illustrates the relationship between the language specification and the language definition method. The concrete syntax, the abstract syntax, the mappings and the behavioural semantics form a language specification. The language specification is still needed to be expressed by some languages. The language that defines languages is a language definition method. The following subsections introduce the components of a language specification. After that, the language definition methods are reviewed in Section 2.2.

2.1.1 Concrete syntax

Every language is represented by notation; this is true of both natural languages and programming language. The symbols and their possible sequences constitute the concrete syntax of the language. One thing to note is that a concrete syntax can be textual or graphical. It is common for textual concrete syntaxes to be defined using context-free grammars, typically Extended Backus-Naur Form (EBNF). Another thing to note is that a language may have several different concrete syntaxes. For example, RELAX NG [146], which has an XML syntax (suitable for processing by applications) and a compact syntax (suitable for human interpretation); this situation also occurs in OWL [124] and BPEL [147, 141].

2.1.2 Abstract syntax

Abstract syntax defines the concepts of a language and the relationships between these concepts [25]. In traditional language design, an abstract syntax has a one-to-one correspondence with a concrete syntax and it is defined by the BNF for the concrete syntax. More recently in language engineering, it has become common to define an abstract syntax using a meta-model [112]. There are several possible notations for defining meta-models one of which is shown in Figure 3; although on first glance this looks like a UML class diagram, each ‘class’ represents a concept present in the abstract syntax and the ‘associations’ relationships between these concepts. The process of linking an abstract syntax to a concrete syntax is done automatically, either by a parser or by transformation.

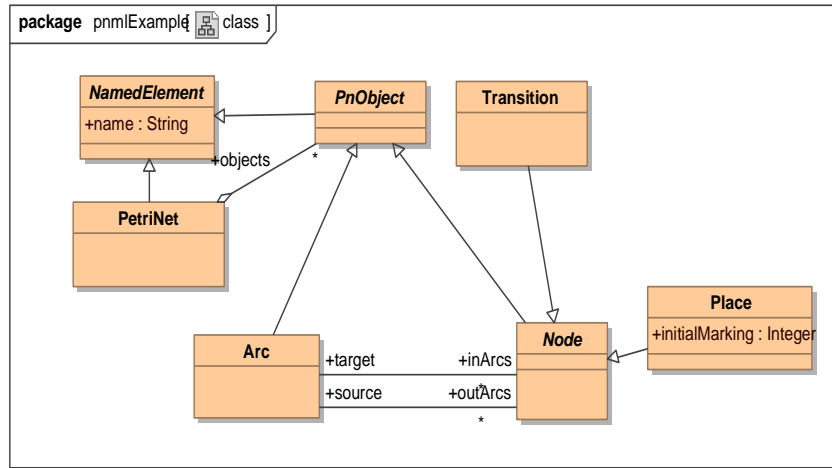


Figure 3: An example of a meta-model.

Associated with the abstract syntax are well-formed rules. These invariance or constraints are often specified by a constraint language, such as OCL [46], and are sometimes called static semantics. In this thesis, well-formed rules are treated as a part of the abstract syntax, and the word semantics indicates behavioural semantics, unless otherwise specified.

2.1.3 Behavioural semantics

Language semantics describes the meaning of concepts in a language. In natural languages, semantics can be defined as a mapping between the language concepts and concepts in the real world. The semantics of a programming language also defines the meaning of that language. It is still possible to define the semantics as mappings if such a mapping exists; however, computer scientists are more interested in how the programs executes in order to build an execution tool according to the semantics of the language.

In the domain of executable programming languages, the semantics of a language is how the programs written in that language would execute on a computer [137]. This is also called behavioural semantics, execution semantics or dynamic semantics. This thesis choose to use the term behavioural semantics for representing how the programs are executed.

Zhang and Xu [161] summarised three ways to formally define semantics, namely *operational*, *denotational* and *axiomatic*. Denotational semantics gives the meaning of programmes by defining mathematical functions or denotations that map the abstract syntax of the programme to a semantic value. It corresponds to the construction of a compiler. The compiler generates target code that is assumed to have meaning in the target machine.

Operational semantics describes the meaning of a programme as a sequence or as the execution history of state transitions. The sequences are specified as operational steps of an abstract machine. Operational semantics correspond to the construction of a programme interpreter; the execution steps describe how the program is interpreted in a machine, using the same notations as the abstract syntax. The most widely used notation for defining operational semantics is Plotkin's SOS approach [121], which uses mathematical language to capture the transition rules.

Finally, axiomatic semantics use a different mechanism, which does not directly define a method that could be used to execute the programme, but defines a set of assertions that express the correctness of the programme or the equivalence of the programmes.

In the development of semantics for GPLs, the approaches used are all mathematical notations. This is because these rigid and concise specifications enable analysis, they are understandable to GPL developers (who are mainly computer scientists who are used to working with formal notations) and because it is possible to abstract GPL semantic concepts in mathematics without losing meaning. However, the same principles do not hold for DSL semantic development, as will be discussed in the next section.

2.2 Domain Specific Language development

This section reviews the basic concepts and methods for DSL specification development. Subsection 2.2.1 discusses what is the contents involved in a DSL development process. Subsection 2.2.2 reviews the traditional methods for DSL

development. After that, Subsection 2.2.3 introduces the DSL development by a model-driven approach.

2.2.1 DSL language specification

The content of a DSL specification depends on the purpose of the DSL specification. There are two examples of the purposes. A DSL specification can purely defines the language itself; or it may contain sufficient details that allow various tools to be built as well, for example, executors or verifiers. In the latter case, a DSL specification may contain examples and tutorials in addition to language syntax and semantics.

Multiple tools can be built as implementations of the same specification. In order to allow this, a form of platform-independence is needed in a DSL specification. In addition, a level of formality is needed in the specification. It is not a requirement to build the language specification by mathematical language; however, the language that used to define the DSL specification must prevent different interpretations of the same language specification.

Kelppa [80] summarises that there are two distinct phases in the life cycle of a DSL. In the first phase, the language is designed, which produces a language specification and usually also provides reference implementations of support tooling. The other phase is the use of the language. By using the specification and the tools created in the first phase, domain experts create DSL programmes.

These two phases involve three different roles, as shown in Figure 4, which are the language engineers who create the language specification, the language tool developers who create supporting tools, and the DSL users. During the design phase, it is common for the roles of language designer and tool developer to overlap.

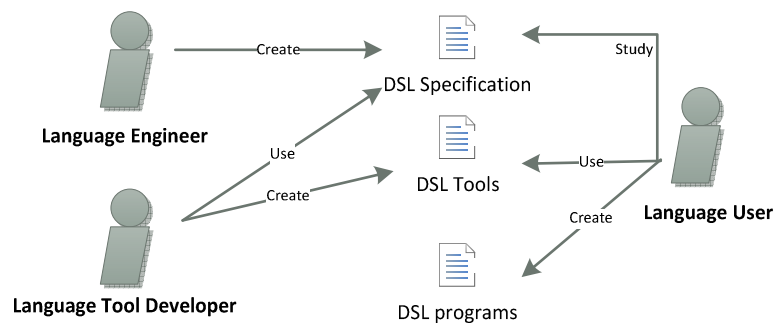


Figure 4: Roles and products in the DSL development process.

As Figure 4 shows, language specification is at the top of language development. It is the basis of language tools and provides instructions for using the language. The main aim

of language specification is to be interpreted either manually or automatically by language tool developers and users. To allow this interpretation, the specification must be understandable to people not familiar with special language design notations. However, because the correctness of the DSL tools and DSL programmes depends on the correctness of the specification, the language used to define the specification must allow at least a basic validation of the specification. Since the language specification is used in various roles, understandability and correctness are the two main aspects of the quality of a language specification.

When talking about understandability, it is necessary to discuss the knowledge that the roles involved in the DSL development process. Language engineers understand the process of MDE; therefore, they can use the technologies in the MDE domain. The language tool developers know the principles of MDE; however, they may not be familiar with all the technology in the MDE domain. For example, the developers that create DSL code analysers may only need to use a GPL and use API to parse serialised models directly into Java classes. Thus, they do not necessarily know how to perform a model transformation. Here, the only assumption of the language tool developers is that they have a general computer science background, and they know programming and UML.

The language users are domain experts; thus, they are seen as not have any programming background. However, since a large amount of DSLs are designed for the domain of software development, the DSLs are used by developers. As a result, if a language user seeks ultimate guidance from the language specification, the user must be an *advanced user* and must be confidently working with UML.

2.2.2 Traditional ways of DSL development

GPLs usually use a context-free grammar for defining the syntax, which results a unified abstract and concrete syntax definition. Conversely, tools such as executors are built from scratch. DSLs share many similarities with GPL, although it has distinct features, which are listed as follows:

- A small number of users means that it must be developed at a low cost and reuse tools rather than building tools from scratch [71].
- The syntax prefers graphics, or multiple syntaxes [75]. An example is RELAX NG, which has an XML-based surface syntax and a compact syntax that is easier to be understood by humans.

- The developers of DSLs can be computer scientists who are equipped with the relevant knowledge of language engineering but, in MDLE, they are more likely to be developers of normal software [79].
- The DSLs can evolve in fast-pace, while GPLs are slow and often standardised [156]. This requires the necessary support of modification of the language definition.

Many traditional ways of building DSLs aim to provide tool reuse. One widely used method is to define the DSL using the *pre-processing* ability of a GPL; hence, the existing tools for the GPL can be reused. An example is defining the DSLs as macros, as in the case of SystemC [113], which used the existing pre-process mechanism of C/C++ to generate the target language. Another group of examples includes the approaches that use the syntax extension mechanism of functional programming languages (LISP or Groovy). The functional programming paradigm can be used to define grammar, whereafter a DSL programme can be written as a valid programme in the functional programming language.

The pre-processing approaches could reuse the existing target language platform. On the other hand, this also means that the DSL is platform-specific to the target language. The syntax of macro-based DSLs is limited by the target language, which does not support multiple types of syntaxes. It also requires the designer to know the target language well, because expanding macros can result in unexpected occurrences [92].

Another way of developing DSLs is by using *parser-generators*. Parser-generators could be used to derive a parser that transforms the concrete syntax of the DSL into an abstract syntax tree, such as JavaCC² or ANTLR³. It is possible to use the same technology to build compilers or interpreters for GPLs. Parser-generators are also limited to textual notations. Moreover, because DSLs have a fewer users than do GPLs, building tools for them from scratch is costly.

In order to advance the use of parser-generators, language working benches like ASF+SDF [17], LISA [109] and the MOSES [36] framework, are special software tools that aim to ease the process. They could use BNF for building syntax and they can generate code-formatter or even executors from formal specifications. While these approaches can build DSLs, because they are built on special platforms, the language

² <https://javacc.java.net/>

³ <http://www.antlr.org/>

specifications are difficult to reuse and are not interchangeable. As a result, these approaches have limited usage and seems out of date [40, 109].

Another embedded approach is to use the target language only as a tool for parsing; for example, using a restricted general-purpose language for representing a DSL (for example, [94] uses Smalltalk). This approach can be seen as giving an existing language new semantics.

Although the traditional ways of developing DSLs support the reuse of the tools, they face the challenges listed below:

- They bind the language specification to textual, single concrete syntax languages.
- They link the DSL to a particular GPL or to a language development framework.
- They require advanced knowledge of the target GPLs in order to develop the macros.
- They combine the specification with its implementation, which may cause the absence of a language specification. In this case, the only way to understand the semantics of the language is to inspect the implementation of the language.

2.2.3 Developing DSL by a model-driven approach

Model-driven approaches take models as the key artefacts in the development process. By building platform-independent models and finally transforming them into working software, model-driven approaches have developed rapidly in the last decade. A detailed description of various model-driven terms is given in Chapter 3. In this subsection, ways in which a model-driven approach could benefit DSL development are discussed.

A model-driven approach analyses the domain of the problem and definitively models a representation of the problem. In the domain of DSL development, a DSL can be represented as a *language model* (as shown in Figure 2), where the meta-model is a model that could define the syntax and semantics of any DSL. The problem of building a DSL then becomes building the language model of the DSL, and this language model is the same as the language specification because a language specification is an abstract model that defines the language.

If a language specification is developed as a language model, then all benefits of model-driven approaches can be achieved. In the domain of model-driven language engineering (MDLE) [25, 43], the limitations of traditional approaches discussed in the last section can be eliminated. The benefits are summarised as the following:

- MDLE defines the abstract syntax of language specifications as platform-independent models, which allows for graphical syntax or for having multiple kinds of syntax.
- The language models are not tied to a particular target GPL.
- By transforming models to various artefacts, the complexity of building DSL tools is reduced.
- For language specification development, MDLE could produce a clear language model that can act as the language specification; thus, readers could research the language models rather than the implementations⁴.

MDLE has become an extremely promising approach. Consequently, many DSLs now have a model-based specification, which could be categorised as follows:

- DSLs that are officially defined as models, for example SysML [57], MARTE [56] and BPMN [52].
- DSLs that are officially defined in other ways, although the working groups are shifting towards a model-based language specification, such as SDL [122] and Petri Net Mark-up Language[14].
- DSLs that are officially defined by other means, but which have a widely used, model-based implementation, such as BPEL (enabled by BPEL designer project) and ASM [86], and ontology definition languages such as OWL (enabled by the Ontology Definition Meta-model [49]).

2.3 Methods for defining DSLs

This section provides a literature review of the state of the art of MDLE. Although the focus of this review is the definition of behavioural semantics, these are linked to the abstract syntax of a DSL. Thus, before looking at behavioural semantics, the definition of abstract syntax is first considered.

⁴ Later, in section 2.3.2, the limitations of MDLE in defining the semantics of a language will be introduced.

2.3.1 Review of DSL abstract syntax definition

In MDLE, as introduced by [136], there are two widely used ways of defining abstract syntax; UML profiles and meta-models. UML profiles use Stereotypes to specialise UML elements to representing domain-specific concepts. Since UML aims at being a generic modelling language, it intentionally retained many semantic variation points. As well as specialising existing concepts, profiles can add constraints or explanations to these. Since the profile mechanism allows the use of existing UML tools, it immediately provides support for the creation of DSL programs. However, UML profiles cannot violate the semantics of UML. Thus, a DSL specification is its DSL profile plus all of UML, which is normally more than required by the problem domain. In conclusion, it is considered [137] that UML profiles often provided inadequate expressiveness and lack precision .

In the meta-model approach to defining the abstract syntax of a DSL, the syntax captured by a meta-modelling language like MOF [51], Ecore or Microsoft DSL toolkit⁵. When using this approach, the specification starts with a ‘blank sheet of paper’. Thus, it is possible to ensure that the final specification only contains elements appropriate to the problem domain. Of course, with this approach no tools directly support the creation of DSL programs. However, tools like Xtext [33] and GMF [39] do simplify the production of such tools.

2.3.2 Survey of semantic description approaches of DSLs

Bryant et al. [19] and Chaudron et al. [23] summarised that defining behavioural semantics is one of the most challenging topics in the MDE domain. One reason is that, unlike abstract syntax that has widely agreed means of defining it, there is no unanimous way of defining behavioural semantics. In principle, the means of specifying the semantics of DSLs are the same as those of specifying the semantics of GPLs. The existing approaches for defining the behavioural semantics of DSLs can be roughly categorised as *translational approaches* (which involves a mapping from the language syntax to semantic domain) and *operational approaches* (which defines the semantics as operational rules).

2.3.2.1 Translational semantics

Translational semantics [40] explains the behavioural semantics of DSLs by translating the DSL concepts to another semantic domain of which semantics are well defined, and

⁵ Now known as Modeling SDK for Microsoft Visual Studio.

then uses the concept from the semantic domain to explain the semantics of the target DSL. It is also called semantic anchoring [24].

The components of translational approaches are illustrated in Figure 5. Translational approaches involve the abstract syntax domain, the transformation, and the semantic domain. The concepts from abstract syntax domain are translated to similar concepts in the semantic domain. These translated concepts are used as the structure of the semantic domain. The behaviours of the target DSL are then developed using the concepts in the semantic domain. Hence, it is possible to build a programme in the semantic domain that represents the behavioural semantics of the original language. The program can be executed by the executor for the semantic domain. By executing the program in the semantic domain, how the DSL program should be executed in the abstract syntax domain can be understood.

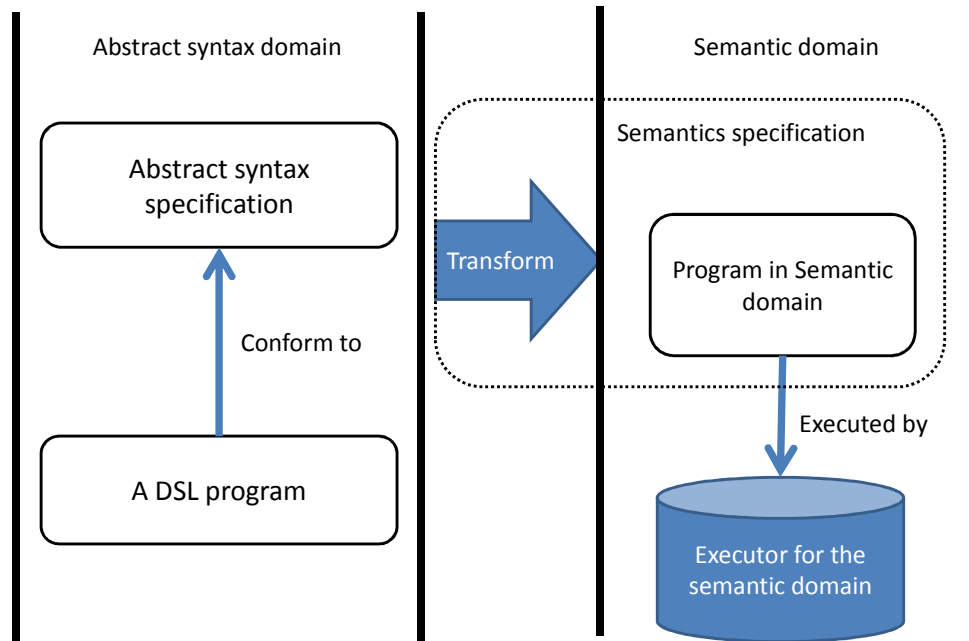


Figure 5: Components of translational approach.

Translational semantics are similar with denotational semantics. However, there are several differences.

- Denotational semantics gives programming languages meaning by translating language concepts to another domain. However, the GPLs already contains sufficient details to create a transformation that can generate a complete program in

the target domain. In this case, when referring to the 'semantics specification', the specification only contains the transformation. In contrast, a translational semantics specification usually involves a mapping and additional programs that are written in the language of the semantic domain.

- Denotational semantics for GPLs usually use mathematical notations and the aim is to explain the compiled results in a formal way. Implementations of the semantics compile the language source code to assemble the code. By contrast, translational semantics usually selects a semantic domain that is at a higher level than assembly code. The selection of semantic domains usually has a purpose, so the existing analysis approaches and tools of the programmes can be reused. Because the aim is to reuse, semantic domains are usually executable programming languages.
- The mapping of the DSL concepts to the semantic domain concepts is usually done automatically. Even if it is not, there are ways of making automatic translation possible, usually by using a model-to-model or mode-to-text transformation.

Abstract State Machines (ASMs) are widely used in describing language semantics. In this approach, an ASM programme first defines an abstract data structure of the state machine, and then defines a set of transition rules. In essence, a transition rule is like the pseudo code '*if condition then update states*', where 'updates' are a set of state modifications. ASM successfully defines the semantics of some widely used, general-purpose languages, such as C#, Java, VDM, etc. According to Gurevich et al. [60], it is sensible to adapt the ASM method for DSLs.

Semantic anchoring [24] is the initial attempt to formalise DSL semantics using ASM. It uses AsmL, which is the ASM implementation in Microsoft Visual Studio, as a semantic domain. It shows how to identify the so-called 'semantic unit', which can be reused for different DSLs. However, the process of finding semantic units is somewhat non-systematic and ad hoc.

Gargantini et al. [40] extended the semantic anchoring approach by defining higher level semantic anchoring in model-based technology. Compared to semantic anchoring, Gargantini et al. used meta-model based technology and the semantic domain is ASMETA [86], which is an open-source implementation of ASM based on modelling technology. Because the core of ASMETA is a meta-model of ASM, it is possible to establish model-to-model transformations using standard model technology. Equally important, Gargantini et al. tried to formalising the process of establishing mapping between the abstract syntax

and ASM. They developed mapping between MOF and ASMETA; thus, the process of creating mapping is automatically derived.

DiRuscio et al. [31] proposed a similar approach. It tries to extend the AMMA framework, which aims to provide a complete modelling toolset, of which the widely used model transformation language ATL is a part, by adding the ability to write ASM rules. Thus, the behavioural semantics specification can be defined in the enriched ATL programme by in-place transformation. [6, 127] use similar approaches; they focus on evaluating the ASM semantic mapping via DSL case studies that are more detailed.

Maude [108] is a programming language based on rewriting logic, which is widely used in the domain of real-time embedded system specification and simulations that involve real-time and concurrency. [69, 138, 128] selected Maude as a semantic domain. They provided the methodology for translating the abstract syntax of the DSL to Maude data structures. The means of specifying real-time and concurrent concepts were discussed, and they eventually proposed a way of integrating a Maude-based model verifier to their frameworks.

There are also other semantics domains in order to perform various analysis. For example, Hahn and Fischer [61] propose using Z-Object language to specify the semantics of the DSL. Kelsen and Ma [77] use Alloy as the semantic domain. If one considers GPLs to be a semantic domain, then works such as [74], which generate code from the language specification, can be seen as translational approaches.

By translating to semantic domains, translational approaches can reuse the editors or checkers in the semantic domain to maintain the correctness of the specification, and translational semantics are usually executable. On the other hand, there are also some challenges of translational semantics, as listed below.

- Consistency. It is possible that abstract syntax and behavioural semantics will evolve separately, causing inconsistencies. If no mechanism is applied for maintaining consistency, it is likely that updating one category will cause the other to expire. Although it is possible to develop a tool that automatically does the cross checking, such a tool is costly in terms of effort.
- Understandability. Readers must understand the abstract syntax, the mapping, and the semantic domain, which requires understanding at least two languages. Considering that translational semantics usually has the purpose of performing a

certain targeted analysis, the semantic domain is usually formal. This is difficult for the language tool developers, and may even be impossible for the language users.

- Interoperability. There are many possible choices for the semantic domain; thus, if two DSLs are defined by different semantic domains, interchange between the languages is nearly impossible and it is not possible to combine these two languages.

2.3.1.2 Operational semantics

Operational semantics originated from the methods of defining semantics for GPLs. It defines semantics as the rules for manipulating the concepts defined in the abstract syntax.

Figure 6 illustrates the components of operational semantics. The semantics are defined as the transitional rules that manipulate the DSL programs by using the concepts defined in the abstract syntax. The transition rules can be interpreted without ambiguity by an executor. The executor can then execute the DSL programs according to the transition rules. When creating a semantics specification by operational approaches, it is the transition rules that are actually created.

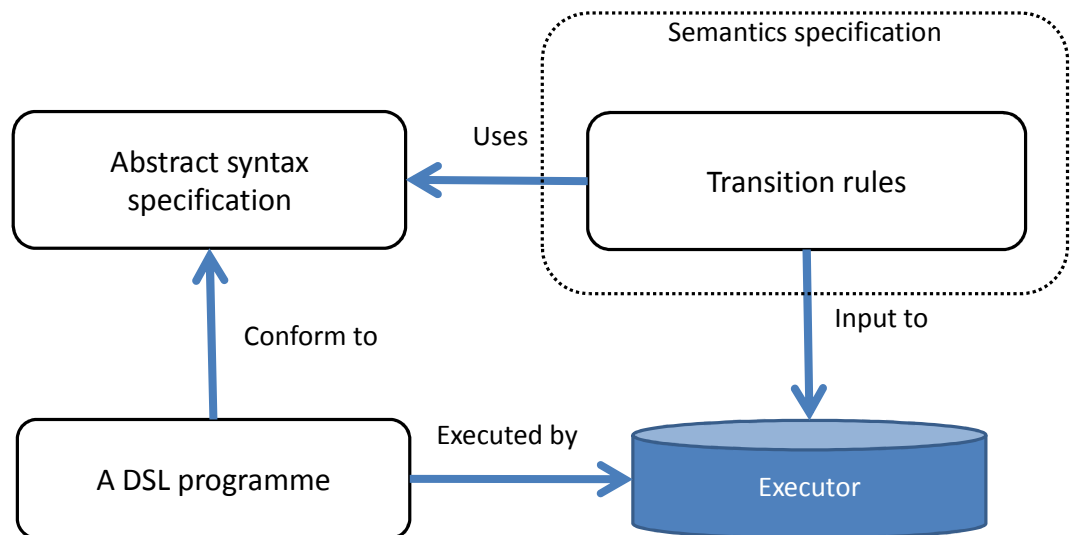


Figure 6: Components of operational semantics.

The transition rules in operational semantics still need a method of specification. Unlike GPLs, the transition rules of which are captured using mathematical language (usually the notations used in [121]), DSLs often select an understandable language to express transition rules. There are two ways of defining these transition rules. The first is *rewriting approaches*, which define transition rules using a rule rewriting language that could directly manipulate the DSL instance model. The other is *weaving approaches*, which use an action language that is capable of defining both abstract syntax and behavioural semantics.

Rewriting approaches

As shown in Figure 7, the rewriting approach requires two languages, one for defining the abstract syntax of the DSL and the other for defining the rewriting rules. The rewriting rules can manipulate a DSL programme directly, thus expressing the semantics as transformations.

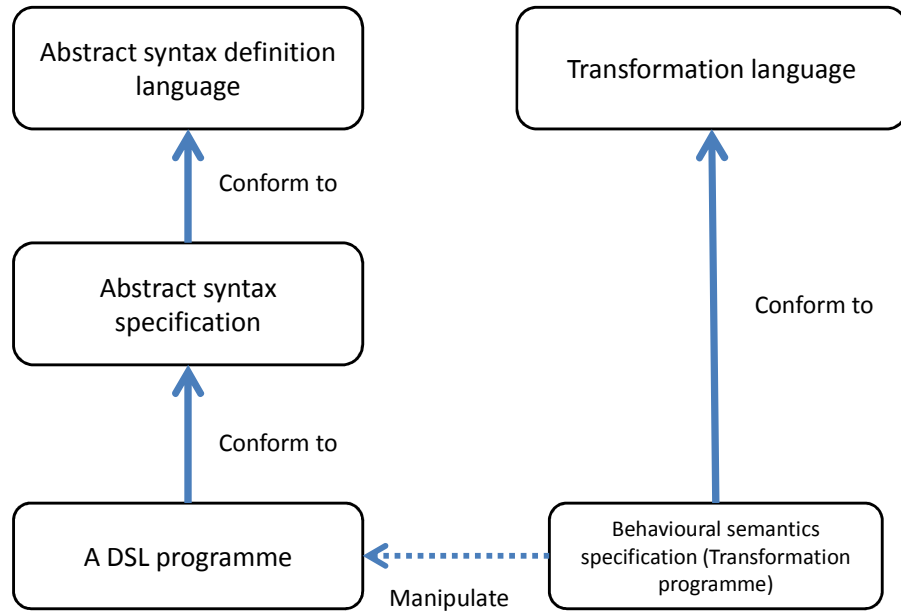


Figure 7: Languages used in rewriting approach.

There are many options for defining the rewriting rules. Graph transformation approaches, such as that of Biermann et al. [13], the Dynamic Meta-Modelling approach [35, 9], and de Lara and Vangheluwe [30], are examples. They use a graph transformation tool that could transform the DSL instances. The graph transformation tools include AGG

[107], Groove [41] and Henshin [7]. Because the transition rules of graph transformation approaches are graphic, these approaches are preferred by domain experts who lack a computer science background [142, 149]. On the other hand, graph transformations for defining semantics of DSLs also pose many challenges. Rensink [125] summarises these as graph transformations have problems of scalability (when a graph is too large to show), maturity (theoretical maturity and technical maturity) and that they lack of tools.

In model-driven development, model-transformation languages can also be used for defining behavioural semantics. Examples include Baar [8], Sadilek and Wachsmuth [131] and Wachsmuth [157], which all use the QVT language. These approaches are strong in terms of interoperability, because QVT is the standard transformation language.

No matter which language is used as the rewriting language, rewriting language still defines a DSL specification as using two languages. Maintaining the consistency between the models and the transformations in an evolving system is still a challenge, and the language engineers still need to learn two languages.

Weaving approaches

The other way approach is to weave behaviour into the language model. The abstract syntax model contains classifiers, the relationship between classes and attributes. The operations in a meta-model usually act as abstract placeholders. Weaving behaviour approaches use the operations as an aspect-joint point in order to add detailed behaviour to the meta-models by using an action language. These action languages have the ability to represent both abstract syntax and behavioural semantics; therefore, the language specification becomes a programme of the action languages, as shown in Figure 8.

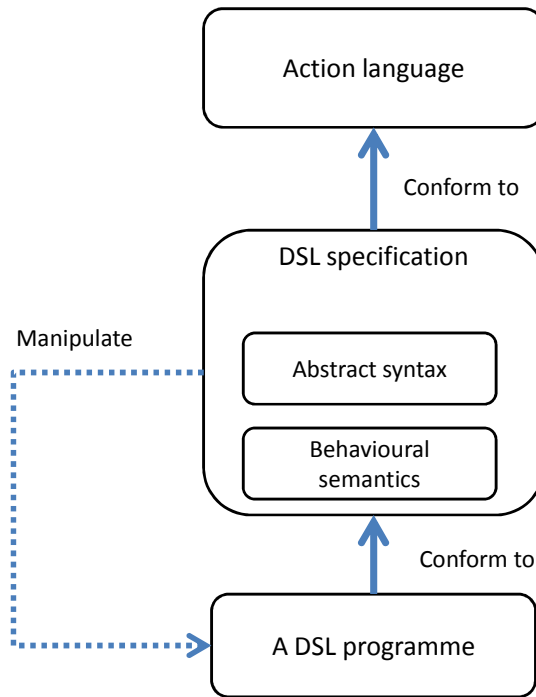


Figure 8: Languages used in weaving approach.

Action languages can be languages that are designed for model management, such as XOCL [25], Kermeta [18], Scheidgen and Fischer [132] or Epsilon [81]. These languages have their own features. For example, because Kermeta supports aspect-oriented programming, it works seamlessly with the Ecore model. Scheidgen and Fischer [132] uses graphical notations that enhance understandability, and Epsilon [81] is a platform whereby all management languages share syntax and a semantic basis. On the other hand, various action languages share similar benefits, as listed below.

- High understandability. The human consumers of a semantics specification often have a programming background which means that they prefer UML. Using an action language to manipulate instance model concepts is readable for this purpose.
- Elimination of inconsistency. Compared to translational semantics, in which the semantic mapping and consistency must be managed, weaving approaches do not have a separate semantic mapping; thus, there is no need to manage it. As it uses a unified approach for both the syntax and the semantics, inconsistencies that occur in translational approaches, such as outdated references, will result in a syntactic error in the language specification.

Similarly, they also share the same limitations.

- Lack of interoperability. It is impossible to share or combine languages that are defined in different action languages. These action languages, although they are maintained and are actively developing, are not standard technology because their use is not widely agreed upon, nor are they widely used.
- Limited expressiveness. The action languages are usually similar to GPLs. They need a starting point, such as a main method to execute. In addition, existing weaving approaches do not natively support concurrent behaviours; thus, the specification of current behaviours requires using the language to define a concurrent programming architecture.

UML as a weaving approach

In order to address the limitations of weaving approaches, researchers look for inspiration from UML. In MDLE, meta-modelling languages such as MOF and Ecore are the *de facto* standard for abstract syntax. There is a close relationship between meta-modelling language and UML, as MOF and Ecore are a subset of UML class diagram. It is considered promising if a subset of UML could be used for defining both syntax and semantics, since UML already has the ability to define behaviours. The challenge is that UML does not have formal execution semantics – the semantics of UML are defined by prose, with indented semantic gaps.

Sunyé et al [150] proposed using UML Action Semantics [44] as an action language for defining semantics. Since UML action semantics is an OMG standard, it could introduce interoperability into weaving approaches. Although the action semantics approach was proposed a decade ago, the use thereof is very limited. The following challenges may explain why UML Action Semantics is not suitable as a standard method of defining language semantics.

- As it was built upon UML 1.4, the syntax of UML changed dramatically from 1.4 to 2.x.
- It does not provide concrete syntax; instead, one action language needs to define the mapping between the meta-model of the action language and the meta-model of UML action semantics. Since there are many action language and none of them have obtained universal support, it is challenging to use it to create an understandable behaviour model.

- UML Action Semantics still lacks formal semantics, because its base semantics are described by natural language, which makes performing a formal analysis a challenge.

The semantics of the foundational subset of UML (fUML) [55] have been published by OMG, which is a promising technology for addressing the existing problems of the weaving approach [137]. This thesis seeks a new DSL definition method by using the textual notation of fUML. In the same way, XMOF [96] shares similar opinions regarding the method of semantic definition. However, XMOF seeks to combine MOF with fUML by building a new modelling language that shares MOF and fUML concepts. XMOF and the method described in this thesis, although sharing some technological basis, can be distinguished by many aspects, such as the manner of defining models and the method of executing the DSL.

Summary

Translational and operational approaches can define a language specification formally, thus eliminating the limitations of a prose-based specification. However, each of the methods has certain limitations, and these limitations hamper the quality of the language specification.

2.4 Quality of DSL specifications

Many DSLs already have, or are evolving to, a model based specification, and much research focuses on the ways of defining a DSL specification that enables a particular analysis purpose. However, these approaches do not emphasise the quality of the specification. This section provides an introduction of the quality of models. Subsection 2.4.1 introduces the goals of model quality. After that, Subsection 2.4.2 reviews the practices to improve the quality of models. Finally in Subsection 2.4.3, requirements of a high quality language specification are proposed.

2.4.1 Model quality goals

Since the final products are derived from the initial models, the quality of these models could affect all products that are derived from them. Models can be checked at an early stage, enabling any errors found to be corrected during the design process, at which time it is easier to identify them than after they are implemented. Language specifications are also

models, so the quality of a language specification shares similarities with the general model quality features. However, assuring the quality of the language specification is challenging, particularly considering the errors in widely used DSL specifications, such as UML and SDL.

According to the literature [101, 103, 57], it is possible to classify the goals of model quality into two categories.

The first goal is correctness. The models, as an abstraction of a real system, do not violate any properties that exist in the real system. This means that the model must not specify something that is wrong. [101, 103, 57] separate this goal into correctness and consistency. Correctness means the model is syntactically correct (well formed) and semantically correct (the logic and relationship of the abstraction conform to the real system). Consistency means ensuring that different models of the same system do not contradict each other, which can be seen as a special property of correctness. In the same way, incorrect models can be seen as inconsistencies between the model and the meta-model. Regardless of the definition, a correct model is the pre-condition for applying model-driven activities, because an incorrect model will lead to incorrect systems.

The other goal is communication. Models are read by both humans and machines; thus, they must be able to fulfil the purpose of communication. This means that models must be captured by a language that is understandable by the intended users of the models, and is interoperable between its users and tools.

2.4.2 Practice to improve quality of models

The quality of models seriously affects the quality of the final products. However, model quality is not easy to monitor, since many factors can affect the quality of the models. Factors such as the choice of an improper modelling language, limitations of tools and developer knowledge, as well as quality assurance techniques, can affect the quality of the models. Nelson and Monarchi [105] identified that most of the MDE processes lacked quality assurance activity. The lack of quality assurance restricts the quality of the models. Lassen and Aalst [89] identified that complex models can cause many problems, including design flaws, and are difficult to implement. Thus, controlling the complexity of models can enhance the quality of the models.

The practices for model quality assurance can be roughly divided into two categories, *processes* and *automation* [103]. Processes approaches create a modelling process that the modellers need to follow, which can include many factors

- The particular activities the developers need to follow, or the activities they need to enforce, include code review [98], using Agile [4], building reference specifications [26] or using a test-driven approach [142].
- Guidelines and conventions, such as forcing the developers to use a particular modelling language (for example, UML) or a programming paradigm (for example, Cariou et al. [22] forces design by contract principle), documenting best practices and error-prone places.

Automation is another way of maintaining the quality of the models. Automatic error detection and testing are widely used and effective ways of programming, and modelling is no different. It is more difficult to apply automation in models because, when testing or simulating models, the models need to be executable and to have a good execution tool.

There are many works aimed at automatic model analysis that transform models to a formal domain, which use reasoners or model checkers for the analysis. These include [119, 126, 28] for checking reachability and contradiction, [117, 118] for checking executability⁶ for detailed behaviours, and various approaches for checking the consistency of UML [34]. While these approaches are useful for checking the special properties of the models, they are limited by the reasons listed below.

- Using a reasoned or model checking technology faces the problem of the state explosion problem, which makes checking large models difficult.
- It involves knowledge that normal developers do not have.
- A way of tracing the errors back to the original model is still needed when errors are identified.

The formal approaches uncover various kinds of advanced errors in a language specification. Features such as reachability are important. However, the current practice fails to reveal extremely basic errors, which should have been considered in the first place. Wilke and Demuth [160] and Glaser et al. [42] not only reveal the large number of errors in the language specification, they also identify that these errors are not tricky errors, but most of them are inconsistencies or syntax errors. Consequently, it is more desirable to use a method that could identify these simple errors.

⁶ Here, the concept 'executability' is not whether the language specification can be executed as a programme, but to whether the system is still in a valid state after the execution of a method.

Static code analysis tries to find bugs or errors without executing the program. Researchers prove that they increase the quality of the software [70, 27]. It could involve using a type system, applying analysis based on bug patterns, restricting code styles, or involving use of theorem prover or model checking.

2.4.3 Requirements of high quality language specifications

While language semantics specifications are transforming from a prose-based specification to formal specifications, a definition method that can produce such a specification must have the following features:

Consistency

If a language specification is defined in different languages, these different parts must not be contradictory when several parts are combined as one language specification. The consistency discussed in this thesis is also referred to as horizontal consistency in some of the literature [103]. Horizontal consistency may occur when the language specification is defined by different languages or models, as each part is correct, but the concepts that cross paths are used incorrectly. For instance, when using translational semantics, the structure of the semantics specification is derived from the abstract syntax specification; however, as they have both evolved, the semantics specification may refer to an element in the syntax specification that has already changed name, or which does not even exist.

Consistency is an important factor in a high quality language specification. Thus, a language specification method must design a method to ensure the consistency of the language specification.

Correctness

The correctness of the language specification is important, because the audiences rely on the specification to solve ambiguity, and machines use the specification as input for other products. Any flaw in the language specification will confuse the audience, and the flaw will propagate to any other products that rely on the language specification.

It is possible to categorise the correctness of a specification as *syntactic correctness* and *semantic correctness*. If a language specification is syntactically correct, this means that it does not violate the constraints of the language definition technique. For example, if the abstract syntax of a language is defined as a model, it must be a valid model of its meta-model. Another example is when the semantics of the language are defined in a particular language, the semantic specification itself must be a valid programme of that

definition language. Syntactic correctness is the basic condition that a correct language specification has to meet.

The other category is semantic correctness. The specification must really reflect the language designers' intentions. A semantically incorrect language specification may be a syntactically correct specification; however, such a language specification describes an incorrect language for the application domain.

The syntactic correctness of a language specification can be ensured by applying automation methods, such as developing a tool for checking syntactic errors. Semantic correctness of the language specification is not easily checked by a single tool, since the semantic correctness of a language is subject to human interpretation. A software process that combines inspection and testing is one way of ensuring semantic correctness.

Executability

The syntactic correctness of a language specification can be automatically checked, and errors will be reported to the language designers. However, the semantic correctness of a language specification is not easy to check, as it involves the same difficulty as debugging a programme that has been successfully compiled. Thus, testing is the most effective way of checking semantic correctness. Testing requires the language specification to be able to act as an executor of the language, which can load an input programme and output the results. In addition, executability is also expected when working with domain experts, as the domain experts judge whether a programme ends with the expected behaviour.

Understandability

A language specification must be widely accessible; this is why many language specifications use natural language, because anybody who can read can access the language specification. However, such a specification usually cannot prove or disprove one example of a programme as being correct or wrong, nor can be automatically processed. On the contrary, a specification that uses pure mathematics makes it possible to prove correctness, but is not accessible to the audience of the specification. Much of the literature [137, 65, 79] suggests that mathematical definitions are not preferred due to a lack of knowledge. Thus, a language definition technique must balance the formality and the accessibility of the specification.

The primary audience is the language engineer, who wrote the specification, and language tool developers, who use the language specification as their product input, and advanced language users, who are the users that are interested in the language specification

itself. The normal users may also be interested in the language specification. However, a language specification is not the best way to learn how to use a language. Thus, domain experts who do not have any programming language experience are not the target audience. The language engineers, language tool developers and advanced users have different knowledge, but it is reasonable to assume that they have some experience in UML, model-driven technology and object-oriented programming.

Expressiveness

The method of defining a language specification can also be seen as a domain-specific language that targets defining languages. In such a situation, the language engineers are the domain experts for defining languages. The ability of the language they use affects their ability to design high quality language definitions.

Expressiveness is another important feature of a language that is used to define language specifications. An expressive language specification technique should be able to define common semantics in a concise and precise way, while a non-expressive approach may still be able to define the same semantics, albeit in a difficult way. Meanwhile, the expressiveness of a language is not an absolute concept. While graph transformations can naturally support concurrent executions, they do not support express rewritings that have complex conditions in a precise way.

Model-based specification

Language specification must be model-based, using meta-modelling technology, making it possible to achieve the various benefits discussed in Section 3.1.

Interoperability

Bryant et al. [19] discussed another desirable feature called *dissemination*, which is the ability to share the language standard among shareholders. The ability of a language specification to interoperate with other shareholders (either human or machine) is defined as *interoperability*.

Many semantic definition techniques introduced in Section 2.3 have a special language or method to define the language concepts. Specially designed languages, although making a particular analysis possible, lack interoperability. A language specification is interoperable if the specification is defined using widely accessible technology. Such technologies are usually backed up by an international standard consortium, such as the OMG or ITU, or a large community, possibly an open source community, or are led by a

large industry player. Interoperability is an important aspect of a language specification. As it is an increasing requirement of composite DSLs, it must be accessible by other tools.

- Using an interoperable standard notation to capture the models makes the models understandable by a larger community.
- An interoperable specification can be accessed by other tools. A language specification is read by various actors in the development lifecycle of a DSL. If the specification is interoperable, different users can use their tools to access the language specification.
- In the same way, considering the frequent requirement to transform a language specification to other representations for purposes of analysis, such a transformation may already be available if the specification is interoperable.
- Interoperable specifications make composite and reusable DSLs possible. The ideal scenario is that, if a DSL is defined using an interoperable approach, when the language engineers implement another DSL that shares domain concepts or semantics, the language specification can be reused. For example, if XPath is defined using an interoperable approach, then all XML-based languages contain an XPath expression, as their query language will benefit from the standard specification by reusing it.

A development process for DSL

A description method should be accompanied by a process and tools. The described technique can affect the quality of the specification. Although a method may support high quality language specification, it is still dependant on the language engineer's work to design, analyse and implement the specification. A unified syntax and semantic specification can allow consistency and correctness checking to be done in an easier way, but such a task still needs to have a means of being performed automatically. Meanwhile, the language developers need assistance to carry out the language development process. Such assistance can be via conventions (general guidance, including rules, styles and design principles), methods (which recommend following a sequence of steps to gain quality), or a framework (which is a software or a tool chain that provides general functionality for performing the tasks in language development, and which can be extended by developers). All this assistance can be summarised as a software development process for language specification.

In summary, a model-based, interoperable semantic description technique that is unified, executable and expressive is needed. Such features make building consistent, correct and understandable language specification possible. To support this language specification method, a software development process and a software tool chain also need to be built.

2.5 Summary

This chapter introduced the foundational concepts in domain specific language development, and presented a literature review of existing approaches for defining domain specific language specification. Starting from the concept of a programming language specification, it is explained that such a specification includes concrete syntax, abstract syntax and behavioural semantics.

Following this, the concept of domain specific language specification was defined. The actors involved in the development process of DSLs were defined as the language engineers, the language tool engineers and the users. The expectations of a high quality DSL specification were discussed.

The literature review of existing approaches was presented as traditional approaches and model-driven approaches. Model-driven approaches were categorised as translational approaches and operational approaches.

Finally, by discussing the quality of the specifications produced by these approaches, the thesis proposed that a model-based, interoperability, understandability, correctness, consistency, executability and expressiveness were features of a high quality DSL specification. These features could be achieved depending on what the selection of the language specification method and the degree of support in terms of the development process and tools.

Chapter 3.

Model-driven foundations

Section 2.2.3 discussed what benefits MDLE could bring to language development. Subsequent chapters of this thesis will show how the proposed language specification framework can achieved this. Before doing so, this chapter provides a general introduction to the model-based technologies that are used to build the framework.

As MDLE is a specialist form of Model-Driven Engineering (MDE), Section 3.1 outlines the elements of MDE that relate to the proposed framework. One significant aspect of the framework is the support for defining the semantics of a language. As the approach used builds on work being done to give UML formal semantics, Section 3.2 examines attempts to define formal semantics for UML and Section 3.3 introduces the ALF technology that forms the basis of semantic definitions in the proposed framework.

3.1 Model-Driven Engineering

The widespread use of UML diagrams in software development means that model are widely accepted has having a role in software development. However, often this is as blue prints or diagrams that informally document the design. Model-Driven Engineering (MDE) takes models and makes them first class entities; c.f. objects in object-oriented programming. Developers take the initial models and translate them into models (preferably automatically to reduce the cost) with more concrete (implementation) detail and, finally, these concrete models are used to generate an implementation.

It is widely believed [155, 5, 32, 71, 102, 116, 84] that MDE has several benefits; these include:

- Increased *understandability* and *communication*. The audience for models consist of two groups, namely domain experts and application developers. By providing a vocabulary understood by both groups, MDE provides enhancement communication between these groups. In addition, because a model is not

dependent on users' native language, MDE also enables international cooperation both within and between these groups.

- MDE increases *productivity*. As MDE includes code generation, it can reduce the time needed to produce implementation code. Additionally, as it avoids the copy-paste-edit processes often involved in standard code writing, implementation code has fewer errors and there is a reduction in the cost of maintenance. Further, the shift to developing at a higher level of abstraction means flaws are identified and corrected earlier in the design cycle, with the consequent improvements that result from this.

Despite the benefits of MDE, there are drawbacks. One of these is the upfront cost of building models and supporting tools. Another is the challenge of changing the culture of an organisation to use MDE [71]. There have even been reports stating that MDE does not enhance productivity [32]. However, reports [76, 144, 16] of the successful application of MDE outweigh these drawbacks. Hence, the basic features of code generation from the models are provided in every modern IDE and many companies formally adopted MDE as part of their development process [102, 159].

Having established why a MDE approach has benefits, the following subsections discuss the definition and models (meta-modelling) and how these models are used in MDE workflows.

3.1.1 Meta-modelling

MDE takes models as first class entities. So, what is a model? Seidewitz [135] defines a model as “*a set of statements about some system under study (SUS).*” OMG [45] defines models as “*A description or specification of that system and its environment for some certain purpose.*” Muller et al. [104] summaries nine alternative definitions of models, and suggests there is no common acceptance of what is a model. Although there is no agreement on the definition of models, there is an agreement that models are abstractions that capture something about an application domain.

Part of the previous definitions of models are that they are “*descriptions and specification*” [45] or “*statements*” [135]. This implies the need for a notation to capture/describe models that itself must conform to a definition. In the model-driven world, everything is assumed to be a model. Thus, the notation in which models are defined is also as model. To distinguish the fact that this is “*is a model of a modelling language*”, or

more precisely, a model of models, it is normally referred to as a meta-model. From an MDLE point of view, a meta-model is the abstract syntax of a language; Ramsin and Paige [123] provide a more precise definition: “A *meta-model* is a description of the abstract syntax of a language, capturing its concepts and logic, using modelling infrastructure”. In this thesis, Ramsin and Paige’s definition is used.

3.1.2 Model-driven work flow

The context for an MDE workflow is shown in Figure 9. The goal is to produce the implementation artefacts that execute in the target platform (environment). This is achieved by a ‘top-down process’ of system construction that iteratively creates increasing more implementation oriented models [16].

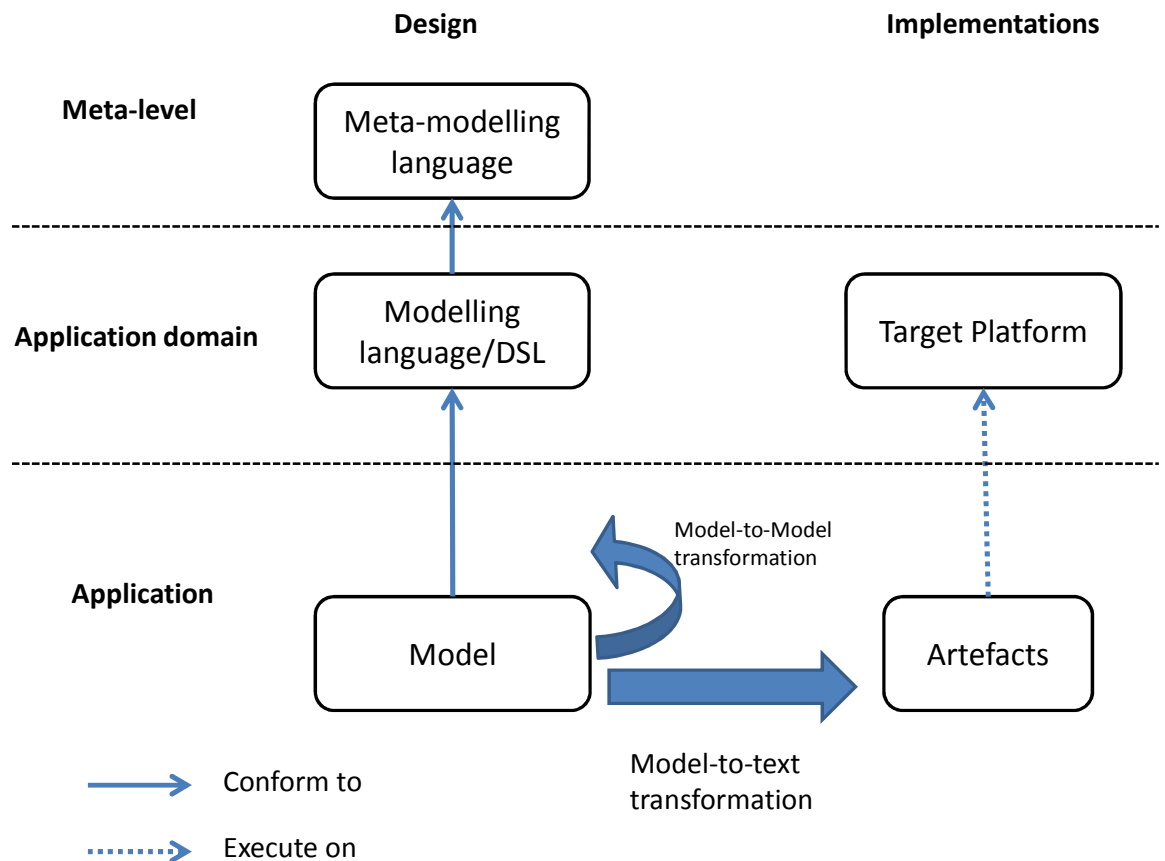


Figure 9: The context of model-driven workflow.

The iterative model-refinement and final artefact generation step of Figure 9 is what is implemented in a model-driven workflow. The Model Driven Architecture [45] proposed by the Object Management Group is the best known realisation of this workflow. Probably the most significant thing about this workflow is that, as shown in Figure 10, it identifies models for different purposes. The development process starts with a Computation

Independent Model (CIM) or domain model that describes the expectation of the system without giving details of how it is implemented. The CIM is transformed to a Platform Independent Model (PIM) that gives more detail about the system without tying it to a particular platform or technology. The PIMs are then combined with marking models that add platform specific configurations to produce Platform Specific Models (PSMs). The PSMs are enriched with further details of the system, including the necessary details regarding to the platform. Finally the system is derived from these PSMs by either code generation or model interpretation.

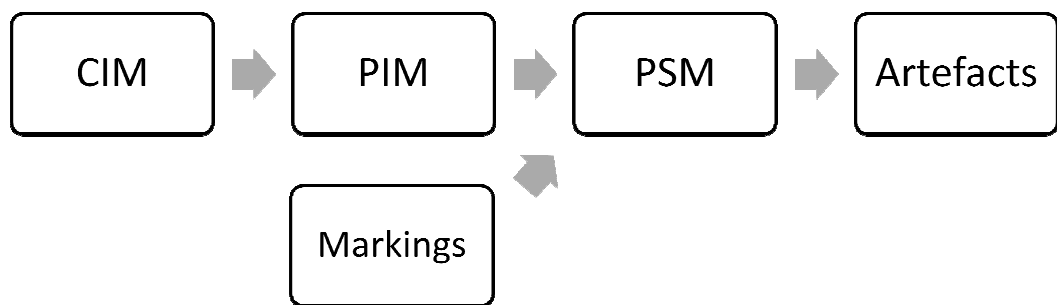


Figure 10: MDA workflow.

3.2 Semantics of UML

Currently the semantics of UML are defined by the English prose of the standard, which allows for ambiguity in human interpretation. There are also semantic gaps, e.g. the semantics of executing actions, which leaves aspects of the semantics undefined and allows UML tool vendors to apply their own interpretations. This section outlines previous attempts to formalise the semantics of UML and examines current work to do this based on a foundational subset of UML.

3.2.1 Previous work to formalise the semantics of UML

A detailed discussion of the need for UML semantics is summarised by O’Keefe [110]. This need means many researchers have tried to provide UML with formal semantics; approaches include the use of Petri Nets [83] and graph transformation [85]. However, as UML is a standard, one of its main purposes is to be interoperable among different parties. Using a particular technology could provide with UML formal semantics and make it serve a particular application. However, if the semantic definition of UML is not agreed upon by

most of the UML shareholders, this still does not solve the problem. UML action semantics [44] tried to define a standard method for manipulating models. However, it still does not define the semantics of UML, but provides a way of defining the behaviour of method bodies in a clearer way.

3.2.2 Foundational subset of UML

As a step towards giving UML formal semantics, OMG oversaw the development of the semantic foundational subset of UML (fUML) standard [55]. This standard provides execution semantics for a subset of UML2. The subset selects the more basic elements of UML2; e.g. classes, common behaviours, actions and activities. The intension is that fUML will provided a shared foundation for building the semantics of higher-level UML concepts.

The subset of elements imported from the UML meta-model defines the abstract syntax of fUML. Its semantics are defined using a two-stage approach; firstly, the semantics of a core fUML, often referred to as base-UML, are created and then these are used as the basis of the semantics for complete fUML. The core covers the very basic concepts of UML, e.g. primitive types, control flows, edges and events. The semantics of this core are defined using Process Specification Language (PSL) [15] axioms; that is, first-order logic assertions and equivalence to the PSL ontology. This means that the semantics of base-UML do not provide a means of interpreting UML models or mapping them to other languages. Instead, these semantics dictate constraints that a legal implementation of fUML must satisfy.

fUML does have some semantic variation points, including time, inter-object communication and concurrency implementation. These variation points allow tool vendors to make their own interpretations when supporting the fUML standard. However, this semantics variation does not prevent higher-level semantics being defined in terms of elements that have this variation. Thus, although semantic variation points could be an issue, in practice they are not. For example, it is possible to specify that actions are executed concurrently, without knowing how the concurrency is achieved (using real concurrency on a multi-core platform or simulated concurrency on a single-core platform).

3.2.3 Summary

fUML became an OMG standard in 2008. Since then, tool vendors, including MagicDraw, IBM, and open-source projects, such as Model-driven solution's

implementation [143] Papyrus, and [91], have developed native support for fUML or compatibility mechanisms. Hence, it is now possible to see fUML as a stable standard. A view reflected in the fact that recent work on UML semantics has included UML Composite Structures RFP [54] where these semantics are being expressed in terms of fUML.

3.3 Action Language for fUML (ALF)

As mentioned previously, the lack of a concrete syntax for fUML is a problem when using it to define semantics. The reuse of the UML graphical concrete syntax was considered/has been attempted [96]. However, using any graphical syntax for detailed behaviour modelling can result in verbose models, additionally some concepts, such as LoopNode, do not have a standardised graphic concrete syntax. In practice, behaviours are better captured using textual notations.

Attempts at creating textual notations for describing behaviour include tool-specific action languages (such as the xUML [99] and IBM rational) and OMG's UML Action Semantics standard. In addition to the failures of UML Action Semantics described in Section 2.3.1.2, none of these have the same scope as fUML. Thus, OMG developed the Action Language for fUML (ALF) [50] which acts as a true concrete syntax for fUML.

ALF is a key part of the semantics definition approach proposed in this thesis. Thus, this section provides an introduction of the language. Firstly, the features of ALF are described. Subsequently, the syntax and semantics of ALF are introduced in order that ALF programs presented in later chapters can be understood.

3.3.1 Features of the ALF notation

The ALF standard highlighted the features of ALF:

- The users of the community are already familiar with UML and an object-oriented language like Java or C++, which primarily have C-legacy syntax. UML-specific usage, such as double colon syntax for name qualification or colon syntax for typing, should also be supported. These syntax requirements result in the syntax of ALF being a combination of C-legacy and UML syntax.

- ALF supports reference to any models that are defined using the graphical syntax of UML, even when there are special characters in the model's name. Furthermore, ALF does not need the models that are defined by UML graphical syntax to change.
- ALF has special, highly expressive syntax, which has the same expressiveness as OCL. Because ALF semantics are built on fUML, it inherits the concurrent nature of UML, which allows the simple specification of highly concurrent computations.
- It also provides notations for structured programming, such as classes, associations and signals, which means that ALF has the generality of a standard programming language.

One significant difference between fUML/ALF and UML is that fUML models can be unambiguously executed, which enables the simulating/testing of the models before they are processed to other artefacts. The ALF standard specifies three ways of executing ALF programmes:

- **Compilative Execution** maps ALF concepts to fUML and executes the fUML models. This is the approach applied by the ALF reference implementation and our former work [87]. This approach requires a good fUML executor; in our experience, there is still no available tool that could execute complex fUML models generated from ALF text.
- **Interpretive Execution** directly interprets and executes ALF text. The approaches include ALF open source implementation and IBM rational designer. These approaches do not support the full specification of ALF language.
- **Translational Execution** translates ALF text to a non-UML platform and executes the translation in the other platform.

3.3.2 A tutorial of ALF

The ALF standard separates the language description into three components, namely expressions, statements and units. Like other programming language, an ALF expression evaluates a specific value, which can be empty or can be a collection. Statements are like the statements provided in any other imperative programming languages that execute a behaviour with no specific values evaluated. The units define the structural aspects of fUML.

As indicated previously, ALF has Java-like syntax, this means that some concepts of ALF are identical to the equivalent Java ones both in terms of their syntax and semantics. Table 1 lists these identical concepts. This tutorial assumes that readers are already familiar with these.

ALF also provided notations that are different from Java, which directly represents fUML concepts. The following paragraphs summarise these concepts.

Name	Example
Expression	
Literal expression	<code>"String", 123, true, false</code>
This expression	<code>this.invoke()</code>
Super expression	<code>super.init()</code>
Property access expression	<code>student.name</code>
Invocation expression ⁷	<code>student.enrollToCourse(course)</code>
Arithmetic/logical expression	<code>1+2, condition && condition2, etc.</code>
Statements	
local name declaration	<code>String s = "hello"</code> <code>let s:String = "hello"</code>
if statement	<code>if (condition) {</code> <code> }</code> <code>else {</code> <code> }</code> <code>}</code>
switch	<code>switch (month) {</code> <code> case 1:...</code> <code> default:</code> <code> }</code> <code>}</code>
while/do while statement	<code>while(condition){</code> <code> }</code> <code>}</code>
for statement	<code>for (int i=0; i<size; i++){}</code> <code>for (element in collection){}</code>
break statement	<code>break;</code>
return statement	<code>return value;</code>
Units	
class	<code>public class Student{}</code> <code>public abstract class</code> <code>AbstractClass{}</code>
Enumeration	<code>enum Color{RED, BLUE, GREEN}</code>

Table 1: ALF syntax that is similar to that of Java.

⁷ The Invocation expression of ALF has another usage that Java does not have, which is sending a particular signal to an active class.

Tuple

A tuple is a list of expressions that could be used for method invocation. ALF supports Java-like tuples, such as “expression1, expression2”. In addition, it also supports *named tuples*, for example:

```
name=>"John", age=>34, married=>true
```

Associations and link expression

Association is a relationship between classes; one end of the association could affect the other end of the association. ALF could define associations like this:

```
public assoc Student_School{
    public students: Student[*];
    public school: School;
}
```

Link operation expressions are used for managing the instances of associations, which corresponds to the UML Link concept.

If John is an instance of Student and Manchester is an instance of the School, their link can be established by:

```
Student_School.createLink(john, manchester);
```

Meanwhile, the assigned new value of one association end will also affect the other end. For example, if cambridge is another instance of School, then

```
john.school = cambridge;
```

will result in cambridge.student including john.

Active classes and signals

An active class is a class in which the classifier behaviour of its objects runs in an independent thread. For example,

```
public active class Execution{}
do{
    accept(SignalStart);
}
```

The active class defines its classifier behaviour in the ‘do’ block. When creating a new instance of the active class, its classifier behaviour starts to execute automatically.

Signals are a kind of specialisation of the classifier. Active class instances could receive signals asynchronously. There are two ways to define a signal in ALF. The first is to declare a standalone signal classifier:

```
public signal SignalStart{}
```

The other is to define the signal as a signal reception in the members of an active class, for example:

```
public active class Execution{
    public receive signal SignalStart{}
...

```

An ‘accept’ statement defines receipts of signals. When executing the accept statement, the active object will suspend and wait for special kinds of signals. A simple accept statement, which will only wait for one type of signal, is shown above. It is possible to use compound accept statements to accept several kinds of signals, and one type of signal triggers the corresponding blocks to execute. For instance,

```
accept(sig1:SignalStart){
...
} or accept(sig2:SignalTerminate){
...
}
```

Accept statements can only appear in the classifier behaviour of active objects. The reasons are that only active objects can receive signals, and the classifier behaviours of active objects are the behaviours that are running. On the other hand, any method can send signals to active objects using a signal sending call that happens to have the same syntax as an invocation expression. For example, if ‘e’ is an instance of Execution, if a `SignalStart` is needed to be sent to e, the invocation statement will be

```
e.SignalStart();
```

Annotated statement

Certain statements can have annotations that are given special information for their execution. For example, “parallel” annotation can be added to a block or a statement. The statements inside the block will be executed in parallel, which means the sequence of the inside statements does not reflect the execution sequence.

Inline statement

Inline statements can embed the code other programming languages in the ALF programme.

3.4 Summary

This chapter provided the technical foundation of the thesis. Starting from the concepts of model-driven engineering, it presented meta-modelling and the model-driven workflow. Thereafter, an introduction to fUML and ALF were presented, with a tutorial on ALF. In

the next chapter, a framework for defining language specifications are build based on the technologies introduced in this chapter.

Chapter 4.

A Framework for Quality Language Specification (FQLS)

Previous chapters have highlighted why DSL specifications are often of low quality or, alternatively, why high quality specifications are difficult to produce. This chapter presents a framework for building language specification – the FQLS - that simplifies the production of high quality DSL specifications. The framework consists of three parts:

- A language definition method using ALF/fUML,
- A software architecture formed of three layers, and,
- A software development process.

Each of these is covered individually in the following subsections.

4.1 DSL syntax and semantics definition

The DSL syntax and semantics definition approaches reviewed in Section 2.3 all lacked at least one features associated with high quality language specification. The characteristics with the three approaches considered (translational, operational rewriting and operational weaving) are:

- Translational approaches are useful when a well-defined semantic domain exists. However, because their abstract syntax and semantics are defined in two different languages, they are harder to understand than approaches based on a single language and the possibility for inconsistencies exists. There is also currently no single agreed semantics domain on which to build.
- Generally, operational rewriting approaches are easier to understand than translational ones because they require less additional knowledge. However, since they still require two languages, the problems caused by multiple using languages still exist.

- Operational weaving approaches define both abstract syntax and semantics in a single executable meta-model. Thus, the language engineers only need to know one language, and the possibility for inconsistency between syntax and semantics does not exist.

The above characteristics mean that an operational weaving approach has been selected as the approach used in FQLS for defining both the abstract syntax and semantics of a DSL. However, the choice of the action language needs to be considered. Existing action languages such as Kermeta [18], XOCL [25], or Xcore have demonstrated that these languages can be used to execute models. However, these languages usually have two problems. Firstly, they are implemented as a platform-dependent technology and are not supported by a standardisation organisation. Secondly, these action languages have limited expressiveness, for example, neither of them has native supports for concurrency nor communication. This means that a semantics specification that specified via them is prone to be at a lower abstraction of the language, which results a type of implementation rather than specification.

When thinking about what it required from the action language to be used, core among the features is an ability to define semantics for a language defined by a meta-model. This brings into consideration fUML which is intended to allow semantics to be attached to UML meta-model elements. fUML also has the attraction that it is a standardised language supported by OMG that has produced interest from both industrial and academic sectors, and it understandable to language engineers.

Of course, since fUML is intended to attach semantics to UML meta-model elements, there is the question of whether or not it can be used to give semantics to the meta-model elements of a DSL. The fact that fUML is a highly expressive with natural concurrency support and communication support, gives promise that the answer to this question may be in the affirmative. Chapter 5 examines this question further and shows that fUML can indeed be used to define the semantics of a DSL. Within the FQLS, ALF is used as the concrete syntax for fUML.

4.2 Architecture of FQLS

While ALF forms an ideal candidate of describing DSL syntax and semantics, proper software architecture is needed in order to utilise ALF as a language definition method. Therefore, this section will introduce the architecture of the FQLS.

Figure 11 is an overview of the software architecture. The FQLS is divided into three layers, the definition layer, the analysis layer and the execution layer. In each layer, different tools are provided to support the development of a language specification, which are shown in the left-hand columns. The components of the FQLS are loosely coupled, which means the concrete tools can be substituted if better tools exist. Within the tools presented in the left column, the ALF editor, transformers, built-in checkers, and executor are developed as part of the thesis. The external checkers are checkers developed by other sources, but they can be integrate to the framework.

The right-hand column of the figure specifies the models. The arrows represent the transformation flows of different representations of the language specification. The rest of this section introduces the design of FQLS by layers.

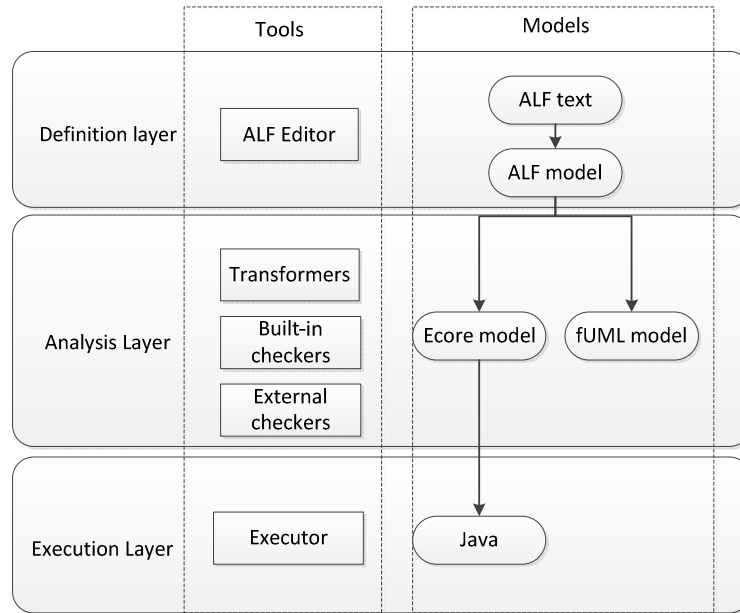


Figure 11: Architecture of FQLS.

4.2.1 Definition layer

The definition layer is the layer that the language engineers will work with directly. It is also the layer that provides input to the analysis layer. Language engineers develop a

language specification in this layer and the language specification is captured using the concrete syntax of the language specification approach, which is ALF. The editor in this layer provides three types of features, which are listed as follows.

ALF text to model transformation. A grammar that enables the parsing of the ALF text to models needs to be implemented. The ALF standard defines the grammar of ALF in an abstract way by using BNF but, in Appendix I, an implementation of the parser by JavaCC is provided. As reusing a language development project based on EMF and Xtext is desired, a new grammar file that is compatible with Xtext is implemented, namely the `Alf.xtext` file. Most of its content is derived from the JavaCC implementation of the ALF standard, because Xtext supports the generation of an Ecore model from the grammar. Therefore, a grammar that is similar to the meta-model of ALF is created, in order to generate a meta-model that is similar to the ALF meta-model, thus making the model transformation easier in the analysis phase. In addition, certain control extensions that could be placed in a comment are added, such as the “`//@OCL`” annotation, which is introduced in detail in Chapter 5. These extensions also need to be parsed to the model.

Syntax highlighting, error reporting and code completion. A language specification, regardless of the method used, needs to be captured by a notation. The language engineers need to have the ability to compose language specifications according to the notation. Language specification development usually suffers from a lack of support in composition. It is possible that the specification is created without a proper editor. An editor is the first interface with which the language engineers need to work and is the first tool that assists them.

After building a grammar for the ALF, the Xtext framework generates a skeleton of an Eclipse code editor, providing syntax highlighting and reporting parsing errors. Other tasks must be implemented manually, including:

- Code completing. When the language engineer develops code using the editor, the editor should provide reasonable code content assistance.
- Variable scoping. Each name acknowledged in a DSL that is built by Xtext is a global name. The given variables are proper scopes; thus, a local variable that cannot be accessed by other methods is necessary.
- Exporting ALF text as XML. This means that XML can be loaded with other tools that support loading models.

Bridging the analysis layer code. In the definition layer, the ALF programme is parsed to an internal model representation, and then become the model for the input to the analysis layer. The analysis layer analyses the ALF models and reports back errors/warnings to the definition layer, which delivers reports to the editor.

4.2.2 Analysis layer

An ALF-based language specification still needs a mechanism to maintain its correctness, since the correctness of a language specification is largely reliant on human factors. An incorrect language specification will confuse its audience, and will particularly impair all the other products that depend on the language specification. Because of its importance, many methods [1] that aim to find errors in models or programmes have been proposed. Depending on whether or not to execute the programmes, the methods of error checking can be categorised as *static methods* and *dynamic methods*. Static methods directly analyse the programme without executing the programme, while dynamic methods involve executing the programme and observing the execution results. The most widely accepted method is testing. To be precise, there are two ways of performing static methods, as follows:

- Inspection, review and walk-through: These are all performed by using a systematic method for reviewing the system under study. McConnell [98] reported the effectiveness of inspection and review, and determined that the result is largely dependent on human factors, such as management.
- Automatic checking: This is usually supported by an automatic tool to check for particular kinds of errors. For example, compile time error checking, including syntax and type error checking, is equipped with nearly every programming language.

Automatic checking can be roughly divided into two kinds: Formal checking, which involves transforming the programme into a formal, usually first-order logic model, using mathematical knowledge to analyse the desired properties. While the contributions of formal methods are crucial, they are usually difficult to verify in practice [137] because of the lack of skill in the industry [65].

The other kind of automatic checking is the pragmatic method, which often involves implementing a static checker that can analyse programmes and match a suspicious code to *bug patterns*. According to Hovemeyer and Pugh [70], many errors that exist in

programmes are easily detectable errors; thus, even simple detectors can find bugs in a real application. Other tools, such as [72, 27], also applied a similar approach, and they also support the checking of styles of the programme. The developers can easily extend the checkers and build new checkers into the framework. These pragmatic checking tools are widely adopted in software development.

Similarly, the errors that occur in language specifications are usually small errors, like typos [42, 160], which can be identified by applying similar methods of pragmatic checking. The FQLS implements pragmatic static checking to enhance the correctness of the language specifications.

There is no single method that could identify all the errors in an application. Pragmatic checkers are simple to implement, but are usually neither complete nor sound. As false positives and false negatives can both occur, the developers still need to be cautious when treating the error reports. Formal methods are reliable, as they usually eliminate at least one type of error. The best results for software quality assurance are achieved by combining existing methods. Thus, the analysis layer must support the performance of pragmatic and formal methods.

As automatic checking is a significant way of assuring the correctness of a language specification, FQLS should provide an architecture that enables the use of many kinds of checkers, rather than leaving the validation of the correctness to a later stage. This is the principle to design the analysis layer of FQLS.

The analysis layer takes the parsed ALF model as input, and then uses static checkers to check the correctness of the language specification, reporting any errors to the definition layer. As shown in Figure 11, this layer contains three software components.

The **built-in checkers** enable syntax correctness checking, such as basic syntax checking, bad practice checking and type checks. These built-in checkers are performed on ALF models, which differ from *external checkers* that perform checks on other types of meta-models. Since the checking and error reporting is done in the same model, a built-in checker can be implemented as a Java validator that queries and examines the ALF models. These checkers are integrated using a plugin architecture that allows further static checkers to be included in the framework.

The **external checkers**, as indicated above, mean that the checking has to be performed in a different analysis domain; thus, some analysis models need to be generated by the *transformers*. Since there are many kinds of possible external checks of the ALF programme, and the language engineers may have their own checkers, it is not feasible to

include all types of external checkers. As an example, FQLS demonstrates the integration of an external checker by integrating the Ecore OCL checker, which inputs an Ecore model with OCL constraints, and checks the syntactic correctness of the OCL expression.

The transformers. Instead of integrating many kinds of external checkers, FQLS supports exporting the ALF programmes either as Emfatic models or as fUML models, because there are already many approaches for analysing these two types of models and they can be reused. The ALF to fUML transformation is developed as an ATL project, and the ALF to Emfatic transformation is developed as an Acceleo M2T project.

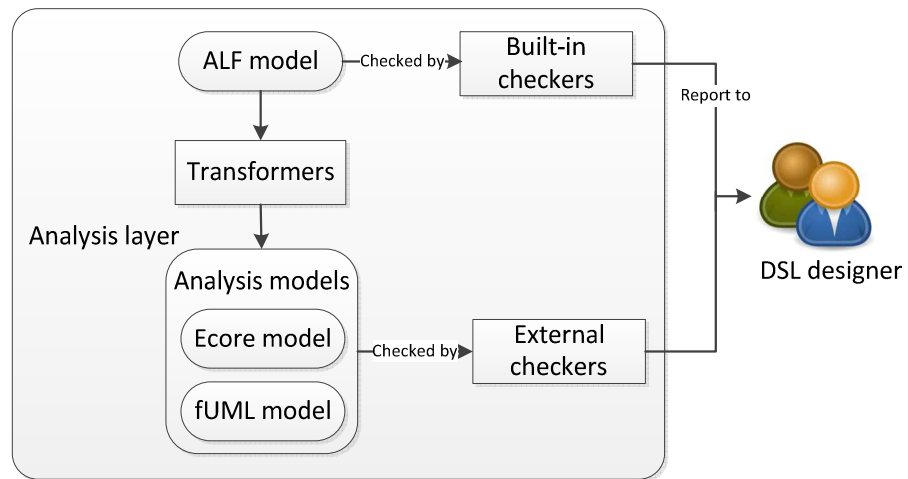


Figure 12: Workflow of analysis layer.

Figure 12 summarises the workflow of an analysis layer. The ALF models are checked by built-in checkers. Meanwhile, the analysis layer provides an interface that performs external checking. The result from the checkers is reported to the definition layer and is eventually reported to the language engineers. The Emfatic models generated from the language specification also support the reuse of EMF and are the basis of the execution layer.

4.2.3 Execution layer

The dynamic method is another category of software quality assurance. It executes the programmes in the executor. The results produced by the executor are observed and the behaviours of the executor are monitored. It is difficult to imagine that a software system would be released without testing, and test-driven development even considers tests as the first activity of development. Because of the importance of testing, a high quality language specification also needs to be tested.

Non-executable language specifications are not testable, because there is no evaluation of the semantic correctness of the specification other than review; in other words, the

language specification is semantically correct because the language engineers have declared that it is. By contrast, an executable specification that ensures testability, with the support of tools, enables language engineers to observe the behaviour of the language, thus identifying semantic errors. As a result, the execution layer also aims to assure the **correctness** of the language specification.

The term *executor* is referred as a system that could load the language input models and output the results, without attention being paid to the form of the mechanism. In order to execute a language specification, the first function that is needed is an executor of the language. ALF follows the usual method of executable modelling by providing a flexible executing definition whereby an ALF executor belongs to interpretive execution, compilative execution and translational execution, as introduced in Section 3.3. The three ways of executing ALF require different kinds of tools.

- Interpretive execution needs an ALF interpreter. The ALF open-source implementation is a current example⁸.
- Compilative execution needs a transformation or a model compiler that generates fUML models from ALF text. The transformation is the same transformation as in the analysis layer. The models can then be loaded by a fUML executor, such as the fUML reference implementation.
- Translative execution also requires a transformation, which can be a model-to-model transformation or code generation.

Although it is possible to select any of the execution approaches of ALF, FQLS prefers translative execution as it is the only feasible method of executing ALF code. The reasons for this are listed below:

Firstly, the current ALF tools are not sufficiently mature for interpretive or compilative execution of ALF language. The previously mentioned tools, such as [143], do not support the complete language. For example, the ALF reference implementation does not support the inheritance of classifier behaviour. Furthermore, the tools have a limited error reporting mechanism and a limited library.

Secondly, the unification of language specification and a reference implementation could save the effort of developing a reference implementation. However, it is still

⁸ Although the inner mechanism of ALF open-source implementation is done by mapping ALF abstract syntax to fUML and executing fUML models, from the users' point of view it is treated as a black box, because the process of transformation is not visible to the users.

necessary for them to be separated, because a language specification may not provide enough detail to work as a complete implementation. For example, a language specification may leave something intentionally undefined, like the semantic variation points. If such a separation between language specification and reference implementation is needed, then their consistency must be maintained, which could be done by deriving the reference implementation from the language specification.

Lastly, the development of a language specification needs more tools than merely an executor, such as the tools for creating/loading model instances and for validating instance models.

Since the FQLS has adopted the translative execution of ALF, the execution layer must deliver a reference implementation, which is generated from the language specification with minimal effort by reusing EMF. This reference implementation can then be used for testing purposes (testing the semantic correctness of the language specification), and eventually save the cost of building a reference implementation from scratch. In the Execution layer of FQLS, the tasks of the execution layer are listed below:

Generating executors for the target DSL. The execution is responsible for the code generation. The code generator loads the checked ALF models and finally generates an interpreter of the target language in Java.

Creating test cases with an instance model editor. This is another facility that is required for executing the language specification to support the language engineers when creating instance models for testing purposes.

Executing the language specification. The language engineers will eventually need to test the language specification in the generated reference implementation. The executor needs to load the specified model instances and output the result after the execution.

4.2.4 Summary

To sum up the features of the general idea of FQLS: it uses ALF as the definition language, performs static checks on language specification and generates language reference implementation from the language specification. All these methods are aiming to maintain the quality of the language specification.

4.3 A development process for DSL

As developing DSLs becomes a more general idea, a detailed software process could save effort on the part of the language engineer, since a software process, or a development methodology, is one important method of quality assurance. A software development process involves several different concepts. It could involve the identification of concepts, such as roles, artefacts, tasks or activities. It could also include guidelines or suggestions that a shareholder in the development process should follow.

In the area of DSL development, it is worth mentioning that Mernik et al. [100] summarises the activities of, and some guidelines for, DSL development. It does not discuss the roles and products in a DSL development process. Moreover, the process definition is difficult to reuse or adapt, because there is no automatic way of checking the consistency of the process document, nor is it possible to transform the document to a new process if certain sub-modules are changed. [156, 43] also contains many guidelines that a developer could adapt. In addition, Kleppe [80] also proposed a set of tasks or activities that need to be performed in a DSL development process. However, these works only show a general outline of how to create language specifications, rather than a detailed process that language engineers can follow.

The ad hoc definition of process prevents a more accurate presentation thereof, since all these processes are defined as text. Thus, it is the author's choice to include or not to include something. The text-based DSL development process tends to focus only on one specific topic of the process; however, it is preferable for the software development process can be formalised as a model that conforms to a standard representation, allowing the development process to be captured in a systematic and reusable manner.

Software & Systems Process Engineering Meta-Model (SPEM) [48] is a meta-model that are designed to fulfil the role of a standard way to capture a software development process. It is selected as the method for presenting the software development process the FQLS. Hence, an introduction of SPEM in terms of its features, workflow, and tools is presented firstly. The development process of FQLS is presented thereafter.

4.3.1 Software & Systems Process Engineering Meta-Model

SPEM is an OMG standard that attempts to solve the problem of the lack of standard representation and management of software methodology and processes. The meta-model and its supporting tools give [48]

- *A standard representation and managed libraries of reusable method content.* By using SPEM, the key tasks and practices can be documented, and these provide knowledge regarding the knowledge of performing these tasks for developers. For example, there are several tasks in the process of developing DSL, such as designing the language specification, testing the language specification and implementing reference implementations. The developers need to know how to perform these tasks, as well as the relevant knowledge for supporting the process.
- *Developed and managed processes for performing projects.* Although the library of method content is developed, the developers still need to select a development process. The creation of the process is based on the reuse of the method contents.
- *Configuration of a cohesive process framework, customised for specific projects.* In reality, as different projects have different scopes and characteristics, it is impossible to determine one process that can be executed for any project. SPEM supports modelling variability, so developers can define their own extensions for the reused method contents.
- *Enactment of a process for development projects.* A process known as guidance must be deployed and must be able to be accessed by the process enactment systems (the workflow engines that the developers and managers used every day). SPEM defines how a process will be enacted via these systems.

Figure 13 illustrates the workflow using SPEM model. It starts from creating method contents, which includes the atomic concepts that form a development process, such as products, activities, and guidelines. Then these concepts are used to create processes for a general domain, for example, a process for general DSL development. When developing a particular project, the developers can customise a process that fits a particular project. Finally, these models can be accessed by other project management tools and should be enacted to the real development activities. Therefore, the development process is not just a document, but also a working system that developers need to interact with in normal development activity.

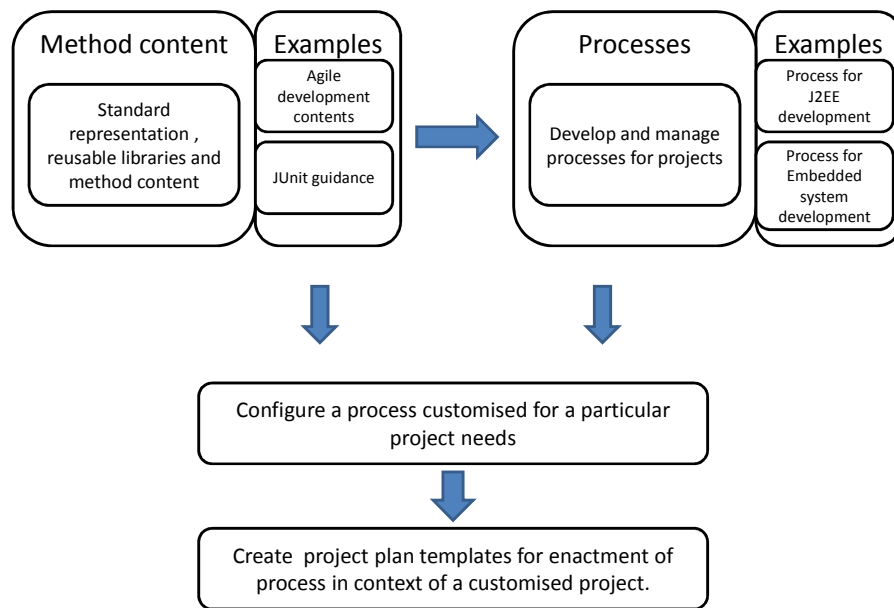


Figure 13: Overview of SPEM.

Many tools have implemented SPEM and support the definition of the SPEM model in an easy to use and easy to deploy way. The major parts of the SPEM standard are implemented by an open source project, namely the Eclipse Process Framework (EPF). Due to its completeness and wide accessibility, the EPF was selected as the tool for defining the DSL development process.

4.3.2 Method content

Method content describes the basic elements of a SPEM model. It defines the basic vocabulary in the development process, which includes concepts such as Roles, Tasks, Work Product, and guideline definitions.

Section 2.2 introduced the three roles involved in the DSL lifecycle: The language engineer, the language tool engineer and the language user. They are involved in two processes, namely the development of the language and the use thereof. Here, this thesis concentrate on the development process of the language.

The development of the language specification is the core activity of this process; because other activities performed by language tool engineers and language users are all depend on the existence and quality of the language specification.

The process of language specification development guides the language engineers to create a language specification via ALF by increments. An increment of the language specification is initialised by creating a feature of the language specification, or by modifying a feature of an existing language specification. The increment finishes when the language engineers have completed the necessary testing and validating of the new feature; thus, they can assume the feature has been implemented successfully. An increment of the language specification may involve one or more iteration. Each of the iterations delivers a successful working language specification at the end.

An iteration is separated into four phases, as shown in Figure 14. As these phases are collections of activities, each takes a set of products as input and outputs certain products, as can be seen in Figure 15.

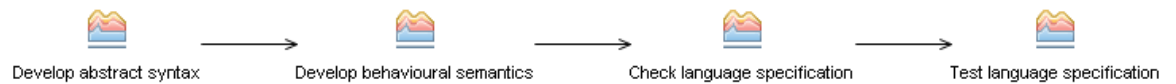


Figure 14: Phases of DSL development.

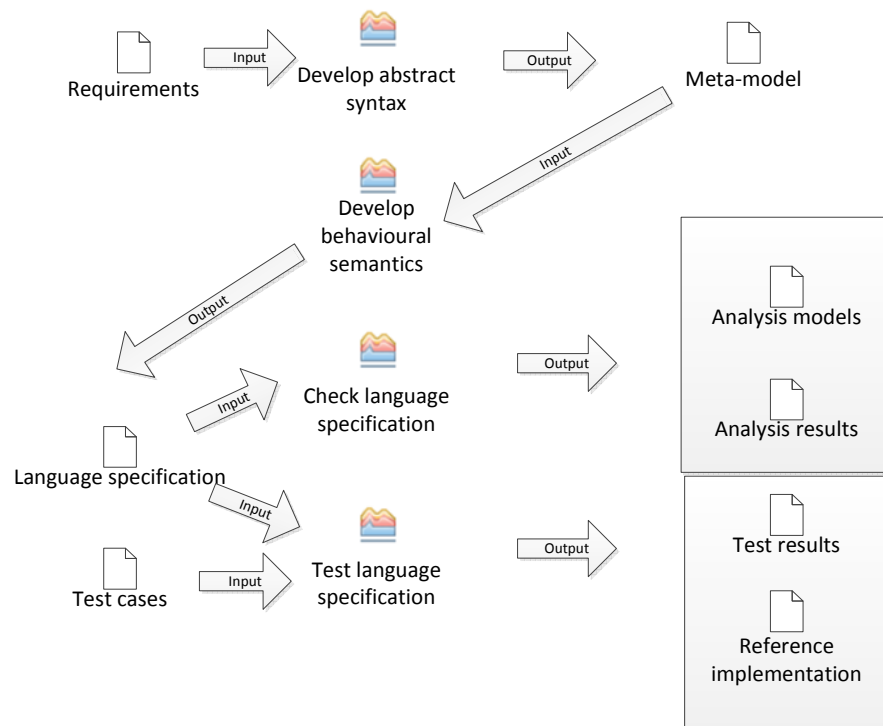


Figure 15: Products and process phases.

Starting with the requirements that were generated by the language users, the first phase, **develops abstract syntax**, which creates a **meta-model** that forms part of the

language specification. The meta-model is then used as the input to the **develop behavioural semantics** phase, which delivers the complete language specification.

The **language specification** is the most important product, and is the final product of the process. Because ALF is selected as the language for defining the language specification, the definition of a language specification in FQLS can be narrowed as follows: *A language specification is a set of models that define the concepts of a domain-specific language. It consists of abstract syntax and behavioural semantics, which are defined as executable meta-models captured by ALF.*

An unverified language specification contains syntactic and semantic errors. Thus, there are two phases – **checking language specification** and **testing language specification**. These deliver analysis models, analysis results, test results and reference implementation. These are not products that will be released to the public, but the internal products that assist in the development of the language specification.

The analysis models are the models for analysis purposes. All internal or external representations of the language specification, except the original models that are intended for analysis purposes only, can be categorised as analytical models.

Both the analysis and the testing phases produce certain results. The results may not be accessible to language users, but they help to identify errors and bad practices.

The **reference implementation** serves two purposes, namely as a tool for testing the design of the DSL, and as a tool for language tool developers to test the DSL or, if the reference implementation is mature enough, it can be the real implementation of the language.

Given the understanding of the products, the details of each phase are introduced below.

Develop abstract syntax

The first component of a language that should be developed is debatable. Selic [137] suggested always starting to design using a semantic model, thus encouraging developers to think about designing the semantics at an early stage, which avoids inconsistent design decisions that could occur when designing them as separate things. Selic's suggestion raises the importance of semantics. Treating semantics as additional work may cause problems; however, it also indicates that many language designers start by designing syntax. In fact, starting by designing syntax is a common approach and is applied by [25, 156, 80].

Developing abstract syntax is listed as the first task of the language specification development process. It is possible that the domain already has a type of language that is but not formalised, or that migrating an existing DSL to model-base is required. In both cases, the syntax should be considered first. In addition, when defining a language specification as an executable meta-model, syntax and semantics are not separate and are linked from the beginning. The semantic definition uses the abstract syntax concepts as the vocabulary.

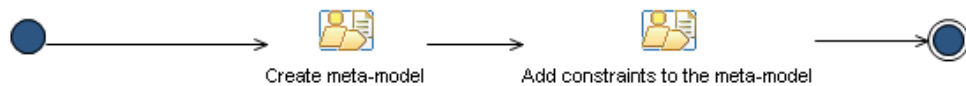


Figure 16: Developing abstract syntax.

Figure 16 illustrates these activities. The first activity when designing abstract syntax is to **create the meta-model**. The meta-model may originate from two different sources: Either it is designed from scratch, or it is imported/rebuilt according to an existing representation of the language syntax. When creating the meta-model from scratch, the language engineers need to work closely with the requirements and the domain experts, and to abstract the domain to a set of entities, attributes and relationships. This process should follow the experiences and best practices in language design and domain engineering.

When formalising a language specification to an ALF-based representation, the meta-model may already exist, but be defined as another format, such as BNF grammar or XML schema. Such languages capture the abstract syntax tree; it is possible to convert such representations into an ALF programme by representing an element as a class, and containing/reference relationships as associations. Many works, such as Xtext, EMFtext and EMF could convert a BNF grammar/XML schema to an Ecore model. Regardless of the tools that have been used, it is possible to derive a meta-model from other formats of defining the abstract syntax with reasonable effort.

Once there is a meta-model, the next activity is to **add constraints to the meta-model**. These constraints define a well-formed model. Many technologies can be used to define constraints, and OCL is the standard constraint language for UML-like meta-models. Thus, a strategy for defining constraints needs to be defined before developing strategies. The explanation of how to add constraints to ALF is detailed in Section 5.1.2, which introduces three different ways of adding constraints to an ALF specification. These are adding them

as additional files, using extended ALF, and defining constraints as semantics. When the constraints are developed, the output of this task is the abstract syntax definition, which is an ALF programme representation. This ALF programme is ready to have semantics added.

Develop behavioural semantics

The purpose of this phase is to create the behavioural semantics of an ALF-based language specification, adding operations and behaviours to the abstract syntax definition. The input of the task is the abstract syntax definition, and the output is a completed language definition.

Because of the complexity of developing semantics, the task is split into five activities, as shown in Figure 17.

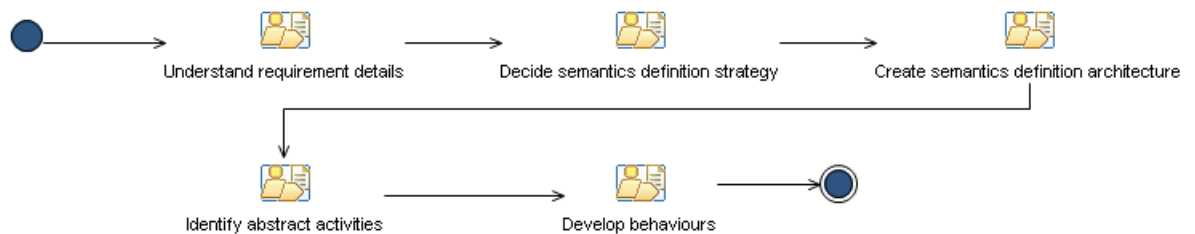


Figure 17: Develop behavioural semantics.

The first activity is to **understand the requirement details**. In various scenarios, the requirement can be gathered from the domain experts and the knowledge of the language engineers. No matter how these requirements are gathered, a deep understanding is the first step of developing the expected language specification instead of an incorrect specification. When formalising a language, the requirement is clearer because it is already defined in another format.

The second activity is to **decide the semantic definition strategy**. Semantic specification is a type of programme and it involves many design decisions. There are many ways of implementing the same semantics. However, this does not imply there are no rules for design decisions. Before implementing the semantics, several questions need to be clarified, for example, how to deal with semantics variation points. A full list of the questions and guidelines on developing behavioural semantics is listed in Appendix B, where interested readers can check in detail.

The next activity is to **create semantic definition architecture**. An executable language specification is also a kind of software; thus, it follows the general rules of software design. Specifically, the language engineers need to decide how to attach

behaviours to the meta-model and how the state-machine of the runtime model should be build if needed.

The last activity is to **develop the behaviours**. With regard to the requirements and the decisions and rules given above, the language engineers can develop the language semantics.

Checking the language specification

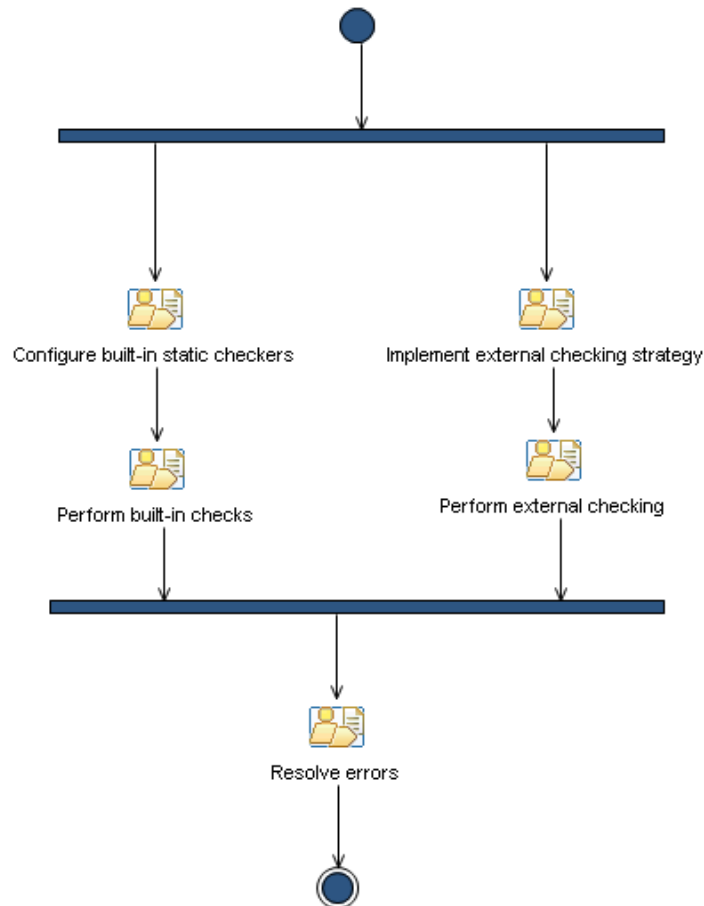


Figure 18: Checking language specification.

Figure 18 illustrates the activities in the **checking language specification** phase. The first activity is to **configure built-in static checkers**. During the task of developing behavioural semantics, the language engineers decide the architecture, the coding style and ways of dealing with various issues. Since the static checkers indicated that there were various errors and bad practices to check, depending on the language engineers' decisions, something that could be considered as an error might be acceptable in other circumstances. For example, a checker that forces developers to use Java style or UML styles needs to be turned on or off.

The next activity is to **perform built-in checks**. According the configuration, the static checkers will perform the selected checks on the language specification and report errors and warnings to the editor. the internal checks are defined as checks that do not involve translating the language specification to other representations. Because these kinds of checks do not need to consider translation and message tracking, such checks are easier to implement than ones performed by external checkers.

Concurrent with the use of built-in checkers, if the language engineers decided to use external checkers, these could be done via an activity that **implements an external checking strategy**. External checks, as discussed above, are more complex than internal checks. When considering an external check, the language engineers should have a particular checking property in mind, as well as the approach that will be taken for external checks. There are several possible scenarios, depending on what is to be checked. An interface of fUML and Ecore is provided, but did not integrate any other analysis approach. The reason is that there is no external analysis that needs to be performed for every language specification. The selection of concrete analysis and the way of annotating analysis results are left to the language engineers.

The next activity is to **perform external checking**. Once the checking tools have been decided, and the necessary transformations and reporting tools have been developed and tested, the language engineers need to perform the checks.

The final step is to **resolve the errors** reported by the checkers. However, resolving errors is not really the final step; it is only the final step of the iteration. The checking and resolution of errors should be repeated until no further issues are found.

Testing language specification

Although the language specification has been statically checked, these kinds of checks can only discover syntactic errors. Semantic errors can also be detected, but most semantic errors are just incorrect semantics that are not expected by the users. Since the correctness of the language specification is actually decided when the standard is released, a semantic error that remains in the release stage would entail great effort to fix. As a result, the language specification needs to be tested to see whether the semantics reflect the language engineers' intentions.

Language specification is an executable meta-model. To test such a specification, several sub-tasks must be executed in order to create a test suite for a language specification.

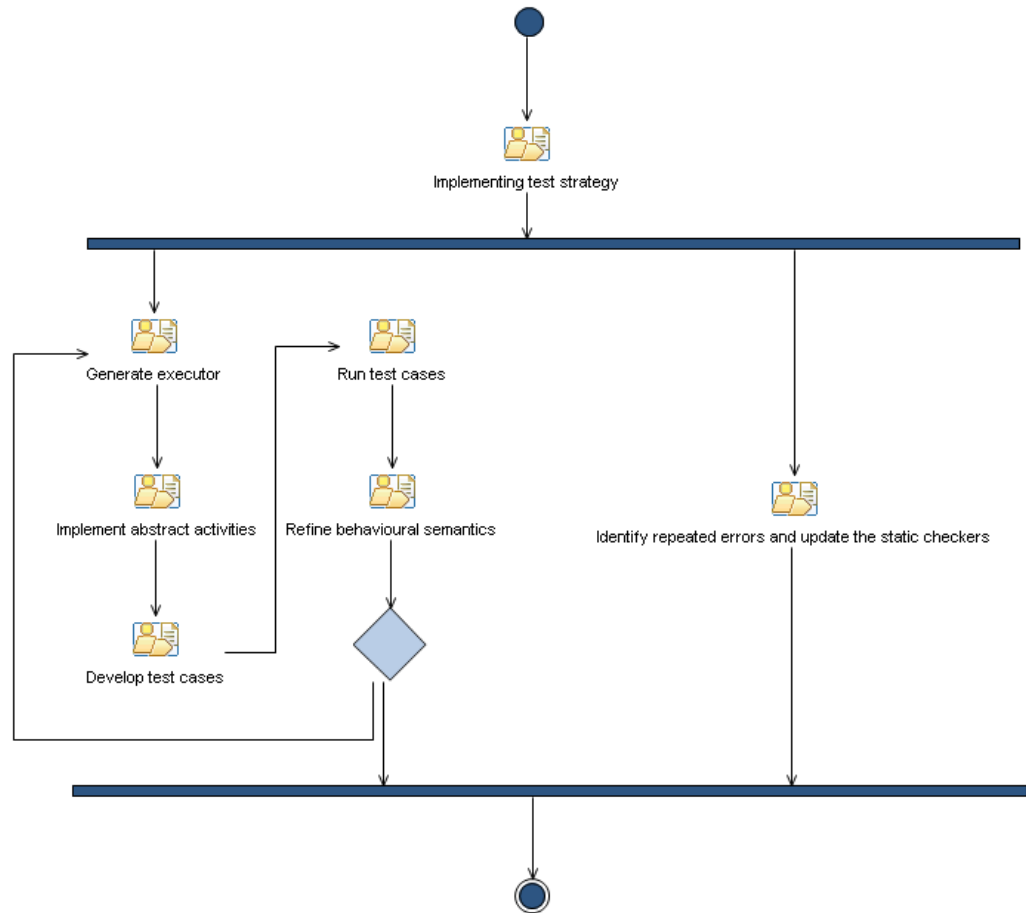


Figure 19: Testing language specification.

Figure 19 illustrates the process of testing language specification. The first activity is to **implement a test strategy**. An executor enables testing; a programme cannot be tested without an executor. However, an executor alone is not enough, as a framework for supporting it is also needed. Since a DSL under development is not like a GPL for which a testing framework such as JUnit exists, enabling automatic unit testing requires the creation of a framework. Language engineers may also consider manual testing.

When executing a test case of a GPL, which is fairly straight forward, the unit being tested is given relevant inputs and executions; the developers then analyse whether the outputs were expected. In comparison, language specification as a whole is a large black box in which inputs can be complex and outputs may not be obvious. The inputs are a DSL programme and the language engineer must define the ‘result’ that is to be analysed. Executing the DSL programme may cause the executor to output something to an output channel. This is an output of the DSL programme, but is not necessarily the result that the language engineer wants to analyse. The DSL programme may not have output in terms of its own input. As an abstract model, the DSL programme may only result in the internal

states being changed. In such a case, the language engineer needs to test the internal states of the language executor.

The next activity is to **generate an executor**. The language specification is sufficiently detailed for any GPL that has the same structures as the language specification to generate an application, and the behaviours are also transformed into GPL implementation. Chapter 7 demonstrates this via a code generator that generates Java. This executor is completed in the next activity and is used for testing the language specification.

Depending on the completeness of the language specification, the language engineers may need to **implement abstract activities**. As indicated above, unspecified behaviours are defined as abstract activities in FQLS. However, to make the reference implementation executable and to produce the expected result, these abstract activities need to be implemented. Depending on the purpose of the generated implementation, when it is a prototype to show how the standard works, the implementation does not need to be as complete as a real implementation. On the other hand, if the generated implementation is the basis of a real language executor, then many issues such as efficiency, the quality of the generated code and the use of the API or libraries must be considered. However, creating a language implementation of a language standard is out of the scope of the language specification developed.

When the language reference implementation is ready for execution, the language engineers need to **develop test cases**. Language developers can create input models using the generated model editor. When the input models are created, the expected result should also be created. When working with domain experts, they should confirm the result. It is possible that the language engineers' understanding is different from that of the domain experts, thus creating incorrect test cases and passing them via an incorrect executor. Good practice [142] suggests building tools for the domain experts so that they can build the test cases and the expected result directly.

Then the final activity is **to run the test cases** and to **refine the behavioural semantics**. When a test case fails, the language engineers should make the necessary changes in the language specification and regenerate the specification rather than modify the generated code. When the language specification changes, the implementation of abstract activities may also need to be changed.

There is another optional task, which is **to identify repeated errors and to update the static checker**. FQLS is designed to allow the language engineers to add new rules or algorithms for static checking. When testing and refining the language specification, it is

possible that types of errors can occur repeatedly. In fact, when developing the case study in Chapter 8, it is identified that many errors that cause the executor to fail are small mistakes that could be checked at the language specification level with little effort when compared to leaving this to being identified at the generated implementation stage.

4.4 Summary

FQLS is a framework that uses ALF to define both abstract syntax and behavioural semantics, which performs static checks on the language specification, and enables the generation of reference implementation for testing a language specification. To apply the idea, the software architecture of FQLS, which provides support to define DSL syntax and semantics via ALF, is illustrated. Finally, a software development process, which is captured by SPEM, is proposed. This process defines the roles, products, activities and guidelines in DSL specification development.

Chapter 5.

Defining DSLs using FQLS

In the last chapter, the development process and the architecture of FQLS is introduced. Thereafter, Chapter 5, Chapter 6 and Chapter 7 explain the layers of FQLS in detail. This chapter introduces how to use ALF as a method to define a DSL by starting from introducing the principles in Section 5.1. In order to illustrate these principles, Section 5.2 gives an example of a Petri net language, which is defined as a meta-model and the abstract state machine transition rules. In Section 5.3, the same example is then defined using ALF language, with explanations regarding the ways in which ALF units can represent a meta-model, how it can be integrated with OCL, and methods of defining behavioural semantics as operations. Finally, Section 5.4 discusses how an ALF-based definition can simplify consistency checking and why it is an interoperable, understandable and expressive language specification.

5.1 Defining DSL via ALF

As introduced in Section 3.2, fUML defines the semantics by itself, and is able to define higher level UML semantics. Hence, there is no doubt that fUML can give semantics to UML concepts. However, there is the question of whether or not it can be used to give semantics to the meta-model elements of DSLs. In order to represent the behavioural semantics in an operational way, the most important requirement is a language that can specify basic behaviours, for example, basic calculations, conditions and loops. As long as these basic behaviours are provided, the complex behaviours can be composed using them.

The semantics that a language can express depend on the expressiveness of the semantics definition language. Existing action languages support variable assignments, sequential execution, conditional statements or loops well; what they do not support well is to specify concurrency and communications. In comparison, fUML supports the basic behaviours in the same way as other action languages. It also natively support concurrency

and communications. This means that fUML can specify an even wider ranges of DSL semantics than existing action languages in a simpler way.

In order to demonstrate that a DSL specification can be defined via ALF/fUML, the rest of this section provides principles of how to use ALF to represent the abstract syntax, the static semantics and the behavioural semantics of a language specification.

5.1.1 Representing abstract syntax by ALF

Since FQLS aims to use ALF as the single technology to represent a language specification, it is needed that abstract syntax of a language must be able to be defined via ALF. While MOF is the usual way to capture meta-models, ALF provides similar concepts of classifiers, attributes and associations. These are the basic elements for representing a meta-model. Because ALF is a concrete syntax for fUML, its kernel package is compatible with the UML 2 meta-model, and MOF imports `UML::Classes::Kernel` from the UML meta-model. Thus, the kernel classes of fUML and MOF are treated as the same meta-classes. This means that, though UML class models and MOF models exists in different meta-levels, they are the same thing and have the same expressiveness. Hence, it is possible to use ALF to represent a meta-model by treating ALF as a meta-modelling language. Defining a meta-model uses ALF becomes straightforward work.

5.1.2 Representing static semantics by ALF

Static semantics are additional constraints of the meta-model. If the model is captured directly by MOF or an Ecore meta-model, there are ways of adding OCL constraints. Certain specially designed syntax for representing Ecore, such as Kermeta [18] and Emfatic, has a method for adding OCL for representing pre- and post-conditions and invariance, such as the example in Listing 1. Although ALF has the same expressive power as OCL, it also supports OCL collection expressions; however, there is no way of directly integrating OCLs.

In order to allow constraints in meta-model, this subsection proposes three ways to include OCLs in a ALF program, and discusses their benefits and drawbacks.

The first way is to add OCLs as an additional view of the models. Because ALF units are just a concrete syntax, the real meta-model they represent can add OCLs. The published language standard can then contain several parts, namely the abstract syntax defined by ALF or just a MOF model, the behavioural semantics defined by ALF, and the static semantics, defined by OCLs. A compiled fUML model is then created from these

specifications for machine processes. Although this method could represent static semantics, it makes consistency checking necessary. However, much of the research (as summarised in [93]) suggests that consistency checking for UML is not easy.

The second way is to extend ALF syntax so that it can support OCLs. By creating a large meta-model that contains the abstract syntax of ALF and OCL, all aspects of a language specification can be integrated as one extended ALF programme, and consistency checking is easy because it is restricted. However, this breaks the interoperability of the specification, because the extended ALF is not compatible with the OMG standard.

The third way is to use the existing features of ALF to represent static semantics. OCL is used for querying models or constraint models. When querying models, developers can use OCL to specify derived properties, which are the properties of which values are derived from other properties. When giving models constraints, OCL expressions define invariance, pre-conditions and post-conditions. All of these can be specified as special operations on a model, and can be defined as behaviours. Derived properties can be substituted as assignments in the constructors of the class. Pre- and post-conditions can be substituted as ‘if’ statements, and invariance can be written as a validation that must be performed before changing the model.

This method does not necessitate the loss of interoperability and consistency. However, this will make the behavioural semantics specification more complex, and it is difficult to verify the static semantics by other tools.

Inspired by the “`//@inline`” and “`//@parallel`” annotations, a compromised solution is included in the FQLS, which supports adding OCL to the language specification in the same place while still maintaining interoperability. These annotations start with an inline comment symbol; therefore, if a tool does not support these annotations, they will be ignored and the programme will still remain a valid programme. Following the same principle, an OCL annotation is added to ALF:

```
//@OCL("invariance")
```

By using this method, the language specification can be physically unified as one file. When checking its correctness, some crosschecks need to be performed on the ALF programme, as well as on the OCLs. This is detailed in Subsection 7.3.2.

5.1.3 Representing behavioural semantics via FQLS

After a meta-model has already been defined by ALF, the next step is to attach behaviours to the meta-model. Learning from existing approaches, there are three ways to attach detailed behaviours to a meta-model using an action language, namely direct attachment, attachment as separate rules and attachment via a run-time meta-model.

The first method, *direct attachment*, as shown in Figure 20, attaches behaviours directly to the abstract syntax model. General meta-model definition languages, such as MOF or Ecore, all have the ability to define operations' interfaces as members of classes. There is no default method of expressing detailed behaviours; however, if the operations can be extended, it is natural to be able to define detailed behaviours. By defining semantics as the operations body of a meta-model, a complete executable meta-model standing for the language specification is created.

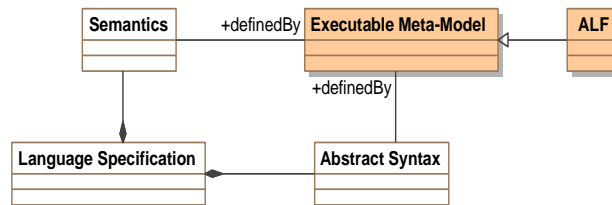


Figure 20: Attaching behaviours directly to meta-model operations.

Direct attachment is the simplest way to attach behaviours, and results in a unified, executable meta-model. However, this method needs to change the abstract syntax model by adding operations, which is not expected if the task is to add semantics to an existing meta-model. It is common that the abstract syntax has already been published and used by many other tools before language developers create the formal semantic specification. As a result, modifying the meta-model breaks the interoperability of the language specification.

The second method *attaches behaviours as an aspect of the system*, using the idea of aspect weaving. Because adding operations to the abstract syntax model modified the original model, it is possible to define the semantics separately in order to avoid this, and to use an aspect weaver to combine. Although this appears similar to translational semantics, there are fundamental differences: the syntax and semantics of translational approaches are defined using different meta-models, but the weaving approach defines syntax and semantics using the same meta-model. It is much easier to develop an aspect weaver for a bidirectional transformation from syntax to semantics.

This approach does not need to add operations to the abstract syntax. However, it also has some limitations. One problem is that direct attachment does not permit the full use of

the signal and concurrent programming of ALF. ALF only allows activities and the classifier behaviour of active classes to use ‘accept’ statements for receiving signals. If the language developers wish to use a signal-driven model as the semantic model, they have to define their own signal management infrastructure.

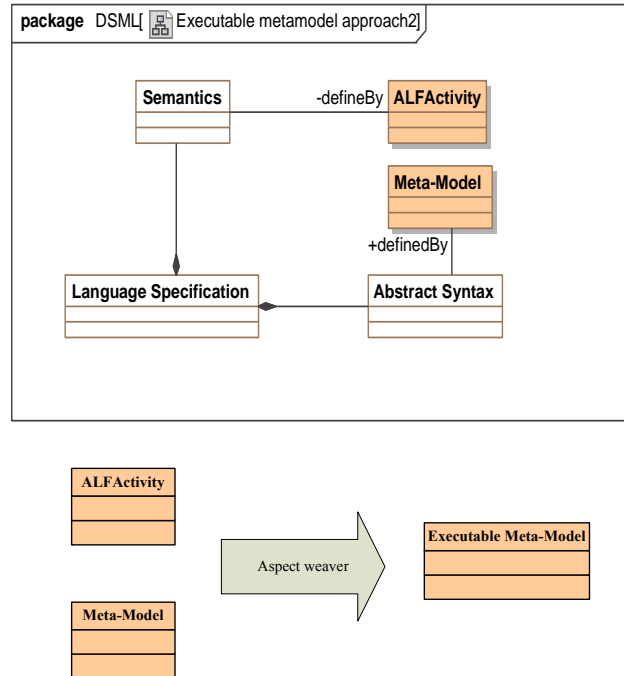


Figure 21: Attaching behaviours as separate operation rules.

The other problem is that the developers cannot differentiate between runtime concepts and structural concepts. The abstract syntax defines the structural view of the language; however, depending on the DSL, it is possible that the concepts that are processed are instance level concepts when executing the language. For example, as illustrated in Figure 22, a language contains `Variables` and `Statements`. `AssignStatement` has a left hand and a right hand reference. The execution semantics of `AssignStatement` copy the value of the right hand variable to the left hand variable. The copy is on the instance level value, which means a class that represents instance level concepts has to be added, as in Figure 23.

At the abstract syntax model level, the variable does not have an attribute called ‘value’ because, in the DSL programme, the variables are only declarations and the runtime value does not need to be specified. The `Object` class is the instance of the `Variable` and has a value property that the `Variable` class does not have. The `Locus` manipulates the instance level concepts, rather than changing the original programme model. This design permits running many instances of the same programme. In summary, it is often necessary to separate runtime concepts from structural concepts.

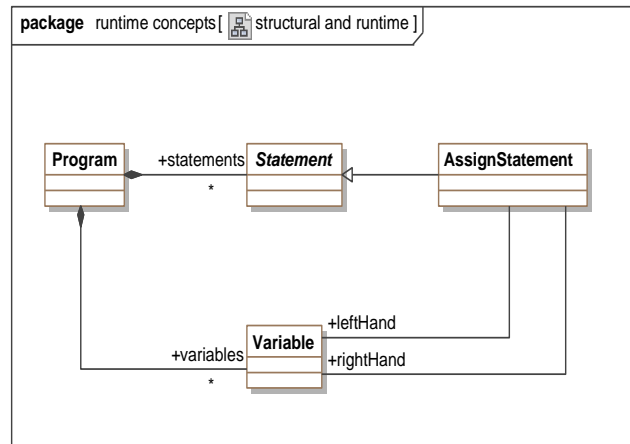


Figure 22: Variable and Statements meta-model.

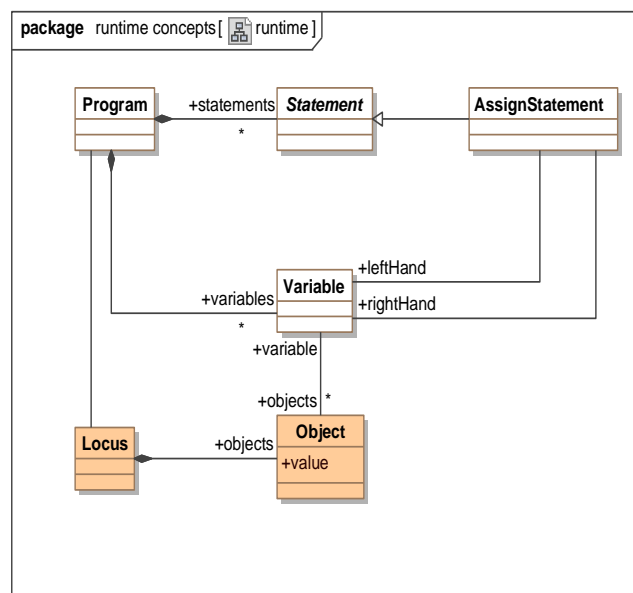


Figure 23: Variable and Statements run-time meta-model.

Because the separation of structural and runtime concepts is necessary, a widely used approach is to create the semantics of a modelling language by starting from a runtime meta-model by adding new instance level classes to another package, and then importing the abstract syntax (the method for defining the semantics of fUML [55] is an example). As illustrated in Figure 24, the runtime meta-model combines structural and runtime level concepts, making it possible to attach behaviours to the runtime meta-model. Since all the models are defined using ALF, the language specification is still a unified model.

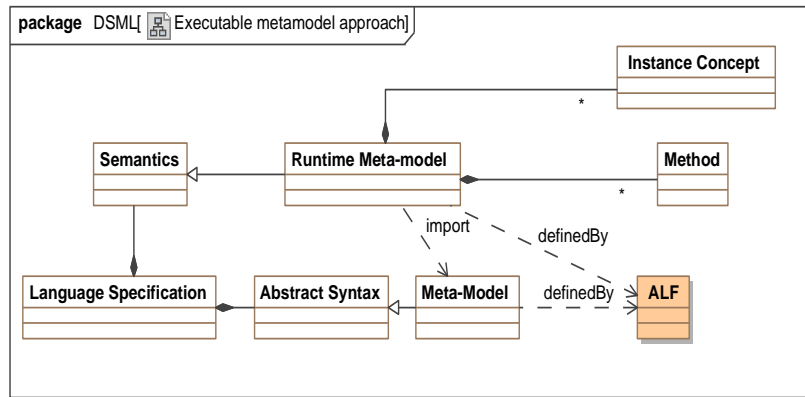


Figure 24: Attaching behaviours to a run-time meta-model.

A runtime model does not change the abstract syntax model; thus, it does not break the interoperability. By extending the abstract syntax model, language engineers can use the full expressiveness of ALF, using active classes and signals for communication. Although creating a runtime meta-model involves extra work, the work is necessary for separating structural and runtime concepts.

In summary, if a simple DSL already represents runtime level concepts and there is no need to add any concepts, it is better to use direct attachment, or weaving behaviour and structure together. However, if runtime concepts need to be added, then a runtime meta-model is necessary. The BPEL case study in Chapter 8 illustrates an example of using a run-time meta-model.

5.2 Defining a Petri net language

After the discussion of how a language specification can be represented as an ALF program, this section provides an example of a Petri net language, and represents it by ALF in the next section.

Petri net is a mathematical modelling language for information flow, which was originally defined by formal language [115]. Petri nets have been widely used as the basic example of DSML development. Works like those of [132, 133, 157] all used Petri net as an example. As this example's syntax and semantics have been exhaustively studied and are widely understandable, this example is also used to demonstrate the definition method.

There are many ways of creating a meta-model that abstracts Petri nets. For example, the meta-models used by [132] and [96] are slightly different. The Petri Net Markup Language (PNML) [14] was created in order to standardise the representation of Petri nets,

and has become an ISO standard for defining Petri nets. The standard provides XML-based syntax, together with an official meta-model based definition implemented by Ecore.

As a standard, PNML intends to define all types of Petri nets, regardless of whether these are basic Petri nets, coloured or high-level Petri nets. Thus, the language is designed to be a somewhat abstract definition that contains the net, arcs, nodes and graphics. Additional information, such as legal labels, is defined as an extension called Petri Net Type Definition (PNTD). A PNTD is an instance of a PNML, and that the PNML meta-model is an abstract class or interface.

Figure 25 shows the meta-model of a simple PNTD, in which the class `Place` is labelled “initialMarking”. The abstract syntax for the PNTD is the standard model, transformed to remove concepts related to concrete syntax, such as positions and shapes. This shows that, conceptually, a `PetriNet` contains `Arcs`, `Transitions` and `Places`, where `Transitions` and `Places` generalise the concept of `Node`. The associations between `Arc` and `Node` enable navigation through the Petri net. The initial marking of a net can be defined using the `initialMarking` attribute of a `Place` that indicates how many tokens are initially located at a `Place`.

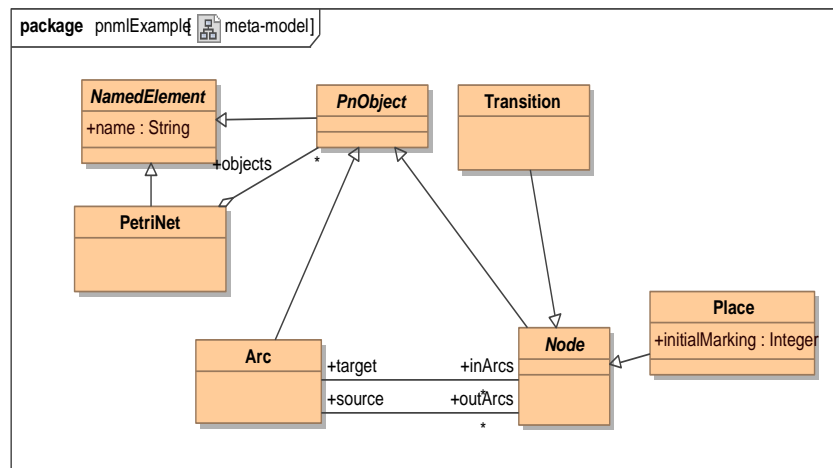


Figure 25: The meta-model of a PNTD.

In addition to the abstract syntax, the model includes static semantics expressed in OCL. The OCL constraint shown below prevents an `Arc` linking a `Place` to a `Place` or a `Transition` to a `Transition`.

```

context Arc inv:
    (source.ocIsKindOf(Place) and target.ocIsKindOf(Transition)) or
    (source.ocIsKindOf(Transition) and target.ocIsKindOf(Place))

```

Listing 1: OCL added to Petri net meta-model.

Behavioural semantics for the PNTD have been created in an ASM formalism that is similar to that used to define the semantics of BPEL [37]. This ASM is created by mapping the concepts in the abstract syntax to the concepts in ASM via a “by name” association between the two technical spaces. UML classes are mapped as a Universe in the ASM, with a generalisation relationship mapped to a sub-set relationship; for example, class `PnObject` and `Node` are mapped as shown below:

Universe: $Node \subset PnObject$

Universe: $Transition \subset Node$

Attributes and properties are mapped to a function with their parent class as a parameter. For example, `NamedElement.name` is translated to

name: NamedElement \rightarrow String

Since the mapping from UML classes to universes and functions are straight forward, they are omitted due to limited usefulness. The transition rules and important functions are listed below:

1. RUNPETRINET
2. $(p \in PetriNet) \equiv$
3. **let** *activeTransition* = *getActiveTransition(p)* **in**
4. FIRE(*activeTransition*)
5. FIRE
6. $(t \in Transition) \equiv$
7. **forall** $p \in getSourcePlaces(t)$ **do**
8. $token(p) := token(p) - 1$
9. **forall** $p \in getSourcePlaces(t)$ **do**
10. $token(p) := token(p) + 1$
11. *getActiveTransition* ($p \in PetriNet, t \in Transition$) =
12. $\exists e \in getTransitions(p) \ isActive(e)$
13. *isActive*($t \in Transition$) \mapsto
14.
$$\begin{cases} True, \forall e \in getSourcePlaces(t) \ token(e) > 0 \\ False, else \end{cases}$$

Listing 2: Semantics of Petri net defined by ASM.

This behavioural semantic specification has several deliberate inconsistencies and errors, including:

- Syntax error: A function name is spelled incorrectly (*Transtion* in line 13 should be *Transition*). If an ASM syntax checker supports the syntax above, such an error will result in a syntax error.
- Inconsistency with the meta-model: In lines 8 and 10, the function of getting the token from a `Place` is named “token” rather than “initialMarking”, which means that the same property appears under different names in the syntax and semantic definitions. An ASM syntax checker will not find this error, because the ASM specification is a valid specification within its technical space.
- Logical error: The rule FIRE should decrease the number of tokens in the sources by one and increase those in the targets by one. Therefore, *getSourcePlaces(t)* in line 9 should be *getTargetPlaces(t)*.

These errors can be found by a careful review of the specification. However, the specification of a DSL can be large, running to hundreds of pages, so not all mistakes will be identified by a review process.

5.3 Defining Petri net language via ALF

In this section, PNTD introduced in the previous section is defined using ALF as an executable meta-model. The definition starts from representing the abstract syntax of PNTD.

It is possible to establish a mapping and use ALF for representing a meta-model in a similar way to other textual representations of meta-modelling languages, such as Emfatic or KM3. The following code defines a class:

```
public abstract class NamedElement{
    public name:String;
}
```

In the example above, the Java-like syntax defines a class, an attribute (can specify the multiplicity), the type of attribute and the `isAbstract` attribute of the class.

Generalisation can be defined using the “specializes” keyword.

```
abstract class PnObject specializes NamedElement{}
```

When defines a reference, the “compose” keyword illustrates that the referenced object is composed by the parent object.

```
public class PetriNet specializes NamedElement{
    public objects:compose PnObject[*];
```

The associations between Arc and Node can be defined as:

```
public assoc Arc_Node_Source{
    public source:Node;
    public outArcs:Arc[*];
}
```

By the same method, all the classes and their relationships can be defined as shown in Figure 25. These result in a meta-model captured by ALF syntax. While defining the abstract syntax, the OCL constraints can also be added to the ALF program.

When defining behavioural semantics, the PNTD model is already describing a Petri net instance, there is no need to add more instance concepts; thus, behaviours can be directly attached to this model. In order to realise this approach, four operations are added to the original meta-model, as shown in Figure 26. The behaviours are captured in the operations.

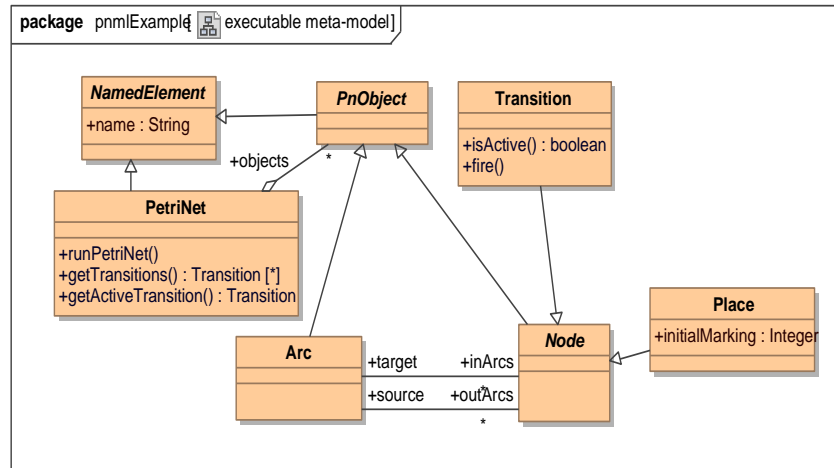


Figure 26: Executable meta-model of PNTD.

Then the body of the operations can be developed. For example, in `Transition` class, an operation `isActive` is needed to tell whether a transition’s source places all have a mark, so that the transition is ready to be fired. This can be specified in OCL as a derived function

```
self.inArcs->forAll(e|e.source.oclAsType(Place).initialMarking>0)
```

It can also be written using ALF syntax

```
public isActive():Boolean{
    return this.inArcs
```

```

->forAll e (((Place)e.source).initialMarking>0);
}

```

The code below is the complete ALF definition of the Petri net language.

```

public abstract class NamedElement{
    public name:String;
}
public class PetriNet specializes NamedElement{
    public objects:compose PnObject[*];
    public runPetriNet (){
        let activeTransition:Transition = this.getActiveTransition();
        activeTransition.fire();
    }
    public getActiveTransition():Transition {
        return this.getTransitions()->select e (e.isActive())->at(0);
    }
    public getTransitions():Transition[*]{
        return this.objects->select e (e instanceof Transition);
    }
}
abstract class PnObject specializes NamedElement{}
abstract class Node specializes PnObject{}
/* @OCL("(source.ocIsKindOf(Place) and target.ocIsKindOf(Transition))
or (source.ocIsKindOf(Transition) and target.ocIsKindOf(Place))")*/
class Arc specializes PnObject{}
public assoc Arc_Node_Source{
    public source:Node;
    public outArcs:Arc[*];
}
public assoc Arc_Node_Target{
    public target:Node;
    public inArcs:Arc[*];
}
class Place specializes Node{
    public initialMarking:Integer;
}
class Transition specializes Node{
    public isActive():Boolean{
        return this.inArcs->forAll e (((Place)e.source).initialMarking>0);
    }
    public fire(){
        for (Arc arcIn : this.inArcs){
            let place:Place = (Place)arcIn.source;
            place.initialMarking--;
        }
        for (Arc arcOut: this.outArcs){
            let place:Place = (Place)arcOut.target;
            place.initialMarking++;
        }
    }
}
}

```

Listing 3: ALF specification of the PNTD.

It is also possible to define the behavioural semantics as separated activities. In order to use this approach, the behavioural semantics are defined as ALF activities that manipulate the input model. Five activities (shown in Figure 27) can be added and one of them can be defined as the start activity.

```

3      public activity runPetriNet(inout pt: PetriNet){}
4      public activity getActiveTransition(inout pt:PetriNet){}
5      public activity getTransitions(inout pt:PetriNet):Transition[*]{}
6      public activity isActive(in transition:Transition):Boolean {}
7      public activity fire(inout transition:Transition){}

```

Figure 27: Defining semantics as activities.

5.4 Discussion

This subsection discusses the relevant quality features that the definition layer supports, and explains why using ALF would reduce the first two types of errors given in the ASM-based Petri net specification.

Consistency

There was a debate regarding the use of either a multi-viewed modelling language, such as UML⁹, or a single-viewed modelling language, such as the single diagram approach [114], for software modelling. The one diagram approach, although providing greater understandability and a higher correctness rate, is not sufficient for modelling a complex software system. It supports the modelling of smaller views, such as structures and detailed behaviour, but does not support higher-level behaviours (e.g. state machines), composition, deployment or requirements. Although it is possible to create a unified language for all aspects of a software system, the meta-model of the modelling language will be complex and the model quality will be difficult to maintain due to the complexity thereof.

Compared to a software system, a language specification contains fewer components, mainly in terms of abstract syntax and behavioural semantics. The common inconsistencies are inconsistent syntax and semantic models, because they are defined in different languages. In order to check such inconsistencies, a common approach is to create a composite meta-model that composes the syntax and semantics specification, then validates the composed model and traces the errors from the composed model to the

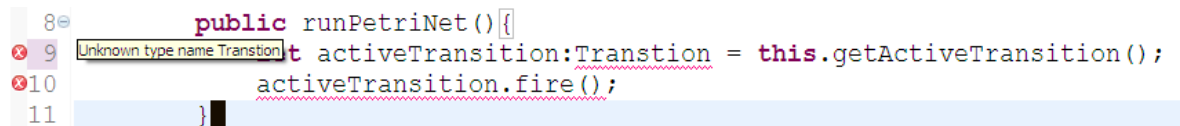
⁹ UML is multi viewed, but the UML meta-model could bring all views together. However, the UML standard does not force the checking of the syntax validation. Hence, the unified meta-model of UML does not make the validation of the consistency of UML easier.

original model. Thus, before automatic consistency checking can be provided, the language engineers need to

- Link the syntax and semantics model by transformations, which is a significant amount of challenging work.
- As the translations and traces may also contain errors, they must also be validated and maintained.

In comparison, ALF already provides a model that can capture both syntax and semantics; thus, the translations, traces and possible errors that could occur during the process are eliminated. In an ALF programme, detecting inconsistency is as difficult as is maintaining the correctness of the syntax. Therefore, if the language engineer resolves all syntactic errors of a language specification by FQLS, inconsistency is eliminated.

Let us now return to the semantics of the Petri net that contains intentional errors, presented in Section 5.2. The ASM version of the PNTD consists of an abstract syntax specification that is defined by a class diagram and a behavioural semantic specification defined by ASM. The first error is a syntax error - a name typo. Such syntactic errors of a language are easier to check by a parser or a model validator, as shown in Figure 28. If the ALF specification has a basic syntax checker, such an error can be found, as in the example shown in Figure 28.



```
8 public runPetriNet() {  
9     t activeTransition:Transtion = this.getActiveTransition();  
10    activeTransition.fire();  
11 }
```

Figure 28: Checking name typos.

Consider the second error, which is an inconsistency between syntax and semantics. The inconsistency is due to different views of language specification using different names to refer to the same attribute. Such an inconsistency will not be detected by a syntax error checker, since both the syntax and semantic specification are correct in terms of the specification language. Moving on to the ALF version, if the same error - the semantics (operations) using different names from the abstract syntax (class structures) for the same attribute - occurs, such an error will cause a syntax error (a name refers to a non-existent property) that is easier to be identified by a parser. Compared to a parser, the other ways usually requires additional methods. Figure 29 shows how the ALF editor can detect such an error.

```

39 public fire() {
40     for (Arc arcIn : this.inArcs) {
41         let place:Place = (Place)arcIn.source;
42         property token does not find in type ClassValidationType(name=Place);
43     }
44     for (Arc arcOut : this.outArcs) {
45         let place:Place = (Place)arcOut.target;
46         place.token++;
47     }
48 }

```

Figure 29: Checking non-existent properties.

Interoperability

As they are backed by the Object Management Group, are supported by many large tool vendors in the modelling community and have the existing resources of UML, ALF and fUML are considered to be interoperable technologies.

Since ALF is an interoperable format that is supported by industrials, it makes language specification reuse and language specification composition possible. If the PNTD, for example, were to be extended to a hierarchical Petri net, the classes and operations could be reused as normal object-oriented programmes with inherited classes, override operations and polymorphisms. This is more difficult if the base language specification is procedural based, or even defined using different languages.

Understandability

ALF is designed to be an understandable language for a wide group of users. Although it is not expected that users who are not familiar with programming and who are therefore not the intended audience of a language specification could understand it, advanced users, language tool engineers and language engineers are all target audiences of ALF.

Expressiveness

Expressiveness gives the language engineers the opportunity to create semantics in a higher-level language, instead of worrying about implementation. An expressive language does not guarantee that the users of the language will develop high quality products but, with the process discussed in Section 4.3.2 and the guidelines in Appendix B, it will certainly be beneficial when defining concurrent behaviours and working with sequences and associations, resulting in code that is smaller and less complex.

5.5 Summary

In summary, defining a DSL using FQLS includes defining a DSL as an executable meta-model, which is represented by ALF syntax. Abstract syntax is defined as classes and their relationships. Static semantics can be captured by using an aspect weaver, by extending ALF syntax, or by representing constraints in behavioural semantics. Behaviours are attached to the abstract syntax by direct attachment, using an aspect weaver, or by creating a runtime meta-model.

In order to demonstrate the method, a Petri net example was illustrated, first defined by translational semantics using a class diagram, a prose-based transformation and an ASM rules for behavioural semantics. The ALF specification of the same language is provided thereafter. Why the ALF-based specification is better in consistency, interoperability, understandability and expressiveness is discussed.

The next chapter discuss describes how static analysis is used in FQLS to maintain the correctness of a ALF-based language specification.

Chapter 6.

Static analysis of DSL specifications using FQLS

Using ALF as a method for defining language specifications leads to a more understandable specification that has improved consistency and correctness and which is more interoperable. However, these benefits still need the developers' efforts in order to achieve them. As discussed in Chapter 4, an automatic method of error reporting is an important aspect of creating a high quality language specification, which requires performing a static analysis of the language specification via the analysis layer of FQLS.

In this chapter, the types of errors that would occur in an ALF-based language specification are first discussed in Section 6.1. After identifying these errors, in Section 6.2, how these errors can be statically checked by FQLS are demonstrated, by using the built-in checkers that traverse the syntax tree and perform type checks. Thereafter, Section 6.3 explains how to transform an ALF specification into fUML, which enables the reuse of UML analysis approaches.

6.1 Extended static checking

This section tries to answer a question: What kinds of static analysis are required? Because a static analyser targets a particular kind of error by looking for a bug pattern, this question can be answered by examining the kinds of errors that can appear in an ALF specification. Research into the kinds of errors that can occur in an ALF specification starts by sourcing errors from two different categories, a static code analyser for other languages, and our experience of using ALF.

The first source of errors comes from the static code analysers for other programming languages, particularly those of Java. Java is selected as a major source because that ALF, as a programming language that is similar to Java, shares the same kind of errors as Java. Therefore, the errors that can be checked by Java code analysers are analysed.

The Java code analysers that are analysed are FindBug [70], PMD [27] and CheckStyle¹⁰. Each has a list of the errors they can check, which ranges from common causes of bugs and styles, to bad practices (or ‘bad code smells’, as described in [98]). If the same error occurs in ALF, the error is listed as a possible target for a static analyser.

Table 2 lists the errors that are migrated from established static checkers. Since they only requires efforts to implement, they are not worthy to be introduced in the main body of the thesis. The details of these errors are listed as Appendix C.

Name
Syntax errors
Class implements the same classes as super class
An Activity that has a return part must define a return statement
Type errors
Unused/unwritten class members or variables
Compare two objects when they are not comparable
Method names differ only in capitalisation
Class defines an attribute that masks a superclass attribute
Unnecessary type check done using instanceof operator
Obvious infinite loop
Impossible cast

Table 2: Errors identified from established checker

The other source of errors comes from our experience in using ALF. ALF has its own features that can cause errors, especially some tricky features that are syntactically similar to Java, but which have different behaviours in Java. When developing the BPEL case study, some errors that were encountered could occur repeatedly. These errors were identified as features of the static analyser; it was found that the effort involved in implementing the checker was less than that of finding them manually.

Table 3 lists the errors that are identified from the BPEL case study. Their details are introduced in the following paragraphs.

¹⁰ <http://checkstyle.sourceforge.net/>

Name
Wrongly placed accept statement
Accepting a signal that is not defined in a signal reception
An OCL expression cannot be interpreted.
Empty code block
Blocked active class thread by signal deficiency
Use of a local variable without declaration
Try to compare a collection with null
Access a collection by index when the collection is not a sequence
Ambiguous ! operator
Check java naming rules
Practical rules regarding sequence expressions

Table 3: Errors identified in the BPEL case study.

Wrongly placed accept statement

An accept statement is a special statement that will temporarily stop the execution of the active object, which will be resumed when the signal it is waiting for arrives. Since it stops the execution of an active object, it must appear in an active object. ALF constrains only the classifier behaviour of an active class, or a pure activity, because activity is treated as a classifier that is active. However, it is easy to forget this and to put accept statements into a normal operation. For example, moving a code block that contains an accept statement to a new operation, as in Listing 5, is incorrect.

```

public active class Test{
    public receive signal SignalTest{}
}do{
    accept(SignalTest);
    //do something
}

```

Listing 4: Correct accept statement.

```

public active class Test{
    public receive signal SignalTest{}
    private receiveSignal(){
        accept(SignalTest); //Incorrect
    }
}do{
    this.receiveSignal();
    //do something
}

```

Listing 5: Incorrect accept statement.

Accepting a signal that is not defined in a signal reception

This error happens when an accept statement tries to accept a signal that is not defined as a signal reception of the enclosed class. ALF has two ways of defining a signal reception; one is to define the signal and its reception in the same place, as below:

```

public active class A{
    public receive signal SignalTest{}
}do{
    accept(SignalTest);
}

```

The other way is to define the signal separately, as below:

```

public active class A {
    public receive SignalTest;
}do{
    accept(SignalTest);
}
public signal SignalTest{}

```

If an accept statement tries to accept a signal that is not a reception of the active class, it indicates either the signal reception is missing, or that the target name is wrong. For example,

```

public active class A {
}do{
    accept(SignalTest); //missing signal reception
}

```

An OCL expression cannot be interpreted.

Section 5.1.2 introduces the three ways that are proposed to specify static semantics via ALF. When defining static semantics by OCL separated from the ALF specification or embedding OCL to ALF, the OCLs can also experience errors. When an OCL interpreter cannot interpret the OCL expression, this error will be reported.

Blocked active class thread by signal deficiency

As discussed previously, an accept statement stops the execution of the active object that contains the accept statement. If the language engineers wish to continue the execution of that active object, another object must send the signals for which it is waiting. If such a signal is missing, then the active class is blocked due to signal deficiency.

It is not easy to eliminate blocked active classes, considering the complexity of a programme. However, it is possible to check a particular kind of deficiency.

```
public active class A{
    public receive signal SignalContinue{}
}do{
    accept(SignalContinue){
        //do something
    }
}
public active class B{
}do{
    let a:A = new A();
    a.SignalContinue();    //----- 1
}
```

For instance, assume that the code above is a complete collection of the ALF programme. The code block contains two active classes, A and B. The active objects will not be blocked, because an instance of B will send the signal to A when B is instantiated. If statement 1 is deleted, then class A is waiting for a SignalContinue. However, such a signal is not sent by any instances of the entire programme, which indicates an error of signal deficiency.

As a result, the check scans the complete programmes and creates an error message if an active class waits for a signal that is not sent by any other classes.

Use of a local variable without declaration

In ALF, the local name can be declared as in any dynamic language, when it is first assigned a value like

```
a = "fdsafsa"; // a was not declared before
```

In the assignment statement above, the type String is omitted. However, it is better to avoid it, because static type checks can reveal bugs. Use implicit type, especially when indicating the type of the variable is not easy, can make static type checks difficult.

Try to compare a collection with null

When testing whether a collection is empty, it is wrong to compare the collection to null. The correct way is to invoke the `isEmpty()` method. The code below shows an instance:

```
if (structuedActivity.activities==null) //not correct
if (strucutedActivity.activities->isEmpty()) //correct
```

Access a collection by index when the collection is not a sequence.

ALF defines a sequence using the syntax below:

```
public class Scope{
    public activities: Activity[*] sequence;
}
```

where a sequence keyword is the same as

```
public activities: Activity[*] ordered nonunique;
```

A property is non-ordered by default. This does not affect the processing of each element of the list if the order is not important. However, it does cause an error when accessing non-ordered multiple properties via an array index operator (“[“ and “]”). When this happens, no correct result is guaranteed: ALF open source reference implementation will return a value in the list, but it does not have the index of the desired one.

This check performs checks on the array access operator and reports errors if the target it tries to access is not ordered. In the Petri net example, the multiple properties are a non-ordered collection. Thus, the in `Transition.fire()`, accessing an arc by the index is wrong:

```
class Transition specializes Node{
    public fire(){
        let firstArc:Arc = this.inArcs[0]; // wrong
        //. . .
    }
}
```

Check java naming rules.

This rule checks whether a name in the ALF text conforms to Java naming rules. It is useful when generating the testing executor. Since the naming rules of ALF and Java are different, it is possible that a legal name in ALF will cause a syntax error or an ambiguity error. An obvious example is that it is legal to name an ALF variable “try”, which will result in the generated code not being compiled. Another unobvious example is to give an ALF class the same name as a class in `java.lang` package, such as ‘Process’, which will

result in the generated code use of `java.lang.Process` in default, rather than using the `Process` class defined by the language engineer.

The rule contains the follow sub rules:

- Names are not reserved Java names
- The first letter of the class is capitalised
- The first letter of variables/operations is not capitalized.
- Signals start with a keyword signal.

Practical rules regarding sequence expressions

The sequence expressions of ALF are powerful because they give ALF the same expressiveness as OCL. Sequence expression can be easily defined as lambda expressions. However, current Java does not support lambda expressions¹¹, which makes generating Java code from ALF sequence expressions a difficult task. It can be done by generating an interface with an anonymous function, which will damage the generated code because each sequence expression results in an interface with a randomly generated name. Some JVM based Java-like languages, such as Scala and Xtend, support lambda expression; however, using a language other than Java makes the reuse of EMF impossible. Finally, FQLS supports the generation of sequence expressions as OCLs embedded in Java with limitations. The details of the generation are introduced in Chapter 7. In order to enable our approach of code generation, the following conditions must be satisfied.

- Avoid sequence expressions in a normal ALF block.
- Put sequence expression into a new operation that only has a return statement.

For example, such sequence expressions should be avoided:

```
public operation () {  
    //some code  
    let flag:Boolean = sequence -> forAll e (e>0);  
    //some code  
}
```

However, sequence expressions can be used if they are defined in a separate method by generating sequence expressions as OCLs; for example, the list below will generate working EMF code in FQLS.

¹¹ Java aimed to support closure by project lambda (<http://openjdk.java.net/projects/lambda/>) in Java 7 (2011); however, it was delayed until Java 8, which was released in 2014 - clearly too late for this thesis.

```

public operation(){
    //some code
    let flag:Boolean = createFlag(sequence);
}
private createFlag(in sequence:any[*]):Boolean{
    return sequence->forall e (e>0);
}

```

Thus, a checker can be implemented to check whether the use of a sequence expression meets the limitation. This option should be turned on if generating code for Java 7 or below is desired.

6.2 Building static code analysers

Once the errors that need to be checked have been identified, the next step is to build static checkers that validate these errors. Many syntactic errors mean that the programme cannot be parsed as a valid syntax tree. These kinds of errors do not need the implementation of a validator, because the ALF editor included in the tool chain of FQLS is built upon Xtext, which automatically generates a parser that validates lexical errors. The details regarding the way in which Xtext generates a parser are omitted, as this is outside of the scope.

On the other hand, many of the errors listed in Section 6.1 should be checked by building customised validators. Fortunately, Xtext provides a plugin mechanism to add customised validation rules in a declarative style. A class that generalises `AbstractDeclarativeValidator` can add a `@Check` annotation to an operation. The input of such an operation is a class from the language meta-model. In the body of the operation, the input class instance is examined using any Java methods necessary and, if any error or warning is found, the operation should tell the editor to report specific error messages. For example, the list below shows a validating rule that checks whether an operation of an ALF programme should have a return statement. `OperationDefinitionOrStub` is a class from the ALF meta-model. If it has a return part (which means the operation should return a value) and the body of the operation does not have a return statement, the validating rule will report an error.

```

@Check
public void validateOperationDefinitionOrStub(OperationDefinitionOrStub
operation){
    AlfFinder finder = new AlfFinder();
    if (operation.getReturnPart()!=null
        && (! finder.hasReturnStatement(operation))){
        error("The operation must return a result", operation, null,
            INSIGNIFICANT_INDEX);
    }
}

```

Listing 6: Validating rule for testing the return statement.

When the ALF editor is running, the Xtext framework automatically invokes the validating operations with relevant input. For example, every instance of `OperationDefinitionOrStub` will be passed to the validating rule in Listing 6 and be validated. The mechanism of the Xtext validator provides a framework to perform validation on special elements of the ALF meta-model, so the developers only need to implement how to identify the errors, rather than how these rules are invoked or how the errors are rendered in the editor.

The validation framework of Xtext enables the addition of customised validation rules to a language in a simple way. It is necessary to mention that, although it is called a declarative validator, the logic for querying the ALF models, finding errors and reporting errors is still performed by imperative code. The errors listed in Section 6.1 can all be validated by validating operations that range from several to hundreds of lines of Java code.

The validators and the helper classes form a Java package. To demonstrate how a validator is implemented, the validator that relates to type errors is taken as an example. The class diagram is illustrated in Figure 30. It was chosen because it is the most complex validator, and the helper classes are used by other validators.

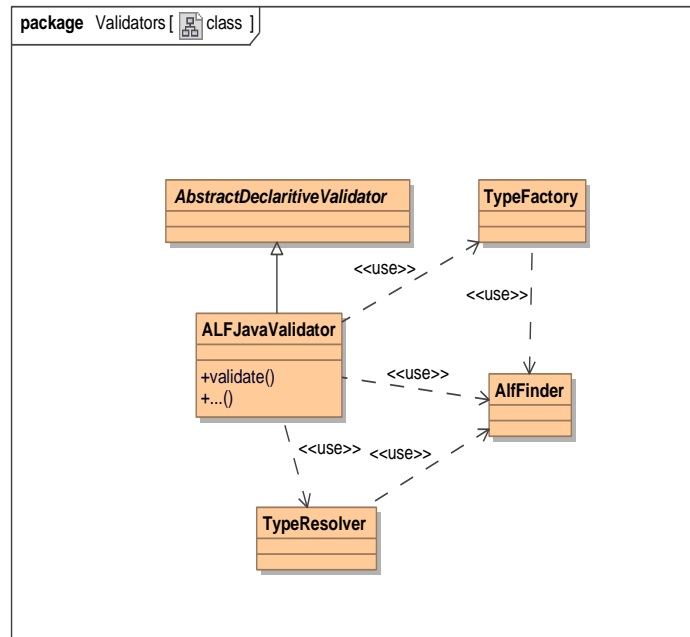


Figure 30: Validation package.

`AbstractDeclarativeValidator` is a base class provided by the Xtext framework. Its subclasses can use the `@Check` annotation and are managed by the Xtext framework. `ALFJavaValidator` is an example that contains validating operations.

Because many different rules need to query the language specification, the queries are separated in the `AlfFinder` class. `AlfFinder` provides helper operations that, when given an input model from the language specification, return a special model that can be used for analysis purposes. For instance,

```
public Statement findStatement(EObject exp)
```

This operation will find the statement that contains the input object. Some statements are only correct when their enclosed expressions are of a certain type. Here is another example:

```
public EObject findNameDefinition(NameExpression exp)
```

When given a name expression, it is necessary to find its definition; whether it is a local variable, a class member, or a parameter. If there is no definition of the name expression, it returns null.

To check type errors, the complete ALF instance model is first traversed. The models that could be a type are identified and their types are recorded. Basic types that are supported by the ALF standard are included, as well as any classifiers that can be used as a type. Each of the expressions is then scanned, and the type of the expression is resolved by the class `TypeResolver`.

`TypeResolver` is a class that computes the type of the input expression. When the input expression has type errors, it will recode the errors, which can then be retrieved by the callers. The `TypeFactory` class provides the relevant utility operations, such as creating a specific validation type, or the static operations that compare two types to see whether they are compatible.

By using these classes, rules related to type checks can be implemented. The implementation details are omitted, since they are not innovative. On the other hand, the numerous small checkers, as a whole, enhance the quality of a language specification.

6.3 Bridging FQLS specification with fUML

Since the built-in checkers validate types and syntax trees, it is possible to create a checker for most of the errors listed in Section 6.1. While these checkers are targeted for the most frequent errors, these checkers take ALF programmes as input. They can only analyse errors by querying the ALF programme. In other words, the errors and bad practices targeted by the built-in static checkers share the same feature, in that they can be identified by querying the syntax tree of the ALF programme.

Formal static analysis method can verify some properties like deadlocks or inefficient use of resources. They are also desired because such kinds of errors reduce the quality of the language specification. Moreover, the result they produced is usually sound, which gives extra confidence of the quality. In order to perform formal analysis on a language specification specified via ALF, it is needed to bridge ALF to an analysis domain.

A direct mapping from ALF to an analysis domain can be created. Since there are many possible candidates, each of the mappings requires significant effort to develop. In addition, such developments require the formal semantics in order to transform them. The semantic basis of ALF is fUML, which means if a transformation from ALF to an analysis domain is desired; this can be done by firstly transform to fUML, and then transforms fUML models to any desired analysis models.

There is another reason why fUML is a promising candidate of analysis domain. Since the release, significant works have been done on how to formally analyse or to verify a fUML models, which are listed in Table 4, specifying the property that can be verified and the input model required for that approach.

Name	Input model	Analysis
[20, 21]	UML/EMF	Satisfiability, Liveness, deficiency, redundancy
[158]	fUML	Data flow analysis
[118, 120]	UML/fUML	Executability, satisfiability
[2]	fUML	Deadlock Checking
[11]	fUML	Performance analysis
[90]	fUML	Control- and data-flow, resources, and time dimensions

Table 4: fUML analysis approaches

The establishment of fUML analysis approaches means if an approach that bridges ALF and fUML exists, the fUML analysis approaches can be reused, thus saves the effort to develop the mappings individually. Fortunately, mapping from ALF to fUML is defined in its language standard. However, the transformation is not a straightforward implementation. The transformation, rather than being defined in a rigid way, is defined by text, which needs to be interpreted and formalised. In total, ALF and fUML contain hundreds of classes, and mapping is not a simple mapping. One ALF element can be mapped to a group of fUML nodes, edges, structured nodes and classifiers, and these fUML elements can be owned by different groups. Thus, the nature of ALF makes the transformation complex and challenging.

In order to create such a transformation, the following parts of this section discuss the choice of model transformation languages, and then how the transformation is created and the results are demonstrated.

6.3.1 Atlas Transformation Language

To develop such a transformation, it is necessary to select a model transformation language. There are several commercial or open-source projects that can be used to transform models, as shown in Table 5. They share similar features, such as providing a mechanism to define transformation rules. Most of them support so-called *hybrid transformation*, which enables the use of both declarative and imperative styles to define transformation rules.

Name	Feature
ATL [73]	Widely used, supports hybrid rules, actively developed by the Atlanmod team.
QVT [53]	OMG standard. Contains three languages. Supports hybrid rules
Kermeta [18]	Supports imperative style transformation.
Epsilon m2m [82]	Has the ability to interoperate with other members of the Epsilon language family.
Henshin [7]	Uses in-place graph transformation to define declarative rules

Table 5: Transformation languages.

Of these transformation languages, ATL and QVT are the most acceptable model transformation languages. While QVT is a language defined as an OMG standard, which forms the model-to-model part of the MDA process, QVT is somewhat complex as it is split into three different languages, namely QVT relational, QVT operational and QVT core. Within three of the languages, only QVT operational has a usable open-source implementation in the Eclipse ecosystem. In comparison, ATL is a model transformation language that has a large user community. It supports hybrid rules, which are powerful enough for a complex transformation, yet still keep the transformation simple to read. Hence, ATL is selected as the language for implementing the transformation.

Although ATL is selected as the means of implementing the transformation, this does not mean that the idea can only be realised by ATL. QVT and any other mature model transformations can do the job. The model transformation implemented in this section is a demonstration of a possible solution.

6.3.2 Mapping ALF to fUML

A model transformation needs to define its input/output meta-models, the source/target models, and the transformation rules. Figure 31 shows an overview of the transformation. This process involves four rounds of transformations, and three meta-models are used.

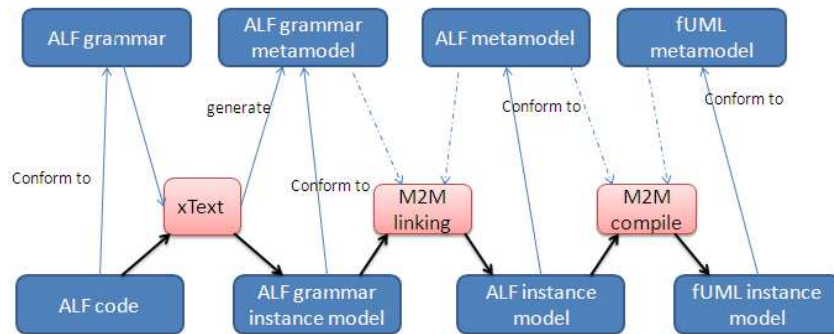


Figure 31: ALF text to model transformation.

The ALF meta-model and the fUML meta-models are Ecore implementations of the standard, imported from the CMOF files given on the OMG website by the EMF standard import wizard. The errors that happened in the importing process were solved manually.

The transformation starts from the ALF text. The first round transforms from text to the ALF grammar model, which is done automatically by Xtext. A model that conforms to the grammar meta-model is passed to the transformation from the definition layer of FQLS.

The grammar model is different from the ALF meta-model defined in the standard, because it still contains the verbose classes that are only meaningful for parsing. Thus, the next step is to translate the grammar model to ALF meta-model. This process, if compared to traditional language development, is similar to constructing an abstract syntax tree from a syntax tree.

The grammar of ALF is divided into expressions, statements and units. When constructing the grammar, the units and statements are identical; thus, the transformation rules mainly copy the contents. However, the expressions of the grammar model and the ALF meta-model are different. To ensure that the grammar is an LL(*) grammar¹² that can be parsed, the class representative of expression is verbose. For instance, an expression “operand1 + operand2” in the grammar model will be:

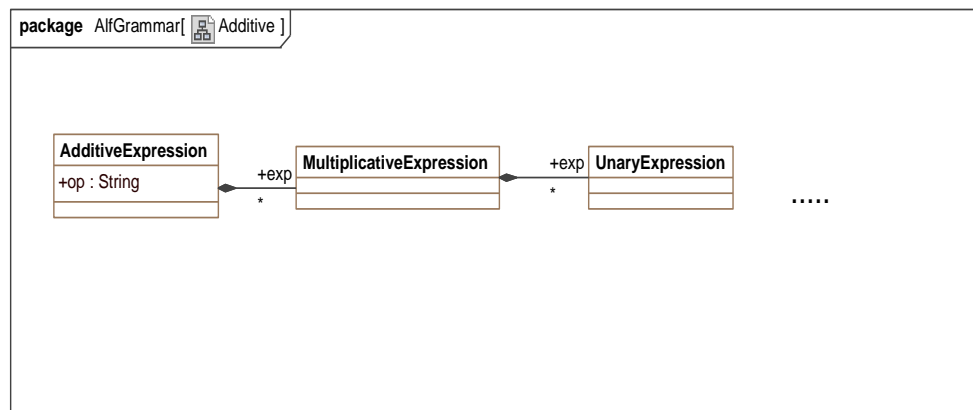


Figure 32: An example in the ALF grammar meta-model.

In the ALF meta-model, expressions are resolved to a proper representation:

¹² Xtext uses ANLTR 3 as its parser generator, which supports LL(*) grammar.

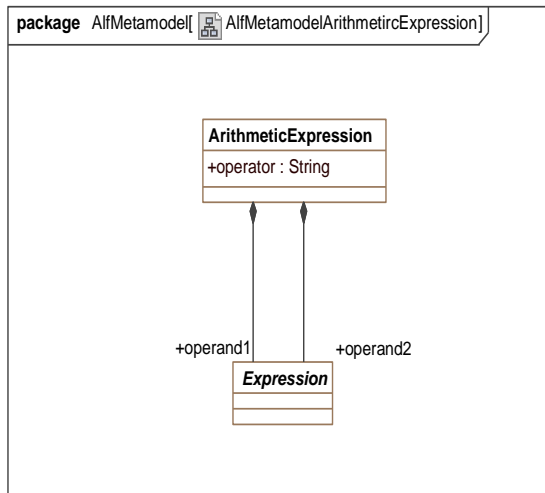


Figure 33: Arithmetic expression models in the ALF meta-model.

The M2M linking transformation mainly contains the transformation that resolves the expressions. This transformation will create an ALF instance model, which is ready to be transformed to fUML.

The last round of transformation compiles the ALF instance model and the fUML instance model. The ALF standard has defined how the mapping should be done. Although this is informative, it is not intuitive if implemented by a model-to-model transformation language.

Table 6 summarises the simple ALF programme and its corresponding UML representation. This table helps to give an idea of how concepts from ALF are translated.

ALF code	fUML models
Activity	Activity with parameters mapping to ParameterNode.
Code block	Maps to a StructuredActivityNode, with each of the statement inside mapped to multiple StructuredActivityNode that are linked by control flows.
Parallel code block	Maps to a StructuredActivityNode similarly to a normal code block, but without the control flows.
Literal expression	Maps to ValuSpecificationAction

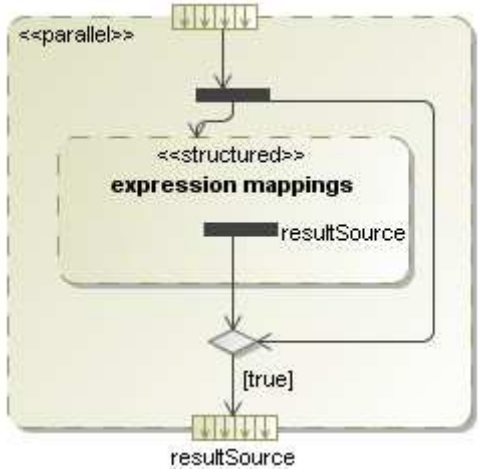
binary expression	The operands are mapped to fUML as normal. The result of the operands are lined to a BehaviouralInvocationAction that invokes the corresponding calculation action defined in the standard model library.
method call	Maps to a BehaviouralInvocationAction.
return statement	The expression part of the return statement is mapped to nodes and edges that are enclosed in a StructuredActivityNode. The result source of that expression has a data flow linked to the return parameter node of the activity.
sequence expression sequence -> select variable (expression)	 <p>Sequence expression is mapped to a ExpansionRegion, as the example figure.</p>
signal sending	Maps to a SendSignalAction
Signal receiving	Maps to an AcceptEventAction
Loop Statement	Maps to LoopNode with the body maps to the body of the LoopNode, and the test codition maps to the Condition of the LoopNode.

Table 6: Mapping between ALF code and fUML models.

The table shows that one ALF concept can be mapped to a group of UML nodes and edges. However, the containers for these UML nodes and edges that are derived from the same ALF class can be different. For example, in the standard (page 347), mapping expressions indicates that an expression statement is mapped to a structured activity node that contains the activity nodes and edges that are transformed from the expression. This is realised by identifying the owner of the target models when writing the ATL rules.

By using this transformation, ALF text can be transformed into fUML models, and can be analysed or tested in a fUML executor if the language engineers desire. Returning to the Petri net example, for the operation `getActiveTransition()`, the code is

```
public getActiveTransition():Transition{
    return this.getTransitions()
        ->select e (e.isActive())->at(0);
}
```

Figure 34 shows the transformation result of the operation. The operation `getActiveTransition` is mapped to a UML operation, of which method body is an activity that contains the nodes and edges. The operation contains one statement (a return statement), which maps to a structured activity node that contains the nodes and edges that are derived from the expression of the return statement. This expression is mapped to a `ReadSelfAction`. The call to `getTransitions()` maps to a `callOperationAction`. The selected expression maps to an `expansionRegion`, with the conditions mapped inside the `ExpansionRegion`. The `at()` operation is an operation that comes from the standard fUML library, which maps to a `callFunctionAction` that return the value with the correct index.

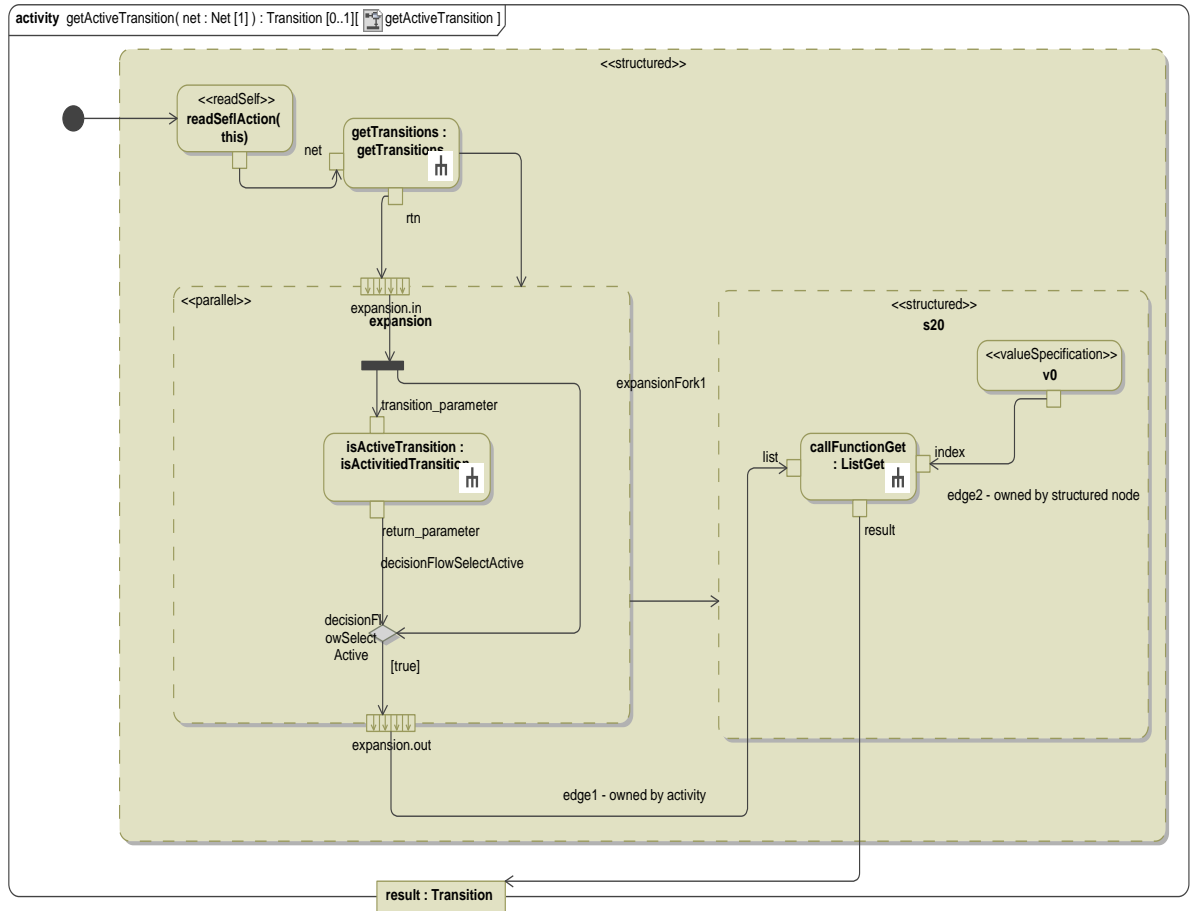


Figure 34: Compiled fUML model of getActiveTransition.

6.4 Summary

In this chapter, the components of the analysis layer of FQLS were presented in detail. The errors that could occur in an ALF specification were identified, and ways of building these checkers using Xtext were discussed. Thereafter, a transformer from ALF to fUML was proposed and demonstrated by implementing it as an ATL transformation application.

Chapter 7.

Executing DSL specifications using FQLS

In the previous chapter, the static analysis of the language specification was performed without executing the specification. Although the process of detecting the problems is automatic, these static analyses are limited by checking syntax errors, bad practices and some semantic errors. The semantic errors that are not easily analysed by checkers require the language designers to identify them. When the semantics specification reaches a sufficiently complex stage, it is difficult to verify its real execution result manually. On the other hand, an executable semantics specification will provide an automatic method for language developers to derive the execution results.

Executability of the language specification is one of the most important requirements as it enables testing and thus enhances the quality of the language specification. Testing a language specification involves creating testing programmes of the language specification, then executing the programmes and checking whether the result is as expected. Thus, the task of testing language specification not only involves an executor, but other tools as well.

Just because a language specification is executable does not mean that software that can execute the language specification exists. A language specification defines the language executor at the conceptual level, and is then implemented as a piece of software.

In practice, language specification is the guidance the language engineers use to create a reference implementation of the language. A reference implementation of a language specification should

- Give the language designer a tool for testing the language specification.
- Give the readers a standard tool to understand the language.
- Give the tool developers a starting point that they can use as a basis for creating other tools.

However, the relationship between a language specification and its implementation does not happen automatically. The release of a language specification is not required to be accompanied by a reference implementation. In fact, it is possible that a language

specification, or at least part of it, is never implemented, even by the time it replaced by a newer version.

Given the importance of a reference implementation, one reason for the absence thereof is the cost. If the language specification is composed of text, a reference implementation needs to be developed manually. Defining the language specification as an executable meta-model enables the language specification to act as a reference implementation. However, this is only possible if the language specification is sufficiently detailed for execution and if certain tools are available.

As discussed in Section 4.2.3, FQLS uses code generation to create a reference implementation of the language specification. Section 7.1 introduces the architecture of the code generation project. Section 7.2 demonstrates the transformation from ALF structural aspects to Java code. Following this, the last section discusses the method of transforming behavioural aspects of ALF into Java code.

7.1 Architecture of the code generation project

Directly generating Java from ALF is time-consuming and can involve a large degree of development because there are many features of ALF that do not have straightforward mapping. It is not worth reinventing features, such as ways of managing bi-directional references, as when one end changes the other end does too. Thus a solution is to generate an Ecore model from ALF. Ecore models are necessary for generating DSL model editors and validators. Furthermore, EMF can generate the skeleton of the Java code that represents the Ecore model. Then Java statements can be generated directly from ALF statements. The Java statements use the skeleton generated by EMF.

fUML and Ecore share many similarities when defining structural models. Although some classifiers, such as signals/associations do not have a direct translation, they can be translated to similar concepts. For instance, associations can be translated to references. Thus, it is possible to develop a similar model-to-model transformation that bridges ALF and Ecore. However, Ecore lacks the ability to define behaviours, which makes it impossible to translate ALF to Ecore without losing behavioural model definitions¹³.

¹³ The experimental EMF project Xcore includes basic variable assignment, loops and conditional statements, which could be used to define detailed behaviours. However, Xcore has limitations in many places, such as lacking of concurrency support, no exception handling and limited documentations.

In order to support the reference implementation generation, the ALF to Ecore model transformation is designed as a model-to-text transformation, rather than as a model-to-model transformation. The reference implementation requires generating Java code from ALF, which is more suitable as a M2T transformation. The generated code is coupled with the Java skeleton generated by EMF. If the structural aspects of ALF are transformed to Ecore using an m2m transformation while the behavioural aspects of ALF are transformed to Java using an m2t transformation, it is too complex to integrate. Hence, it is more suitable to create an M2T transformation that deals with both the structural and the behavioural aspects of the language specification.

The following two paragraphs introduces Emfatic and Acceleo, both of them are key technologies that enable the transformation. After that, the architecture of the transformation is explained.

Emfatic

Emfatic [29] is one candidate for the textual representation of Ecore, and there are many other choices. It is chosen because it supports Ecore annotations well. It directly supports adding Ecore annotations to the Emfatic files, which means an Emfatic file can contain the structure of the models, the OCL and the body of the operations. The body of the operations is specified in @Genmodel annotations as Java statements. Thus, this Emfatic file acts as an intermediate format. In addition, the Emfatic file can be directly loaded by EMF, and the code generation is smoothly integrated, just as in generating Java code from a normal Ecore file. Although a similar effect can be achieved by the other textual representations of Ecore, the author believes Emfatic provides the best solution as it does not need to cross check between the textual representation of Ecore and the genmodel.

Acceleo

Acceleo¹⁴ is a code generation language. Compared to similar languages, Acceleo is an implementation of the OMG M2T standard [47], which means its syntax is template-based, uses OCL as its querying language, and is compatible with the standard. It is possible to implement the M2T transformation in any other M2T language, such as JET¹⁵, AndromDA¹⁶, Xtend¹⁷, or Epsilon Generation Languages [129]. The various M2T languages have their own features, but their syntax is not compatible with the OMGM2T

¹⁴ <http://www.eclipse.org/acceleo/>

¹⁵ <https://www.eclipse.org/modeling/m2t/?project=jet>

¹⁶ <http://www.andromda.org/index.html>

¹⁷ <http://www.eclipse.org/xtend/>

standard. Since interoperability is a fundamental requirement of a language specification, building the complete framework by technologies that are compatible with industry standard will make them easier to adapt and understand.

Architecture

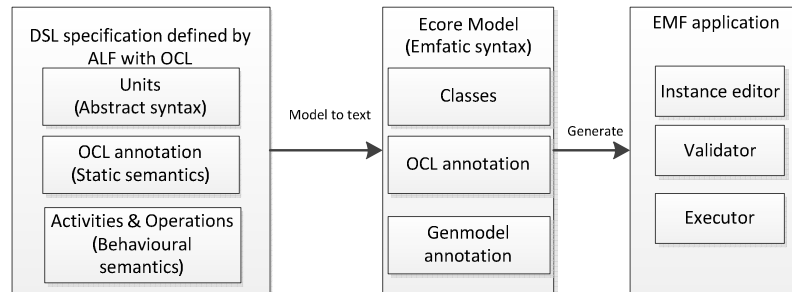


Figure 35: Generating Java code.

As illustrated in Figure 35, the code generator works as two-round code generation. The first round transforms FQLS models to an Ecore model. The structural definition in ALF can be translated as one-to-one mapping, and the behavioural definition is translated to Java code embedded as the body of operations. The transformation is done by using Acceleo M2T, which creates an Emfatic file. The second round of transformation is the standard code generation of EMF, which generates Java code from the Ecore model.

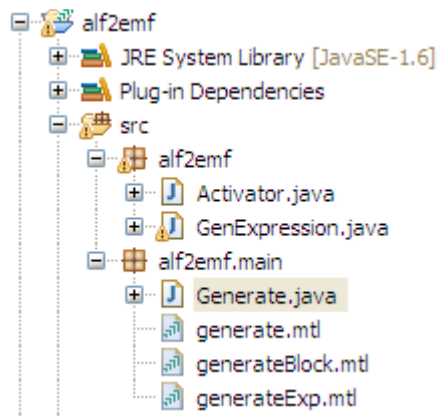


Figure 36: Project structure of the executor generator.

Figure 36 above shows the structure of the Acceleo code generation project. The package `alf2emf.main` contains the code generation templates. `Generate.java` is the class that initialises the code generation framework, which loads the input ALF mode and loads the `.mtl` templates. There are three template files, namely `generate.mtl`, `generateBlock.mtl` and `generateExp.mtl`. As the names suggest, `generate.mtl` is the starting point for the code generation, which defines the location of the output Emfatic file. It also contains the generating structural aspects of Emfatic files, namely the classes and

class members. Finally, `generate.mtl` also generates some Java files, which act as glue code or adaptors.

`GenerateBlock.mtl` maps ALF statements to Java statements. When an ALF statement contains expressions, it also invokes the `generateExp.mtl` template, which transforms ALF expressions into Java Expressions. `GenExpression.java` provides some low-level detailed code generation tweaks.

This and the next section explain the details of the code generation. They are divided as two parts: the structural aspects, and the behavioural aspects.

7.2 From ALF structural aspects to Emfatic

This section explains the mapping of structural aspects of ALF to Emfatic in detail. The concepts in ALF that are similar to Emfatic are listed firstly, following by the mapping of advanced concepts.

When representing classifiers, ALF has a richer vocabulary than has Ecore. Class, enumeration and datatype have correspondent concepts in Ecore.

Table 7 summaries a general mapping of structural concepts from ALF to Ecore. Basic types are mapped from ALF types to Ecore types. Multiplicities are mapped in the same way as in ALF, because they are the same in terms of concept and differ only with regard to syntax.

ALF structures	Emfatic structures
Class	class
signal/signal reception	class extends Signal
active class	class extends ActiveClass
Enum	enum
Attribute	attribute
Association	reference
Operation	operation
Parameter	parameter

Table 7: Mapping from ALF to Emfatic.

Signals/signal reception

The table also shows that some concepts, such as signals and active classes, do not have corresponding concepts. To solve this problem, a base class for all the classifiers that do not have a corresponding Ecore concept is created. Consider the signal definition in an

ALF programme. In reality, signal instances are special objects that can receive/send signals by using an invocation expression or an accept statement. They do not have any special properties; thus, they can be represented as a class. By creating an interface called Signal and then transforming an ALF signal definition to an Ecore class that implements the Signal interface, a signal definition can be simulated by an Ecore class.

Active class

Another, more complex example, is transforming the active class in ALF. The active class starts its classifier behaviour when a new instance of the active class has been created. The most similar concept in Java is a thread. The active class also needs to manage incoming/outgoing signals, as it is possible to receive signals before the object goes to a waiting signal state. As the internal communication mechanism is a semantic variation point of fUML, the detail of inter object communication is left to the tool implementer to interpret. A simple message exchange model is used, whereby each active object has a queue that stores the incoming signals. Once an accept statement has received a particular signal, that signal will be removed from the queue. Finally, a Java interface can be implemented:

```
public interface ActiveClass extendsEObject,Runnable {
    EList<Signal_> getMessageQueue();
    void run();
} // ActiveClass
```

Thus, when generating Emfatic code from ALF, an active class is mapped as a normal class that extends the ActiveClass interface:

```
abstract class Execution extends ActiveClass {...
```

The classifier behaviour of the active class is mapped to the content of the run() method. When an active object is created, a new thread that executes the run() method is created in the generate Java implementation.

An ALF active class can define signal receptions, and other objects can send signals to the active object. The syntax for sending a signal to an active object is the same as invoking it to an operation, for example

```
public active class Test {
    public receive signal SignalTest{}
}do{}
```

Assuming 'test' is an instance of Test, sending a SignalTest to test can be written as

```
test.SignalTest();
```

Thus, signal invoking is mapped as an operation that adds a signal instance to the message queue of the active object. This means that a signal reception is transformed into an operation with the same signal name; when invoked, the operation causes the same effect as being sent a signal, adding a signal instance to its own message queue.

Attribute/association

As shown in Table 7, attributes in ALF and Ecore are the same concept; thus, an attribute in ALF is mapped as an attribute in Ecore that has the same name, type and multiplicity. In the Petri net example, `Place.initialmarking` is defined as

```
public initialMarking:Integer;
```

and will generate Emfatic code like this:

```
attr Integer initialMarking;
```

Translating ALF associations is a direct transformation. In ALF, associations are classifiers that have the same root class as classes, signals and data types. An ALF association has its own name and visibility in addition to the other properties of a classifier, and it can have multiple associations.

In the previous sections, a limitation is set that only bi-directional association is allowed in an FQLS specification. One reason is the difficulty of mapping it to Ecore and, most importantly, a multiple association does not have a semantic basis in fUML. If multiple associations are eliminated in ALF, the closest assembly of an ALF association is a bi-directional reference in Ecore, which can only be a member of a class. Thus, given the association ends of an ALF association, the transformation needs to identify the correct Ecore classes that contain the reference ends. A heuristic is applied to identify relevant associations and to generate references in the Emfatic file. When translating an ALF class to an Ecore class, the code generator transforms the attributes first. It then looks up the complete model, and finds those associations that have one association end that is the target class. Depending on the associations found, the generator finally generates the references.

In the Petri net example, the associations between `args` and `nodes`:

```
public assoc Arc_Node_Source{
    public source:Node;
    public outArcs:Arc[*];
}
```

```
public assoc Arc_Node_Target{
    public target:Node;
```

```

    public inArcs:Arc[*];
}

```

will be translated to the following Emfatic code:

```

class Arc extends PnObject
{
    ref Node#outArcs source;
    ref Node#inArcs target;
}
abstract class Node extends PnObject
{
    ref Arc[*]#source outArcs;
    ref Arc[*]#target inArcs;
}

```

7.3 From ALF's behavioural aspects to Emfatic

This section continues the explanation of the code generation from ALF to Java, focusing on the behavioural aspects of the mapping. Firstly, the scope of the mapping is defined. Following that, the mappings of statements and expressions are explained.

In Java, methods are the only way of defining detailed behaviours of the programmes. By contrast, ALF inherited the UML method of defining behaviours. The basic unit for defining behaviours is 'activities', which is equivalent to the concept of a Java method that also has parameter lists and a body block that contains statements. Activities are the basic unit of behaviours, can be invoked by invocation expressions, and do not need to be wrapped by a class.

An ALF operation is a member of a classifier. An active class not only has operations, but also contains the classifier behaviour. Although operations and classifier behaviours are defined in a similar way to activities in terms of syntax, they are actually a definition of the operation and a definition of an activity. Considering this difference, various basic behaviour units need to be represented in Java without losing their original meaning.

As indicated previously, there is no perfect way of transforming ALF to Java without adding specific information or losing information, because this is the nature of implementing a specification. A code generator generates code automatically, but there are still many implementation decisions that are based on the developers' choice. Although it

would be nice to provide all options and consider all issues regarding the transformation of logic, as well as the implementation efficiency thereof, this is more of a development issue and, as such, is of no benefit to this thesis.

Therefore, the code generator's scope is limited. This code generator is to make the specification testable, to prove the idea that it is possible to generate a language prototype with limited effort, and to suggest that this prototype can give language designers a better way of testing and analysing semantic errors. The code generation attempts to retain the original meaning; however, when it is necessary, the generated code may have to apply an inefficient or naïve implementation. It tries to achieve this by making the look and feel of the generated code similar to that of the original ALF code, which means the generated Java code is similar to the original ALF code, as they have the same structures. By making the ALF and Java codes similar, it is easier to debug the code generator. In addition, when implementing abstract activities, having clearer codes makes development easier.

Following the principles discussed in the previous paragraph, the next paragraphs introduce the details of statement generation. most of the statements are revealed to be translated to their Java equivalents, while concurrent statements are transformed in a simple way to preserve the original statement structure.

7.3.1 Generating statements

An ALF operation is mapped to a Java operation because they share many similarities. The parameter lists were also translated, which generated a similar Java operation from the ALF operation. When translating classifier behaviour, because an ALF active class is translated into a Java class that implements the `Runnable` interface, translating classifier behaviour to a `run()` operation that realises the `Runnable` interface will result in a similar effect to that of an ALF active object.

Some activities do not belong to any class, since they can be invoked without an instance, which is similar to a public static method in Java. Thus, all the standalone activities are translated to the public static method that is contained by the class `GlobalActivities`.

Table 8 shows that many statements can be transformed simply by some syntax tweaking, while others, such as break statements, do not require any transformation at all. The challenging aspects are translating the statements that Java does not have and translating expressions contained in the statements, since some ALF expressions cannot be directly mapped.

ALF statement	Java code
normal if statement	if (condition) {} else {}
while (condition) do {statements} while(condition);	while (condition) do {statements} while(condition);
normal for statement for (Type iterator in collection)	For for (Type iterator : collection)
local name definition let a:Type = initialValue;	Type a = initialValue;
break;	break;
return expression;	return expression
inline statement /*@inline(language=Java) statements */	statements

Table 8: Mapping from ALF statements to Java.

One ALF statement is mapped to a Java statement, or to a segment of a Java programme that simulates the function of the ALF statement.

Accept statements

Accept statements that are simulated in Java by using a ‘while loop’ continue to check the content of the `MessageQueue`, defined in the `ActiveClass` class. When another thread has invoked the `signal-received` method and the signal received is one of the signals for which the current active object is waiting, the block of the compound accept statement (translated to Java) is executed, and then breaks out of the while loop. The following code shows an example:

```
accept (sig1:StartSignal){
  //code block 1
} or accept (sig2:EndSignal){
  //code block 2
}
```

is translated to

```
while(true){
  if (messageQueue.size()==0){
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {}
  }else{
    Signal_ signal_ = messageQueue.get(0);
    messageQueue.remove(0);
    if (signal_ instanceof StartSignal){
```

```

        //code block 1
        break;
    } else if (signal_ instanceof EndSignal){
        //code block 2
        break;
    }
} //if message
} //while true

```

Annotated statement/concurrent ‘if’ statement

The statements that deal with concurrency are the parallel annotated statement and the concurrent ‘if’ statement. Translating them into Java while preserving their concurrency requires implementing each of the concurrent statements as a new thread starts with a new implementation of an anonymous `Runnable` inner class. However, such an anonymous inner class can only access final local variables; ALF does not have such constraints. It is possible to design a mechanism that passes relevant local variables to it - one possible way is to create a local final array that contains the referred local variable, and which passes the array instead of the variable itself. However, such a mechanism is error-prone, and makes the generated code much more complex than that of the original ALF. This is conflicts with our purpose, which is to generate similar code.

Finally, in this prototype, the parallel statements are translated as normal statement blocks. Concurrent ‘if’ statements are transformed to a set of normal ‘if’ statements. This means that such concurrent statements are actually implemented as a sequence execution, which is not in conflict with the ALF language standard, since it does not constrain methods of implementing real concurrency. On the other hand, the omission of real concurrent behaviours in the implementation is definitely a limitation of the current code generator.

7.3.2 Generate expressions

The approach to generating expressions has the same principle as that of generating statements. The `generateExp.mtl` template reconstructs Java expressions from the ALF expression model, because of the difference between Java and ALF syntax. It adds syntactic changes to the Java code, because the code generated from expressions interacts with the code generated by EMF. In the following paragraphs, the mapping from ALF expressions to Java expressions is introduced.

Named expressions, such as classifier names, local variable names and parameters are printed exactly as is.

Value specifications, like numbers, strings and enum values, are printed.

An *invocation expression* tries to invoke an activity that is mapped to a method call of the corresponding static operation in the `GlobalActivity` class. If it tries to invoke a signal reception or a normal operation call, it is mapped to a call to the relevant Java method.

An *instance creation expression* (the ‘new’ operator) is mapped as a call to the creation method of the factory class.

```
let n:Node = new Node();
```

will generate

```
Node n = GenptnetFactory.eINSTANCE.createNode();
```

If the new object is an active object, the relevant Java code also initialises a thread that executes its classifier behaviour.

An *ALF property access expression* is mapped to a call to its getter Java methods.

```
//assume node is an instance of the Node class
```

```
let a:Arc = node.source;
```

is translated to

```
Arc a = node.getSource();
```

In the same way, an *assignment expression*, of which the left hand side is a property, is mapped to an invocation of its setter method.

```
//assume a is an instance of Arc
```

```
node.source = a;
```

is mapped to

```
node.setSource(a);
```

Since EMF uses lists to represent multiplicity, the index access expression in ALF, which is the “[” “]” syntax for accessing an index of a collection, is mapped to a call to `List.get(index)`. Similarly, a call to the standard model library of ALF is mapped to a similar method in Java.

```
//assume arcs is a sequence collection
```

```
arcs -> isEmpty();
```

is mapped to `arcs.isEmpty()`

`GenExpressions.java` provided the helper methods, such as string manipulation (capitalise/lowercase names, delete empty spaces or format comments), and complex condition tests that can be invoked by the code generation templates.

Generating OCL expression

One challenging type of generation is how to deal with sequence expressions in ALF. Sequence expressions in ALF are OCL-like, such as select/collect/forAll/exists expressions. Because Java does not support lambda expression, sequence expressions are difficult to translate into Java. Because the current version of JDK does not support lambda expression, there are two ways to achieve the same effect as an ALF sequence expression in Java.

The first is to create a code block that produces semantically equivalent results using loops and condition statements. However, considering that sequence expression can be chained (Listing 7), or can be nested (Listing 8), implementing sequence expression chain or nested sequence expression using loops will create heavily nested code. For example, the chained expression will generate a three level nest for each loop; inside each loop are the statements that calculate the expression and the statements for generating the result of the sequence expression, as shown in Listing 9.

Directly produce semantically equivalent results unmaintainable code. Ideally, the code of the generated executor does not need to be maintained, because it is lower level artefacts. The language developers should maintain the models and the code generators. However, generating clean and maintainable code is still desired. The generated code can be used for debugging the code generator. In addition, they may also be reused when developing another reference implementation by a non-MDE approach. Hence, another way is selected to generate OCL expression.

```
sequence
-> select x (expression1)
-> collect x (expression2)
-> forAll x (expression3)
```

Listing 7: Chained sequence expression.

```
sequence -> select x (createList(x)-> forAll y (f(y)))
```

Listing 8: Nested sequence expression.

```
sequence->select x (expression)
List local = new List();
for (Object x:sequence){
    TypeOfExpression e = expression;
    if (expression) local.add(x);
}
```

Listing 9: Generating code from sequence expressions.

The other way of implementing ALF sequence expressions is trying to interpret the ALF sequence expression in Java. Although mapping an ALF sequence expression in Java is not straightforward, it is very easy to map ALF sequence expressions to OCL, because the ALF sequence expression is designed to have the ability of OCL expressions. ALF sequence expressions differ only at the concrete syntax level; for example, the chained expression in Listing 10 can be expressed in OCL as

```
sequence
-> select (x|expression1)
-> collect (x|expression2)
-> forall (x|expression3)
```

Listing 10: Generating OCL from ALF sequence expressions.

Therefore, ALF sequence expressions can be transformed into OCL, and OCLs can be evaluated dynamically in Java. This is the approach that FQLS applied: An ALF sequence expression is transformed into an OCL expression, and the result of the OCL is dynamically evaluated as a query of the input model. For instance, returning to the Petri net example, the content of `Transition::isActive()` is a sequence expression. It is translated into an Emfatic operation with the content translated as OCL (see Listing 11). The content of dynamically evaluated OCL expressions in the input model is delegated to EMF code generation.

```
class Transition specializes Node{
    public isActive():Boolean{
        return this.inArcs->forall e (((Place)e.source).initialMarking>0);
    }
    translates to:
class Transition extends Node
{
    //@OCL(body="self.inArcs->
forall(e|e.source.oclAsType(Place).initialMarking>0)")
    op Boolean isActive ();
```

Listing 11: Sequence expression translates to OCL embedded in Emfatic.

It is worth mentioning that this approach is only a practical solution. It suffers two main limitations. The first is that the use of an ALF sequence expression has to be limited as the expression of a return statement, contained by an operation that contains only such a

return statement. The second is that the evaluation of the OCL expression is based on an EMF OCL project. Hence, its limitations are also shared.

Since an ALF sequence expression can easily be represented by project-lambda in Java, implementing sequence expressions in JDK8 is much easier than it is in JDK7. Hence, the OCL generation approach is only temporary, and could be improved by using JDK8 in future work.

Eclipse UI extension

The code generator also generates an Eclipse UI extension skeleton for executing the instance model. The language engineer needs to configure the Eclipse plugin project to add the action to create a new menu action. When the framework recognises an Ecore instance model as a DSL instance that has associated semantic definition, it will display an additional menu to execute the instance model. When the starting action is performed, it executes the classifier behaviour of the root of the instance model by default, and logs the execution steps.

Finally, by generating an implementation of the DSL specification, the DSL designer can test the implementation; should an error be found, the designer could modify the specification and regenerate the implementation. This makes identifying runtime and logic errors at an earlier stage possible.

7.4 Discussion

Having established the details of the execution layer, this section provides some discussions about it. The contribution of this chapter is highlighted by linking it to the complete FQLS. After that the discussion moves on to evaluation. Finally, how future advances of technology would change FQLS is discussed.

Contribution

This chapter presents the execution layer of FQLS, in which the ALF executor is implemented by code generation. Only the code generation itself does not form a novel contribution; however, the code generation is a way to add executability to language specifications via ALF, which is an undividable part of the FQLS.

By adding the execution layer, a language defined via ALF becomes testable. The testing performed on the specification will reveal more errors, and will increase the quality of the language specification eventually. For instance, going back to the Petri net example

presented in Chapter 5, the ASM version of the semantics specification contains a semantic error that wrongly calculates the tokens of places. Such an error will be revealed if the simplest test case is executed.

Evaluation

Considering the efficiency of the code generator, the speed of the generation and the resources it takes are reasonable. The small examples are generated to Java just like a normal EMF generation project, and in Chapter 9, the BPEL case study is evaluated, which contains more than 2000 lines of ALF code, generates thousands lines of code within seconds. The efficiency of the generated reference implementation is also reasonable. Small examples as well as more complex instance models are tested; the time between loading the instance model and giving out the final execution result is not delayed.

The efficiency of the code generation is largely dependent on the efficiency of EMF, which is demonstrated to be stable and efficient by many projects. On the other hand, the efficiency of code generation and execution is not an important factor for language specification development, because the main purpose of a language specification is to communicate to its shareholders. Creating an efficient implementation of the language is outside the development cycle of language specifications.

The correctness of the code generation is evaluated by checking the correctness of the language specification it produced. This is detailed in Chapter 9 after the complete BPEL case study is introduced.

Further changes

The FQLS approach reuses EMF in order to reuse the mature tools and simplify the creation of a reference implementation. On the other hand, reusing EMF also brings a challenge of covering the complete ALF language standard. The method of transforming ALF sequence expressions to OCLs has limitations which prevents the language engineers to use them freely. The reason is the lacking of similar concepts in Java. After the release of Java 8 and the newest EMF, ALF sequence expressions can be mapped to lambda expressions easily. This means that supporting them completely is only an implementation issue.

Another issue that may affect the design of the execution layer is the development of better ALF tools. The ALF tools are still in its early stage. In terms of executing ALF program, the only available choices¹⁸ are

- The ALF open source reference implementation.
- e-alf [91]
- IBM Rational Software Architect®, also support ALF syntax as an action language¹⁹.

They have been tested. None of the open sourced tools can be adapted to load an instance model to a complex ALF program successfully. The commercial tools appear to have a better editor but the mechanism of execution is not clear, thus it is impossible to integrate with it.

The lacking of available tools is one but not the only reason that FQLS chooses to generate Java code. Although not likely to happen in a short timescale, if a fully-functioned ALF executor is available, generating a Java implementation is still useful due to the requirement to separate specifications from implementations. Hence, if a good ALF executor is created, it is a further work to integrate this to FQLS in order to provide better integrated testing, but the code generator will remain as the starting point of an implementation.

7.5 Summary

This chapter discussed the components of the execution layer of FQLS. The code generator that generates Java from ALF code was introduced. The ALF code is first translated to Emfatic code by an Acceleo project. Mapping from the ALF concepts to Ecore concepts and Java concepts was listed. The code generation can also generate the UI extension of Eclipse, giving the model editor an extension menu to be executed.

¹⁸ Distributed Systems Engineering (<http://www.distributed-systems.de/>) presents a tool chain in CodeGen2014 that claims to support the editing and execution of ALF. However, the tool is still under internal evaluation and is not yet shown on their official website.

¹⁹ <https://www.ibm.com/developerworks/community/wikis/form/anonymous/api/wiki/b7da455c-5c51-4706-91c9-dcca9923c303/page/910cd642-c59a-47d7-9739-7459941a9e2b/attachment/9ba9af70-22fa-4756-8128-5ec8a3a08c2d/media/uml%20actional%20language%20in%20rsa.pdf>

Chapter 8.

Case study: Formalising Business Process Execution Language

The previous four chapters introduced each component of FQLS and illustrated how this framework could help to provide quality assurance for language specification development, using a Petri net example. In this chapter, the framework is demonstrated by applying it to a more complex and realistic DSL – the Business Process Execution Language (BPEL). This case study is used as a demonstration of FQLS, as well as a source for the evaluation of FQLS, which will be conducted in Chapter 9.

The case study of formalising BPEL consists of the following sections: Section 8.1 introduces BPEL and its related standards for web service composition. Section 8.2 introduces the scope of the case study. Thereafter, Section 8.3 defines the abstract syntax of BPEL by establishing mapping from the XML definition to a meta-model definition. Section 8.4 uses the development process to define the behavioural semantics of BPEL by creating a runtime meta-model, adding behaviours and performing quality assurance checks.

8.1 Introduction to WS-BPEL

In order to understand the case study that defines BPEL by FQLS, this section provides an introduction of BPEL and related standards. The three subsections introduces the definitions, the structures, and the life cycle of executing a BPEL process.

8.1.1 Compositing web services with BPEL

Web services are widely accepted as a communication method by many organisations. The ubiquitous applications of web services promote a set of technologies that aim to build web services in an easier and more standard way. The technologies promoted by worldwide standardisation organisations, such as W3C and OASIS, have gained wide support. They are chosen as the domain of this case study.

W3C [58] defines a web service as “a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL).” According to this definition, web services need interfaces to communicate with each other, and the interface is defined as a WSDL file.

WSDL is a standard for defining an interoperable interface for machine communication. A WSDL interface includes the input, output, and the fault message, which are independent of the implementation of the web service. A web service with a WSDL file describes its own features in a machine-readable manner.

Many features of the web are provided by web services. In business organisations, web services are used for communication and for achieving business goals. Because interoperability is one goal of web services, there are the requirements that integrate different web service systems and which composite these different web services to realise business logic that is more complex. This is the purpose of BPEL.

BPEL defines a model for describing the interaction between the business process and its partners, and these interactions are performed through WSDL interfaces. A BPEL process defines a set of primitive partner links, variables in the data process, handlers for errors and events and, most importantly, the executable activities that form a business process. The data process technology in the XML technology space, such as XPath and XSLT, are directly supported to be used inside a BPEL process.

Many servers support the deployment of a BPEL process, including as Apache ODE²⁰, GlassFish²¹, and ActiveVOS²². These projects support deploying a new BPEL process to the server, and deal efficiently with web requests. Specifically, they support long time transactions, which means that a process may wait for a response message days after it was first instantiated, and compensate for such long time transactions.

Although a BPEL process is expressed in XML, there is no official concrete syntax except XML. Since the XML syntax is verbose, Eclipse BPEL designer project²³, Netbeans SOA²⁴, and many other tools have provided similar unofficial graphical notation support. Simon et al. [141] also proposed a textual, human readable notation for

²⁰ <http://ode.apache.org/>

²¹ <https://glassfish.java.net/>

²² <http://www.activevos.com/>

²³ <http://www.eclipse.org/bpel/>

²⁴ <https://soa.netbeans.org/>

representing BPEL. In this thesis, the graphical syntax of Eclipse BPEL is used to demonstrate BPEL process examples.

8.1.2 Structure of a BPEL process

An example of a BPEL process is listed in Figure 37, represented by the Eclipse BPEL designer graphical syntax. The graphical syntax defines the activities and fault handlers. However, a complete BPEL process contains four parts, namely partner links, variables, handlers, and activity.

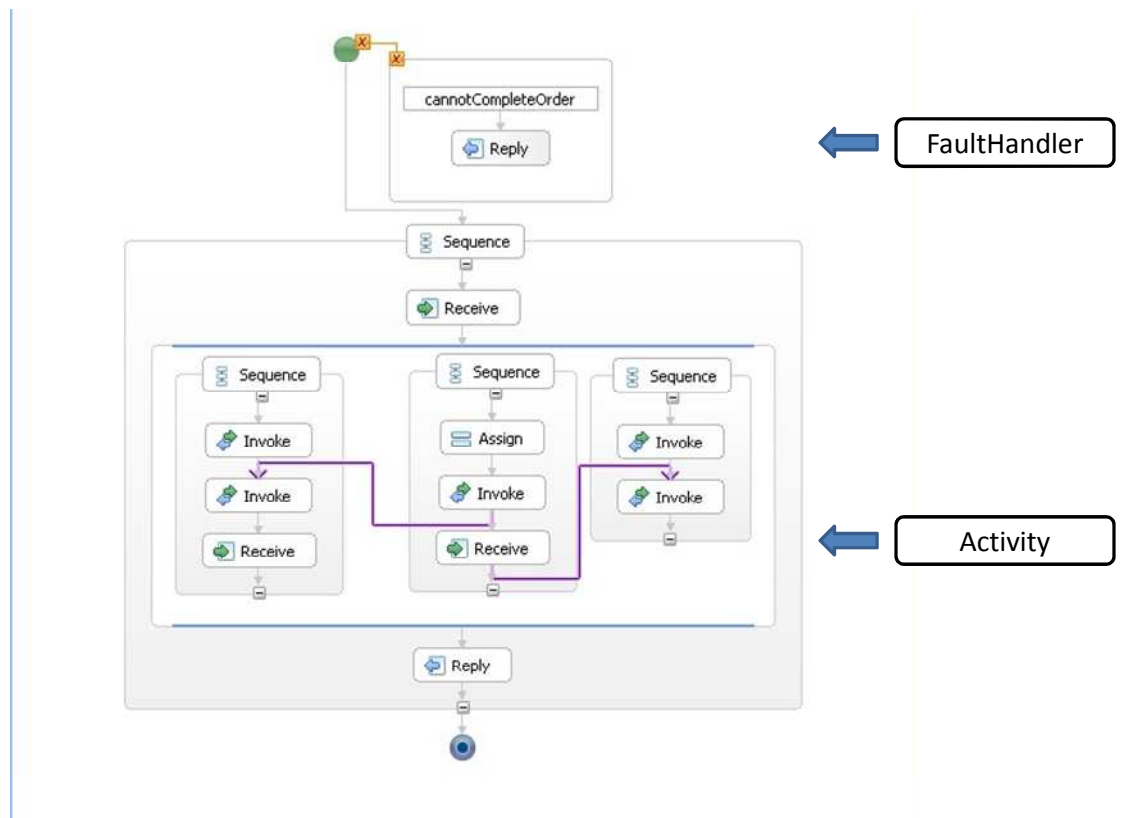


Figure 37: Graphical syntax of BPEL.

The first block of the XML file defines the partner links as a reference to a particular partner link type that is defined in a WSDL file. A port type is similar to an interface type, in that it contains operations. Similar to the operations in programming languages, they have input parameters and output parameters, and can produce a fault message. The type of parameter is defined as a WSDL message, which encapsulates the XML data type of the message. The WSDL file then defines the partner link types, which include the role name of the service. By linking a port type to a role, it restricts the operations that a role can

invoke. Finally the WSDL file contains the physical settings of the web service, including the physical web address of the web service.

Figure 38 illustrates the relationship between BPEL and WSDL. A partner link is defined by declaring its partner link type and its role. Both of these are defined in the WSDL file. The partner link type then refers to a port type definition. The port type defines a set of operations that contains parameters, and these operations are interfaces that can be invoked as a web service. The web service interfaces defined in a WSDL file are platform independent from its physical URL and its implementation technology, thus enabling the change of the implementation or of the physical address without changing the interfaces.

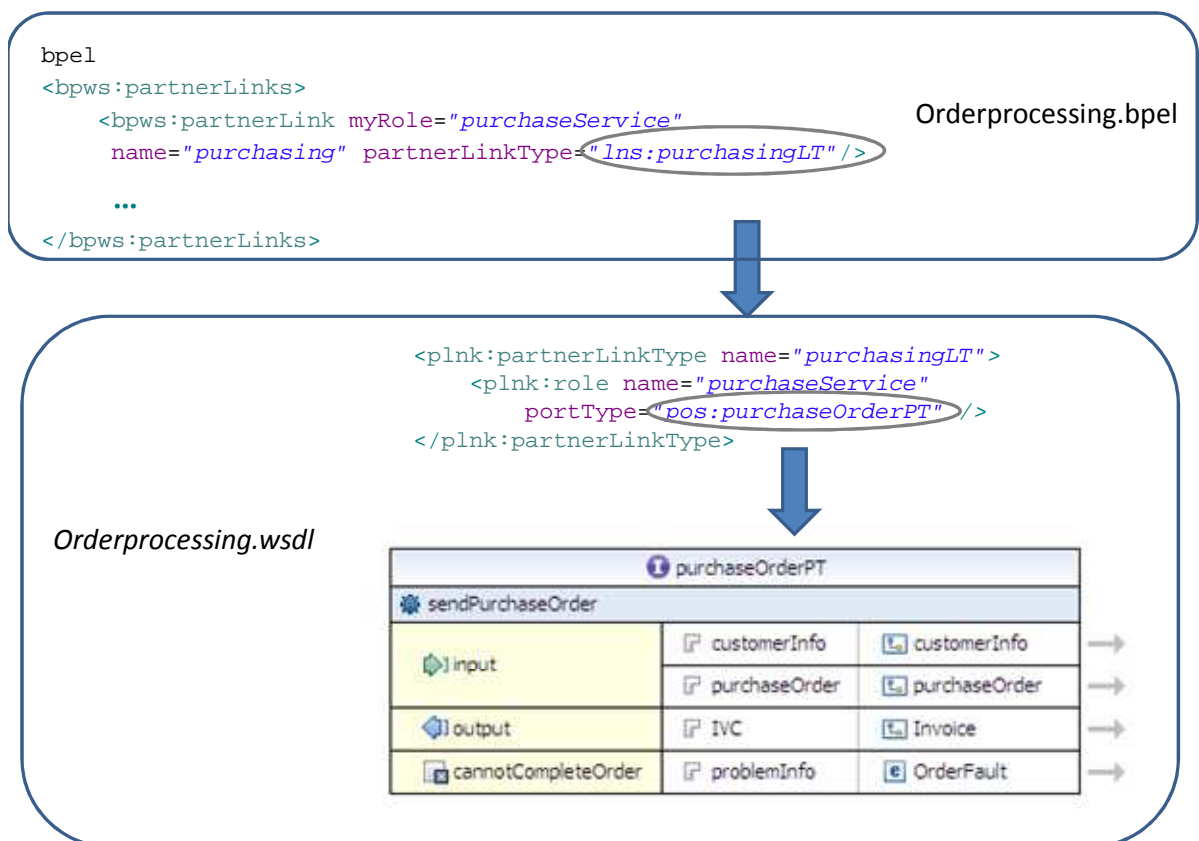


Figure 38: BPEL and WSDL.

The next element defines the variables. The variables define the spaces for holding the data that need to be processed in the business process. As with the concept of variables in GPLs, the execution engine can read and update the value of the variable. The variables are typed, and the type must be an XML type or a WSDL message type. For example, the following line defines a variable, the name of which is Invoice, and the type is defined as an XSD type `lms:InvMessage`, which is defined in a schema file.

```
<bpws:variable messageType="lns:InvMessage" name="Invoice"/>
```

Variables can be manipulated using an XML query language, using XPath by default, and are mainly used in `Assign` activity. When invoking an external web service or being invoked by other web services, the value input and output parameters are stored in variables.

Handlers are started when their triggers have been initialised; thereafter the activity defined in the handlers is started and the normal execution may be terminated. The handlers can be defined inside a process or a scope activity (introduced in Table 9).

Handlers	Introduction
Fault handler	A fault handler will be started when a fault is thrown. The normal execution terminates thereafter. It defines a fault handling activity that handles the errors.
Event handler	An event handler is triggered by events, including time events (after a certain time elapse or a certain time is reached) or message events (when the process receives certain messages).
Compensation handler	A compensation handler is started when the process wishes to undo all the operations it has already executed. This is usually triggered by a "compensate" activity.

Table 9: BPEL handlers

The next element consists of activities. Table 10 and Table 11 give a list of the activities of BPEL. These activities are categorised as basic activities and structural activities. Basic activities enable the exchange of messages to external web services, provide data process actions such as assigning values to variables, and special activities for error handling, such as throwing a fault message. Structured activities are containers of other activities. When a structured activity starts to execute, it starts its child activities by its semantics. The example defines a process that receives messages from the external web services, using `Flow` activity to invoke external web services and waiting for responding messages in parallel.

Basic activity	Introduction
Empty	An Empty activity does nothing, as the name suggests. It can be used as a placeholder, or can be used as an activity to join flow links.
Assign	An Assign activity defines a specification for manipulating the data. It copies data from one part to another, where the parts can be a variable, an expression or a literal value.
Receive	A Receive activity receives a message from its specified partner link and assigns the received message content to a variable.
Reply	A Reply activity sends a response to a previously received two-way (request-response) request.
Invoke	An Invoke activity invokes an operation of a partner link with input message. Depending on the type of output, it can also receive a response message from the partner link.
Terminate	Terminates the process instance.
Throw	A Throw activity throws a fault that indicates an internal error. The fault is then dealt with by a fault handler.
Waiting	A Waiting activity causes the process instance to wait for a period, or to wait until a certain time point.
Compensate	A Compensate activity can only appear in a fault handler, which will start the compensate handlers according to the sequence of execution, and will thus undo the work being done by the instance.

Table 10: BPEL basic activities.

Structured activity	Introduction
Sequence	A Sequence activity executes its contained activities sequentially; in other words, the completion of the previous activity triggers the start of the next activity.
Flow	A Flow activity executes its contained activities in parallel with regard to the flow link definition. Flow links and dead path elimination are introduced in Subsection 8.4.5.
Switch	A Switch activity defines a logic branch. The first branch in which the condition expression evaluates to true is executed.
While	A While activity repeatedly executes its containing activity until its condition expression evaluates as false.
Pick	A Pick activity is similar to a Receive activity. The difference is that it can receive different types of messages, and can define different enclosed activities for each type of message.
Scope	A Scope is similar to a sub-process, as it can also define handlers, variables and partner links, but the variables and partner links only work within its scope.

Table 11: BPEL structured activities.

8.1.3 Execution of a BPEL process

A BPEL process is defined as a “*stateless client-server model of request-response or uncorrelated one-way interactions*” [147]. The process is the guidance and instruction for the BPEL execution engine in terms of managing the instances of the BPEL process. A process instance starts when an activity receives a new message. The activity must be either receive activity or pick activity with its “createInstance” attribute set to ‘yes’. Such an activity is also called a start activity, and no other activity can be placed before it.

After the instance is started by the start activity, the execution engine of BPEL needs to initialise the process instance. If a process has correlation sets, they will first be initialised (see Subsection 8.4.3 Communication and Subsection 8.4.7 Correlations for the detailed semantics of correlations). If these activities contain any handlers, the handlers will then be installed before executing the activities.

The process instance then executes the enclosed activity of the process. The example contains a sequence activity, and the activities it contains will be executed sequentially.

After receiving a message via the receive activity, the flow activity begins and starts all its containing activities in parallel.

A business process instance finishes either normally or abnormally. When the main activity of the process and other handlers finish without propagating faults, the process instance ends normally. On the other hand, if any fault occurs, the relevant fault handler will be started and, depending on the activities in the fault handler, the process may be compensated to undo the work it has done. In such a situation, the process ends abnormally.

8.2 Scope of the case study

In this section, the scope and the architecture of the case study are defined. Since BPEL is a complex language that have two major versions, related to other web service languages and can be extended, it is necessary to limit the scope of the case study, and clearly define what would be included and what will not.

Firstly, it is needed to answer what should be included. The case study is a formalisation of an existing DSL. Unlike creating a DSL from scratch, the requirement of the DSL is clear: It is the BPEL standard [147]. The BPEL standard defines its abstract syntax as an XML schema, while defining the execution semantics as text. Both the abstract syntax and execution semantics are illustrated by examples. The following paragraphs discuss each aspects of the standard and explain the decisions have been made.

Abstract syntax

Rebuilding the language specification of BPEL by FQLS starts with building the abstract syntax. The mapping from XML schema and ALF structures is straightforward. An XML element is mapped to a MOF class, with attributes mapped to an attribute that belongs to the same MOF class. The containing and referencing relationships of XML are also similar to those of MOF. Such a translation has already been developed by the BPEL designer project, which included an Ecore model of BPEL.

Static semantics

Another concern is the static semantics of BPEL. This involves which version of BPEL to implement, since there are certain differences between BPEL 1.1 and BPEL 2.0. The BPEL 1.1 standard does not consider the additional well formedness of the meta-model; or at least it does not define it explicitly. BPEL 2.0 has defined the well-formedness rules in a clearer and more explicit way as text.

There are existing works that attempt to formalise the static semantics, such as OCL expression [3]. FQLS supports one way of extending ALF with OCL expression definitions, and the same way of defining well formedness can be applied. Because completely re-implementing static semantics is outside the scope, the static semantics are excluded.

BPEL 1.1 and 2.0

While BPEL1.1 and 2.0 shares many similar concepts, there are several differences. In the process life cycle, the execution workflow does not change, but some new activities are added and some activities are renamed more sensibly.

BPEL 1.1	BPEL 2.0
switch	if .. then .. else
terminate	exit

Table 12: Differences between BPEL 1.1 and 2.0.

Added in BPEL 2.0	
Compensate scope	Similar to Compensate, but only compensates for the specified scope, rather than the complete process.
rethrow	Similar to the Throw activity, which can be used in a fault handler.
repeat until	Similar to the While activity, which evaluates the condition after executing its enclosed activity.
foreach	Similar to the While activity, of which the ending condition is to execute the enclosed activity a certain number of times.

Table 13: Added concepts in BPEL 2.0.

The activities that appear in both 1.1 and 2.0 are supported. The relevant research in BPEL 1.1 is more complete and mature than that of 2.0. For BPEL 1.1, Stahl [145] and Fahland and Reisig [37, 38] are identified, which tries to use various methods of semantic definition, but 2.0-based approaches are not identified. Hence, using BPEL 1.1 as the base will enable this case study to be compared to many other approaches. The comparison with other semantic definition approaches of BPEL 1.1 is one of the experiments performed in Chapter 9.

In addition, the semantics of the *repeat until* and the *foreach* activities can easily be represented by a *while* activity with a changed condition. The same also happens to the

embedded fault handler in an invoke activity, which is equivalent to a scope activity that contains an invoke activity and a fault handler.

Semantics of abstract processes and extensions

There are two kinds of the BPEL process, the normal BPEL process and the abstract BPEL process. The difference is that an abstract BPEL process can hide certain details, such as the detailed specification of partner links. As a result, an abstract BPEL process is not meant to be executable, while an executable BPEL process is ready to be deployed to a server as a new specification of web services. The aim of the abstract process is to explain the process. Due to the lack of relevant information, the process is not executable; thus, defining the execution semantics is meaningless. Hence, in this thesis, the focus is only on the executable BPEL process.

The extension mechanism can extend the BPEL process with new concepts. Execution engine developers can use this mechanism to provide advanced activities that are not officially supported by the standard. The behavioural semantics of the extensions are outside of our scope for formalising the BPEL standard. Hence, it is omitted.

Semantics of related standards

BPEL is not a standalone language; many places contain the composition of another language. XPath, WSDL and XML schema are commonly used languages that are related to BPEL. BPEL uses XPath for querying data and managing the incoming/outgoing messages, and changing the value of the variables. The BPEL partner link definition needs to refer to WSDL files so that the BPEL execution engine can query information, such as parameters and message types about external web services. An XML schema is used for defining the message types. While BPEL is linked with several other DSLs, defining the other DSLs are clearly outside of the scope of the BPEL definition. Hence, the abstract syntax of the other DSLs are simplified. A simplified meta-model of WSDL is built and imported to a BPEL meta-model, and an XPath expression is treated as a black box, the semantics of which are defined as abstract activities.

By defining the scope of the case study, it is possible to create a meta-model of BPEL via ALF. Hence, it is possible to begin to develop the abstract syntax.

8.3 Defining abstract syntax

The language specification of BPEL starts from this section. Following the software development process proposed in Chapter 4, the abstract syntax is defined as a meta-model. Since WSDL is an indispensable part of BPEL, a basic meta-model of WSDL is firstly presented, following by the BPEL meta-model.

Figure 39 illustrates the meta-model of the WSDL, which is used to define external web services. In reality, port types, service link types and messages can be defined in different XML files, and the types can be types in XML Schema name spaces.

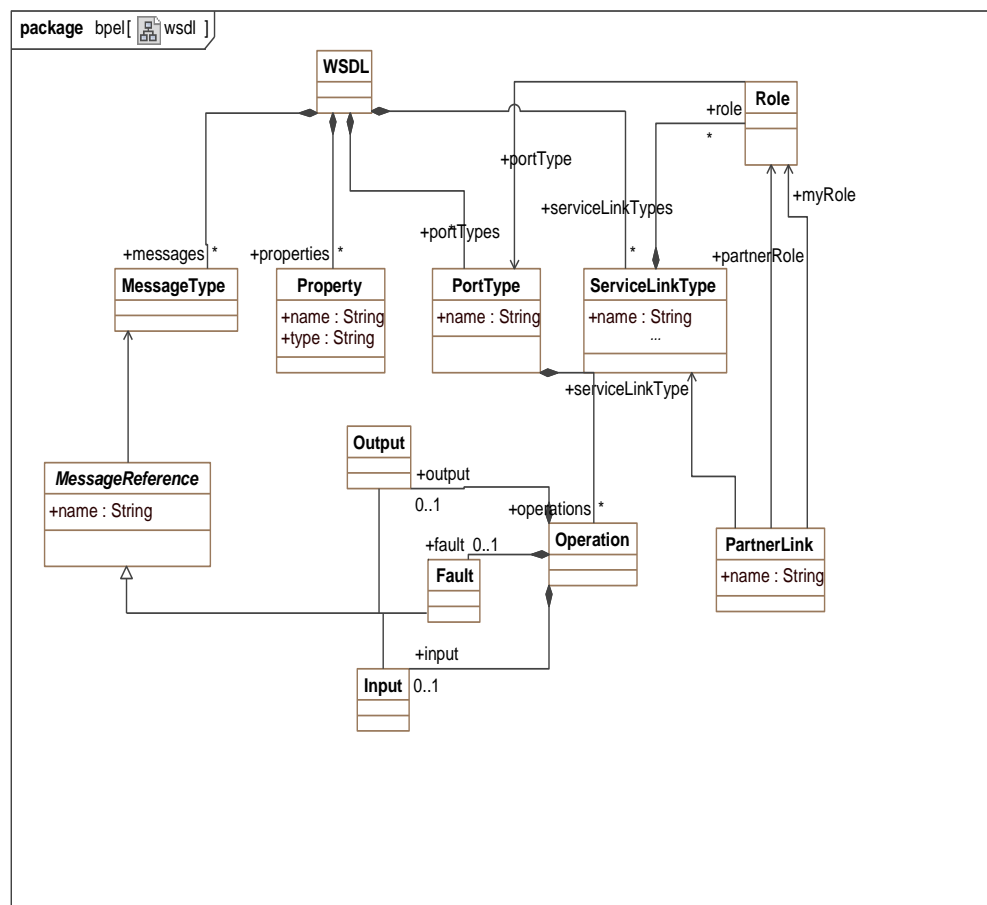


Figure 39: Abstract syntax of WSDL.

The `PortType` class comprises the `operation` list, and is a type of external web service. The operations define the interfaces for communication between web services, including the input, output and fault parameters, and the types of the parameters are defined as `MessageTypes`.

`Property` is a special kind of message that is similar to the ‘derived attributes’ in an MOF-based meta-model. It uses query language to define a particular manipulation of

message contents. Properties are used in BPEL as a way of generating correlation instances.

The `ServiceLinkType` encapsulates an external web service and defines all the parties that are involved in an interaction with the web service. The partners are defined as roles. For example, a web service that receives a request and provides information with regard to that request, two roles are involved, namely the web service that provides the service, which may be given a role name like 'serviceProvider', and the partner that invokes the service, which can be given a role name like "client". The `PartnerLink` class is the same class in Figure 40 and it bridges the two meta-models.

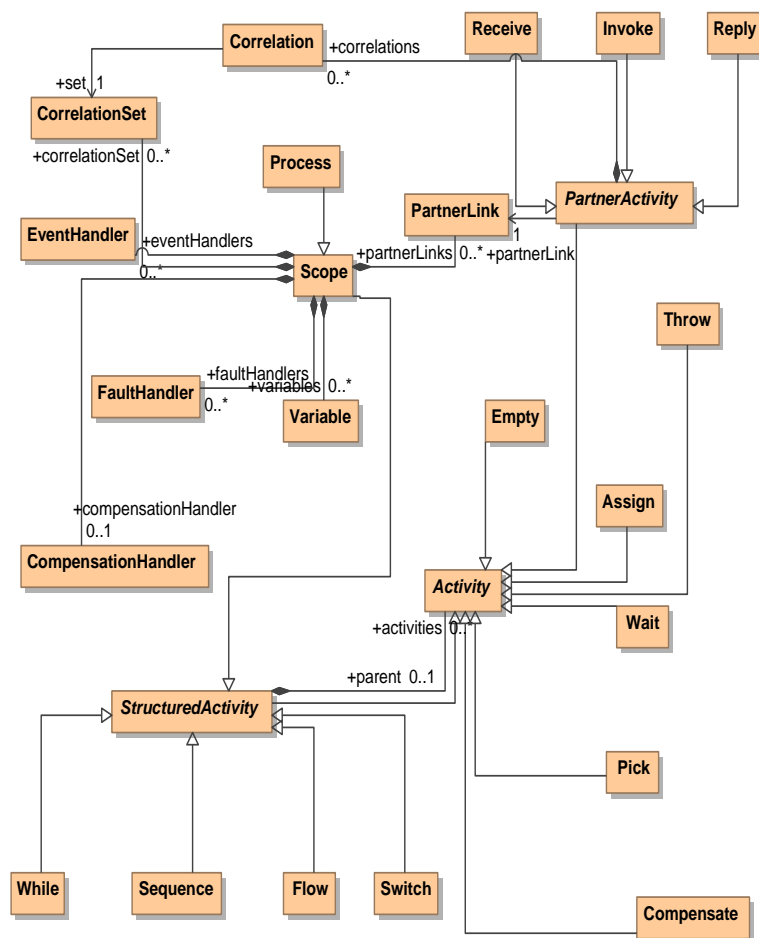


Figure 40: Abstract syntax of BPEL.

Figure 40 shows the meta-model of the BPEL abstract syntax. The meta-model proposed by Eclipse BPEL designer project is reused, but is simplified by removing irrelevant concepts, such as the full definition of WSDL, XPath and XSLT. The meta-model has a similar structure to that of the XML schema defined in the standard. The `Scope` is the basic unit that acts as a container of handlers, variables, partner links and

activities. The `Process`, which represents a BPEL process, is a sub-class of `Scope`. A `Scope` also inherits from a `StructuredActivity` and an `Activity`.

The activities listed in Table 10 and Table 11 are defined as sub-classes of an `Activity`. When they share similar behaviours, they are defined as being inherited from the same class. For example, the activities of `Receive`, `Invoke` and `Reply` need to communicate with other partner links, thus sharing behaviours. These are categorised as `PartnerActivity`.

Finally, it is possible to build the meta-model via ALF. Readers who are interested can find the ALF definition of the meta-model in the Appendix.

8.4 Defining behavioural semantics

The next step in formalising BPEL is to develop its behavioural semantics. Since the requirement is derived from the standard, a strategy for defining semantics needs to be chosen, designing the architecture of semantics, identifying abstract activities and, finally, developing the behaviours. This section begins with building a runtime meta-model in subsection 8.4.1. The behavioural semantic definition of BPEL is presented from Subsection 8.4.2 to Subsection 8.4.8.

8.4.1 Execution model overview

As indicated in Subsection 5.1.3, there are three ways of attaching behaviours to the meta-model, namely adding operations directly, adding behaviours as activities, and adding operations to a runtime meta-model. The semantics of BPEL require adding instance level concepts because, while executing a BPEL process, the execution engine creates new instances of the process and manages these instances. Thus, it is necessary to create a runtime meta-model and add the behaviours to it.

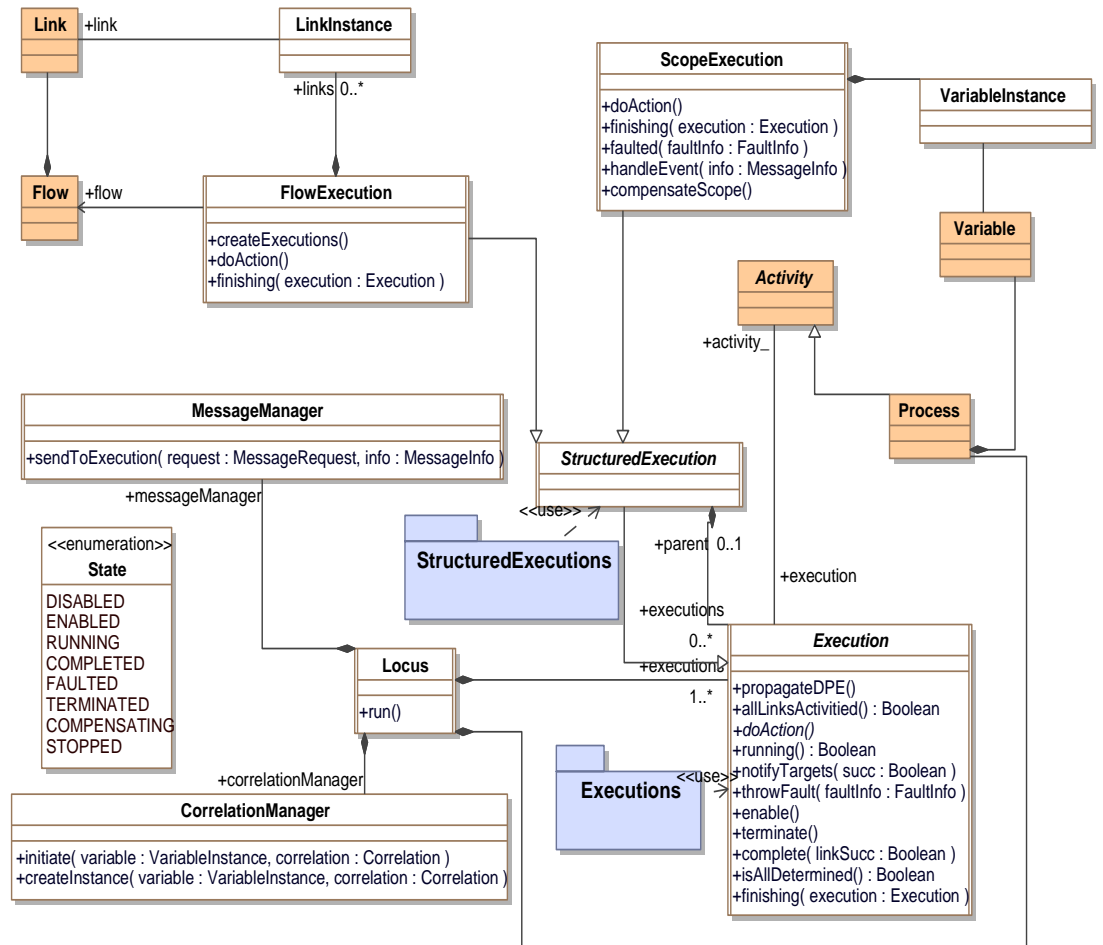


Figure 41: Runtime meta-model of BPEL.

Figure 41 lists the class diagram of the runtime meta-model. The process level concepts, such as variables, activities and correlations, all have corresponding instance level concepts. For example, for each subtype of `Activity`, an `Execution` class is created to represent the instance level model (`FlowExecution` corresponding to `Flow` activity).

To express the execution semantics of BPEL, adding the corresponding instance class to the meta-model is not enough. The BPEL specification defines the architecture for managing process instances and ways of passing and correlating messages between activity instances. Thus, it is natural to create the class of `Locus`, which is a virtual machine for managing instances. It provides the entry point for the semantic specification. As an active class, the class starts when its instance is created.

As described above, a BPEL process instance communicates with its external web services by exchanging WSDL messages. The class `MessageManager` manages the incoming and outgoing messages. When incoming messages are received, the `MessageManager` must dispatch the message to the correct instance. When there are many

instances waiting for the same type of messages, the `MessageManager` identifies the correct target instance by calculating a unique identifier from the incoming message, which is called correlations.

A `PartnerActivity` (the class diagram is illustrated in Figure 39 and Figure 40) can define correlations. When the business process is instantiated, the correlation instance can be calculated by operating the incoming message. When further messages are received, the execution engine dispatches the message to the instances that have the same correlation instance. In Figure 41, the `CorrelationManager` class defines the behaviours related to the correlations.

8.4.2 Variables

The basic data structure with which a BPEL process can interact is messages. The execution of a BPEL process involves exchanging messages between partner links. Thus, it is necessary to provide a method for holding these messages for exchanging or manipulating, and these are the variables. A BPEL variable has a name that identifies it, and a message type that references a message defined in the WSDL documents. At the abstract syntax level, as which other programming languages, the variable is the definition of a data structure which needs to be instantiated as an object for storing message instances.

When executing a process, a process instance creates `VariableInstances`, which reference the original variable definition, and which is given memory space for storing the message content. `VariableInstances` is then used as the atomic data that is exchanged between different BPEL activity instances.

8.4.3 Communication

In the BPEL execution model, two kinds of communication are needed. The first is external communication. This process needs to communicate with external web services. For example, a BPEL instance can receive external messages and can reply to external web services. The communication is done via WSDL messages, and the WSDL message is text. In this case study, the `MessageManager` class is in charge of external communication.

Communication with external web services involves two basic behaviours, namely sending a message to an external web service, and receiving a message from an external server. While sending to externals is quite straightforward, by invoking the remote operation calls defined in the WSDL web service, receiving a message can lead to two different behaviours. This could result in the message being sent to an execution that is

waiting for this particular message, or could result in a new business process instance being created.

When an execution that requests an external message, the execution informs the `MessageManager` regarding the message it is requesting. `MessageManger` saves the information, and when it receives a message, it compares the current message requests, and tries to map the message to the correct instance. Default message mapping is via message type and, if ambiguity happens, the correlations will be used.

There is an unclear definition of who is in charge of creating new instances. The BPEL standard specifies that a starting activity creates a new instance when receiving a new message. Referring to the standard [147],

*“The creation of a process instance in BPEL4WS is always implicit; activities that receive messages (that is, `receiveactivities` and `pickactivities`) can be annotated to indicate that the occurrence of that activity causes a new instance of the business process to be created. This is done by setting the `createInstanceattribute` of such an activity to “yes”. **When a message is received by such an activity, an instance of the business process is created if it does not already exist.**”*

According to the standard, an instance will be created when a receive activity receives a message. However, such an instance of the receive activity does not exist before it can receive a message. Fahland [38] also discussed this, and assumed that an instance is created before the receive activity receives any messages. In this thesis, a similar approach is applied: When the BPEL process is deployed (when the `Locus` class is created and started), the `Locus` immediately creates a BPEL process instance that waits for the message for the start activity. The created instance requests the message by sending `SignalRequest` to `MessageManager`. When the start activity receives a message, it tells the `Locus` to create another BPEL instance that waits for further activity.

In addition to external communication, internal communication between executions are also needed. In our execution model, each instance of the activity (executions) runs as an independent thread. These need to communicate to relevant executions; for instance, when an execution finishes, it needs to inform its container that the execution is completed. Another example is when a fault occurs. The `ScopeExecution` needs to be informed, so that the `ScopeExecution` will start the fault handler. Internal communication is performed using signals and signal receptions.

8.4.4 Semantics of Basic Execution

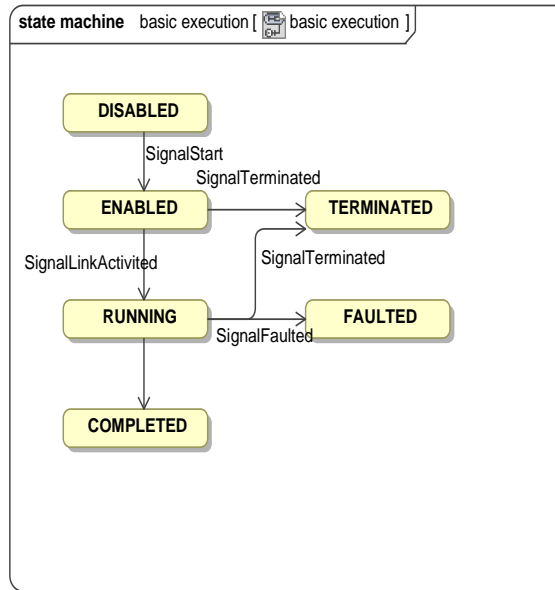


Figure 42: State machine of basic execution.

The `Execution` class is an instance of an activity. It associates the original activity in the abstract syntax and provides operations for executions. The process of executing a BPEL process is to create top-level execution (the process execution, which is a sub-class of `ScopeExecution`). The activated execution creates the contained executions, and each of the executions is executed in parallel before being stopped.

Figure 42 illustrates an abstraction of the internal states of an execution. Because the executions are instantiated as a referenced tree, this means that each execution is contained by a structured execution. The structured execution is responsible for instantiating its contained executions; for example, a sequence execution needs to create executions sequentially. Hence, the structured execution manages the executions it contains, which is called the parent execution. The contained execution is called the child execution.

When an execution is created, its default state is `DISABLED`. Its parent execution then sends a `SignalStart` signal and the state transits to `ENABLED`. Then, depending on whether the child execution has flow links, it directly transits to `RUNNING`, or could wait for the flow links. Depending on the result of the execution, it could be ended as `TERMINATED`, `FAULTED` if it exists abnormally or `COMPLETED`, if it exists normally.

When the `Execution` transfers from `ENABLED` to `RUNNING`. The execution waits for a `SignalLinkActivated` signal. This is because, when an execution starts, it may not transit to `RUNNING`, because it is contained by a `Flow`, and there are previous links.

Consider a `Switch` or `Pick` activity that is contained by a `Flow` activity. When the flow execution executing all its contained executions enters the `ENABLED` state, the `Flow` execution will be complete if all of its child executions complete a normal execution flow. However, not all of them will be executed. For instance, in Figure 43, only one of the execution paths will be successfully executed, and the other executions in other paths should be ignored. The behaviour for checking whether an execution contained by a `FlowExecution` should be executed or ignored is called the *Dead Path Elimination* (DPE). When such an execution finishes, it needs to inform all the executions that have a link as a target whether the execution is successful or not.

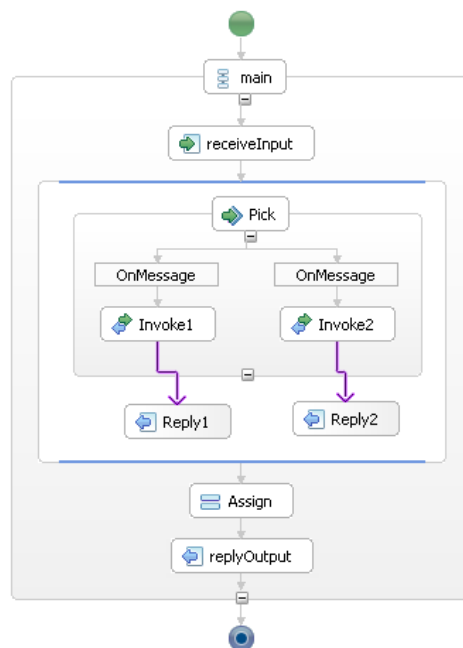


Figure 43: A flow activity that contains pick activity and links.

The works of `propagateDPE()` and `notifyTargets()` are defined as operations in the `Execution` class, and activity specification behaviours are defined as the sub classes of `Execution`.

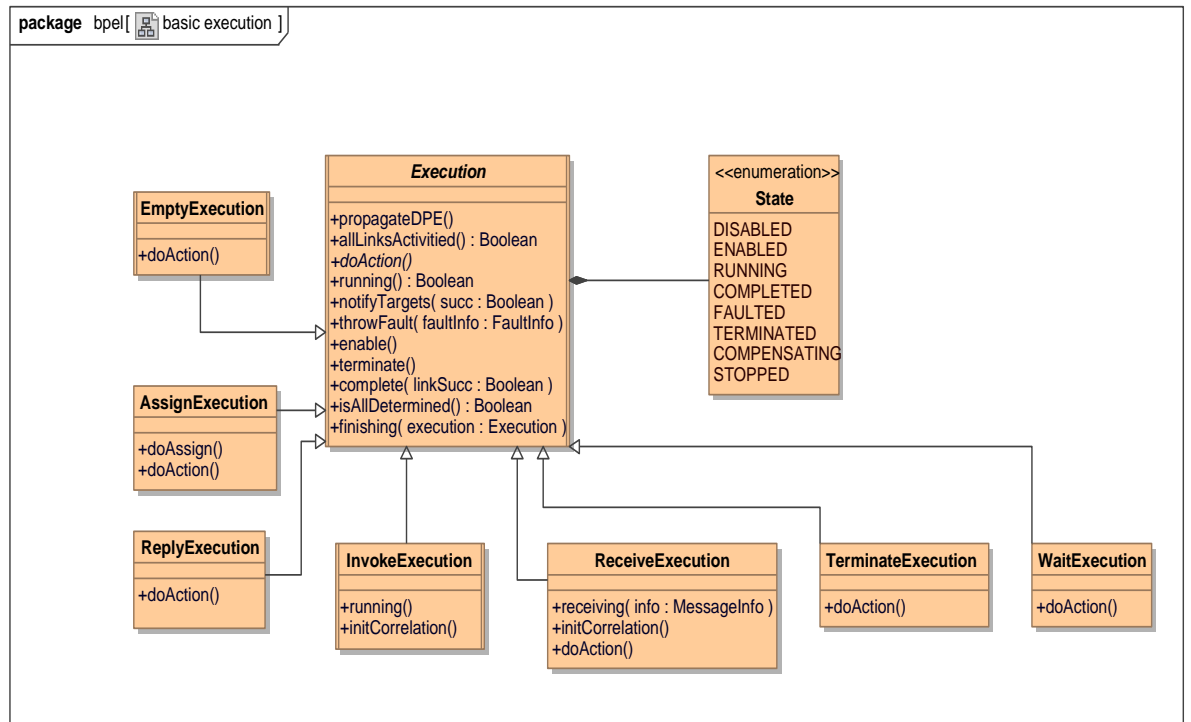


Figure 44: Basic executions from runtime meta-models.

A pattern is used for realising the state machine specified in Figure 44. Operations that have the same name as a state transform the execution's state as the name suggests. For instance, `enable()` involves the necessary behaviours to transform the state to `ENABLED`, while `running()` transforms its state to `RUNNING`, as does `terminate()`. All executions except `ScopeExecution` share the same semantics for enabling and terminating; thus, they can be reused.

The operation sets its state to `ENABLE` and, when its activity does not have a source link that indicates that the execution is not needed when waiting for a link to activate the execution) it will automatically transmit to running by sending `SignalLinkActivited` to itself.

```

public enable(){
    if (this.state==State.DISABLED){
        this.setState(State.ENABLED);
        if (this.activity_.sources->isEmpty()){
            this.SignalLinkActivited();
        }
    }
}
public terminate(){
    this.setState(State.TERMINATED);
}

```

The operation contains various behaviours of different activities. Obviously, the contents of `running()` will depend on the activity. However, the semantics of dead path elimination are performed before the actual behaviours. Thus, the `running()` operation contains the behaviours of dead path elimination. The actual behaviour of running an execution is placed in the `doAction()` operation.

In Listing 12, the class of `EmptyExecution` is presented, which demonstrates the pattern that is used by all executions. The statements in the classifier behaviour defines a loop that will receive signals until a signal that causes the loop ends, at which point the active object will end. Because `EmptyExecution` does nothing, it sets the state to `RUNNING` and then completes the execution. This hides the mechanism such as performing the dead path elimination, and forces to put the activity-specific behaviours to the `doAction()` method. The other basic executions listed in Table 10 have similar "do blocks". Their specific behavioural semantics can be represented as ALF code. Readers can refer to the Appendix A for details.

```

public active class EmptyExecution specializes Execution {

    public doAction(){
        this.setState(State.RUNNING);
        this.complete(true);
    }

}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (! completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivited){
            this.running();
            completed = true;
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}

```

Listing 12: EmptyExecution.

8.4.5 Semantics of Structured activities

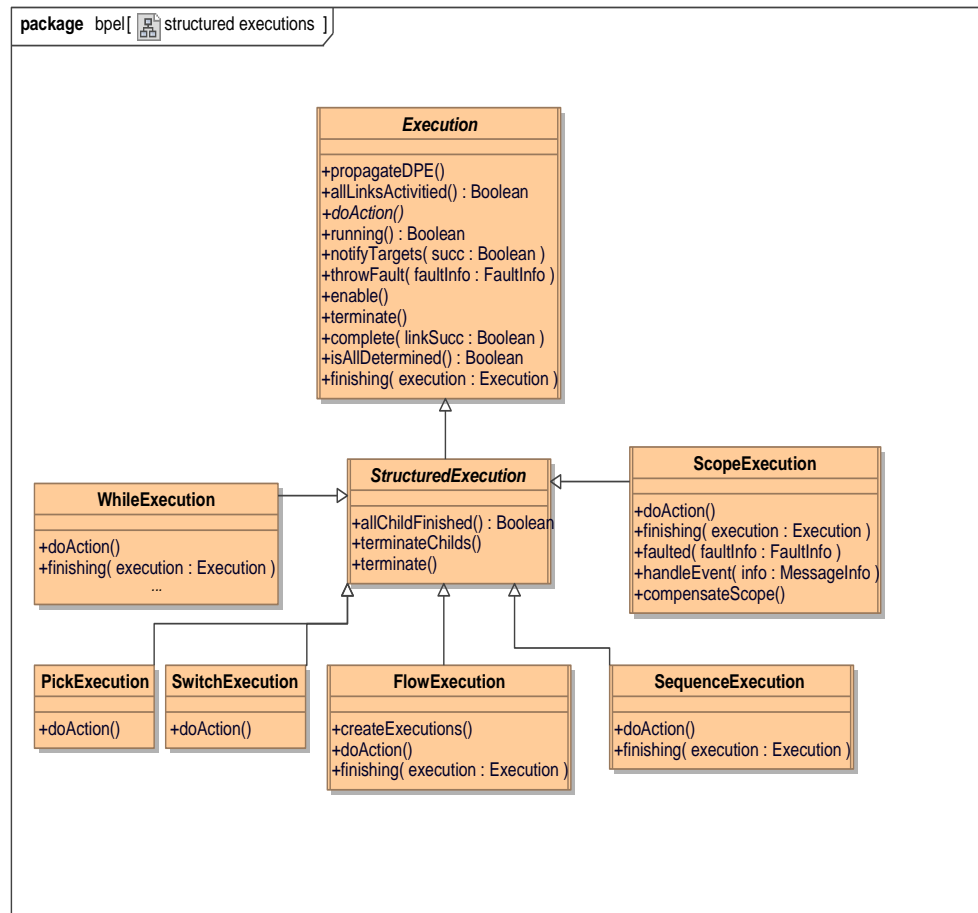


Figure 45: Structured executions.

Structured activities are different from basic activities because they deal with their child activities. A basic activity is completed when its execution ends. A structured activity needs its child activities to send `SignalChildFinished` to it.

Listing 13 shows an example of the pattern of the classifier behaviour of a structured activity. When receiving a `SignalChildFinished` signal, the active class decides whether to wait for further child activity to be finished, or completes the activity.

```

    }do{
        this.setState(State.DISABLED);
        let completed:Boolean = false;
        while (!completed){
            accept(SignalStart){
                this.enable();
            }
            or accept(SignalLinkActivited){
                completed = this.running();
            }
            or accept(sig:SignalChildFinished){
                completed = this.finishing(sig.execution);
            }
            or accept(SignalTerminate){
                this.terminate();
                completed = true;
            }
        }
    }
}

```

Listing 13: Classifier behaviour of structured activity.

The list below shows a `FlowExecution::doAction()` operation. The last statement sends signals concurrently to all `childExecution`. This enables the semantics for executing all activities in a `Flow` activity in parallel.

```

public doAction(){
    this.setState(State.RUNNING);
    this.createExecutions();
    this.childExecution.SignalStart();
}

```

Listing 14 shows the ALF programme for dealing with `SignalChildFinished`, depending on whether the child execution is finished normally or negatively. If the child execution finishes negatively, the `FlowExecution` terminates all activated children, sets its state to `FAULTED` and informs its parent. If all the children are finished, the `FlowExecution` also finishes normally.

```

public finishing(in execution:Execution):Boolean{
    let completed:Boolean = false;
    if (execution.state==State.FAULTED){
        this.setState(State.FAULTED);
        //inform parent execution
        this.terminateChilds();
        this.parentExecution.SignalChildFinished(this);
        return true;
    }

    if (this.allChildFinished() && this.state==State.RUNNING){
        completed = true;
        this.complete(true);
    }
    return completed;
}

```

Listing 14: Finishing operation.

8.4.6 Semantics of Scope and Process

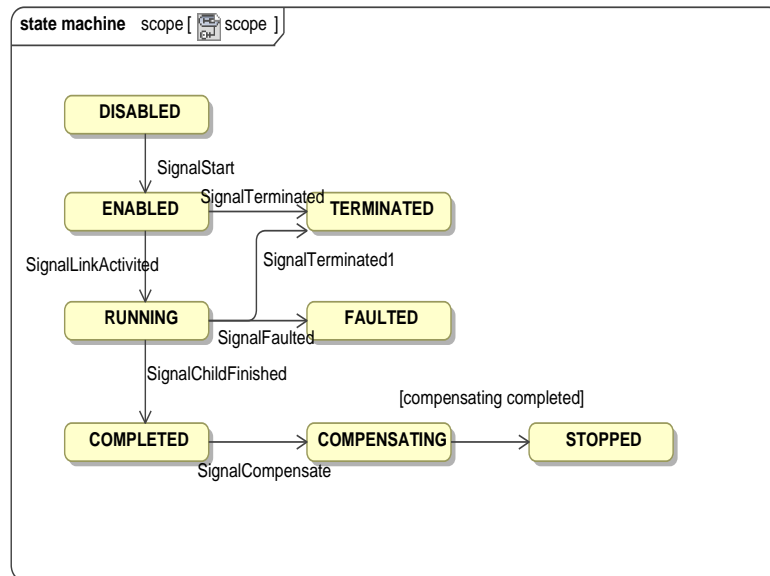


Figure 46: State machine of ScopeExecution.

A BPEL process is a special kind of Scope. A Scope can also define variables and partner links, but such variables or partners are only available within the Scope. With regard to the instance of a Scope, the ScopeExecution class is responsible for error handling, event handling and compensation handling.

A fault can be thrown by a `Throw` activity when an external partner service returns a fault message or when runtime errors that are specified in the standard occur. The fault is passed to its enclosed `ScopeExecution` and, if the enclosed `ScopeExecution` does not define a fault handler, it will be passed to the higher-level enclosed scope until the process instance. If a fault handler does not exist, the BPEL instance should create a default fault handler for that fault.

Event handlers define special kinds of activities. Their enclosed activity will start when it is triggered by an event. A normal `StructuredExecution`, such as the `FlowExecution` or `SequenceExecution`, when child execution starts, will only wait for termination signals. When starting a `ScopeExecution`, it requests messages that trigger an event handler from the `MessageManager`. When such messages are sent to the scope, the `ScopeExecution` manages the creation of the execution of the event handler activity.

Apart from other structured activities, when the child activity finishes and the scope transits to the `COMPLETED` state, the active object does not quit and is not destroyed, but still waits for compensation. This is because a BPEL process is a long-term transaction and may take a long time to find that something is wrong, while the executions that have already been executed need to be cancelled in the compensation handler activity. In order to express a mechanism of compensation handling, a process instance has a `CompensationStack`. When a `ScopeExecution` transits to the `COMPLETED` state, it also registers itself to the compensation stack. When a `CompensateExecution` activates, it will send signals sequentially to all the `ScopeExecutions` in the compensation stack, and the compensating handler activity of the `Scope` will be instantiated to the process of any compensating work.

```

    }do{
        this.setState(State.DISABLED);
        let completed:Boolean = false;
        while (!completed){
            accept(SignalStart){
                this.enable();
            }
            or accept(SignalLinkActivated){
                this.running();
            }
            or accept(sigFinish:SignalChildFinished){
                completed = this.childFinish(sigFinish.execution);
            }
            or accept(sigReceive:SignalReceive){
                this.handleEvent(sigReceive.info);
            }
            or accept(sigFault:SignalFaulted){
                this.faulted(sigFault.faultInfo);
            }
            or accept(sigCompensate:SignalCompensate){
                this.compensateExecution =
sigCompensate.compensateExecution;
                this.compensateScope();
            }
            or accept(SignalTerminate){
                this.terminate();
                completed = true;
            }
            or accept(SignalCompleted){
                completed = true;
            }
            // accept
        }//while
    }

```

8.4.7 Correlations

The `MessageMangaer` is responsible for dispatching incoming messages to BPEL process instances and for dispatching the incoming messages to a particular execution that can receive a message. Because the process must start with a start activity, if only one activity receives the external message, every new message will result in the creation of a new process instance. On the other hand, if more than one Receive activity exists in a BPEL process, the `MessageManager` needs to correlate the received messages to the correct instance. This is done by using correlations in the BPEL process.

To dispatch the message to the correct process, the activity that receives a message requires the identification of a unique identifier. This is done by using *message properties*. A message property queries the incoming message via a certain XPath query and generates a unique identifier for the message. For example, Figure 47 is a simple BPEL process that

contains two Receive activities. The first receive activity (ReceiveM1, createInstance='yes') receives a message in which the type is M1, which will trigger the execution engine to create a new process instance. The second receive activity (ReceiveM2, createInstance='no'), in which the receive message type is M2, will not result in a new instance. Supposing that the messages are incoming, as shown in Figure 48, the first two messages of type M1 will create two process instances (Instance 1 and Instance 2). Both are waiting for a message that has type M2. In this situation, the message manager must know which instance it should send.

Message M1 contains an attribute called ID, and this ID consists of shared information with M2. The shared information does not need to be the same, as long as it can be computed by XPath expressions to create a unique ID.

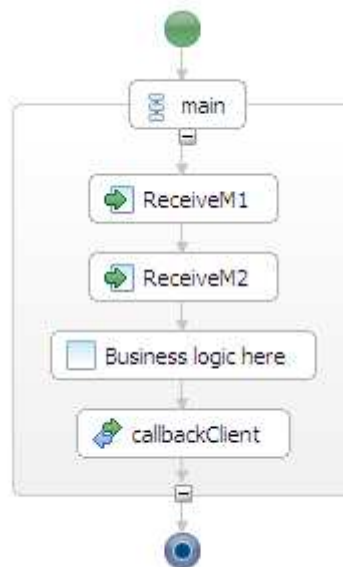


Figure 47: example of correlations

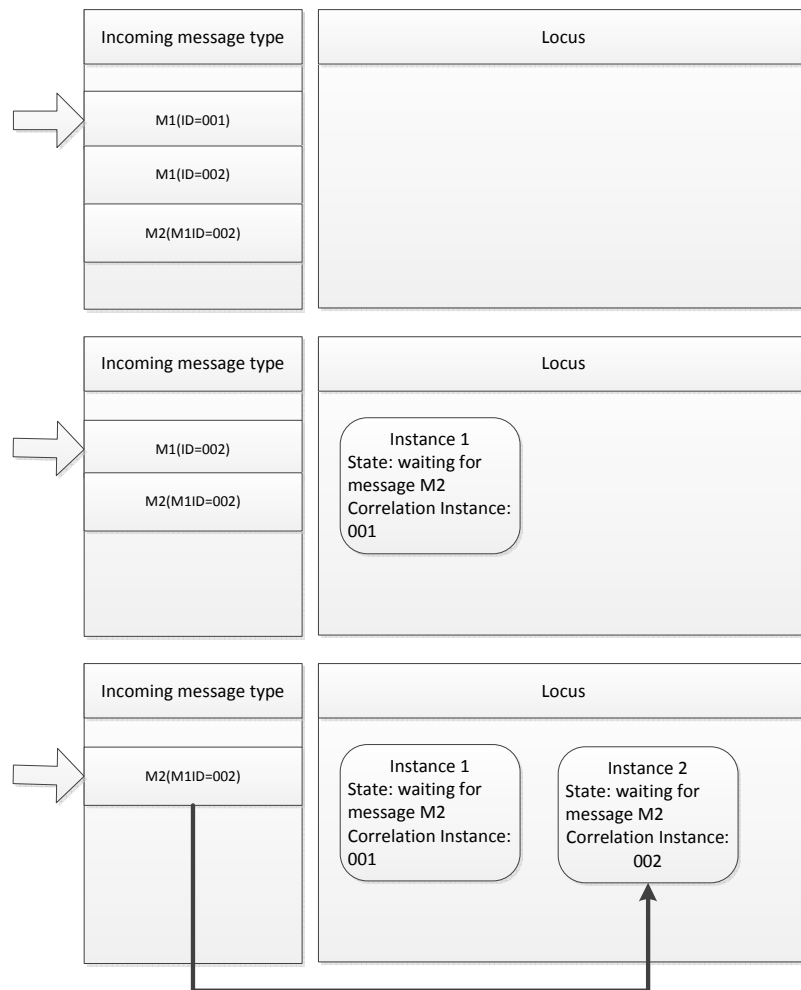


Figure 48: Illustration of correlations.

In order to send the messages to correct instances for the example in Figure 47, the BPEL process can define a correlation set as

```
<correlationSets>
  <correlationSet name="correlationSet1" properties="ID M1ID" />
</correlationSets>
```

In the Receive activity, it must also specify the correlation set

```
<receive name="ReceiveM1" ...>
  <correlations>
    <correlation set="correlationSet1" initiate="yes"/>
  </correlations>
</receive>
```

By adding the correlations, the MessageManager can now correctly dispatch the information to Instance 2, because the incoming message has the same correlation instance as instance 2.

After understanding the semantics of correlations, the next step is to use ALF to define these semantics. The relevant concepts, namely *CorrelationInstance*, are added. In

addition, when dispatching messages to an instance, it performs a check to validate whether the relevant correlation definition is satisfied. The following code shows `MessageManager::requestMessage()`, which is invoked when a `Receive` or `Pick` execution is executed.

```
private requestMessage(in execution:Execution, in
    messageInfo:MessageInfo){
    //internal execution request message
    //find the a incoming message that has the correct type
    let info:MessageInfo = this.findMessageInfo(messageInfo);
    let request:MessageRequest = new MessageRequest
        (execution,
         messageInfo.message.type,
         messageInfo.portType,
         messageInfo.operation
        );
    //when there is no instance request the message type, or correlation
    //is not satisfied
    if (info==null || ! (this.correlationSatisfied(execution,info))){
        //add this message to waiting queue
        this.messageRequests->add(request);
    }else{
        //send the message to the relevant instance
        this.sendToExecution(request,info);
    }
}
```

8.4.8 Abstract activities

Some of the concepts are outside of the scope of a language specification when these concepts are platform-specific. In the language specification, these are represented as abstract activities, which only specify the interfaces instead of the detailed method of achieving the result. For example, the `Wait` activity and the `EventHandler` all involve the use of time, particularly when waiting for a particular length of time, or when waiting until a particular time point.

```
public activity waitFor(in duration:DurationExpression){}
public activity waitUntil(in deadline:DeadlineExpression){}
```

Other examples include the behaviours regarding what a web service should do when it is invoked and how the Xpath is evaluated, as the purpose of a case study is to define the semantics of BPEL, rather than web services or Xpath. When generating reference

implementation prototypes, these operations can be implemented in the Java implementation for testing.

8.5 Checking and testing the language specification

The previous sections introduced a language specification that formalised BPEL to a model based specification. The specification contains more than 1300 lines of code. The process of developing this specification is assisted by the static checkers, and is tested iteratively when one feature has been developed by testing the generated prototype. This section summarises checking and testing during the implementation stage.

As introduced in Chapter 8, an Emfatic file that contains structural and behavioural aspects of the specification is generated. The prototype can load a BPEL process and execute it; however, this prototype also needs to interact with external web services. Thus, it is needed to set up a test environment to simulate external web services. The messages it sends need to be configured according to the test cases. If the tested BPEL processes invoke or reply to an external web service, such behaviours must be logged.

To enable these requirements, the meta-model of WSDL is extended with testing messages, which are defined as following:

- When testing a BPEL process, the developer can define a sequence of incoming messages. These messages will be sent to the `LOCUS` class when initialised.
- Given the WSDL port type, the developer can configure the message that the operation will reply. For example, the developer can create a sequence of messages whereby, when the operation is called, a message in the sequence will be sent back.

The use of this operation can simulate a normal reply or a fault reply.

Finally, the executor can be generated. To make the executor executable, the abstract activities need to be implemented since the implementation thereof is omitted in the language specification. Once these activities are implemented, a working reference implementation of the language specification is achieved. Testing instance models can be created via the Eclipse in-place model editor, and execute them using the generated Eclipse plugin. If any modifications need to be addressed, the language designer can modify the language specification and regenerate everything.

8.6 Summary

This chapter demonstrates the FQLS using a case study of BPEL. While the complete language specification of BPEL is listed in Appendix A, this chapter provides a guide as to how this specification is created from the official standard. The case study is a proof that FQLS is capable of defining a practical real-world language. In the next chapter, the case study is evaluated in terms of quality, proving that FQLS can develop a language specification while maintaining high-level quality.

Chapter 9.

Evaluation

This Chapter evaluates FQLS. While evaluating FQLS, there are two types of products that require evaluation. As the FQLS promises to enhance the quality of language specification development, in order to evaluate that, the quality of a language specification that is defined via FQLS must be evaluated. Furthermore, the quality of the FQLS, especially the software component it has included, also needs evaluation.

This chapter starts from evaluating the BPEL specification that is developed in the previous chapter. In Section 9.2, the quality of the FQLS is evaluated in terms of robustness. Finally, Section 9.3 discusses the limitations of FQLS.

9.1 Evaluating the ALF-based BPEL specification

The BPEL case study produces a realistic example of an ALF-based BPEL specification. Evaluating the quality of the BPEL specification is a method to evaluate the quality of the framework. With regard to the quality features proposed in Section 2.4.3, the interoperability, the model-based features and the executability are features of the language specification technique, which was already processed by FQLS. On the other hand, the correctness, the consistency, and the understandability require further evaluation.

This section firstly evaluates the syntactic correctness and consistency of the BPEL specification by checking its syntax validity using another ALF implementation. Following that, the semantic correctness of the BPEL specification is evaluated by testing cases, both from our design and from open source projects. Finally, the understandability of the specification is evaluated by using software metrics.

9.1.1 Syntactic correctness/consistency evaluation

When evaluating the correctness of an ALF specification, two kinds of correctness must be considered. The first is syntactic correctness, in other words, whether the ALF specification conforms to the meta-model of ALF.. The second is semantic correctness, in other words, whether the ALF specification really expresses the semantics of BPEL.

When defining a language using various technologies, the consistency between syntax models and semantic models must be ensured. On the other hand, if the language is defined in one language, such inconsistency errors will be revealed as syntactic incorrectness. This is one of the fundamental reasons for designing the FQLS. Thus, it can be assumed that the consistency will be ensured if the language specification is correct in terms of syntax.

If the ALF-based BPEL specification is successfully parsed, the editor will not report any kind of error. However, it is possible that our ALF editor produced false negatives, which means that some errors are missed. In order to eliminate this, ALF open-source reference implementation is used as a benchmark. If the ALF open-source reference implementation successfully parsed the BPEL file without reporting errors, the syntactic correctness of the BPEL specification is assured.

The ALF open source reference implementation is used as follows:

- Remove the annotations that the current ALF open source reference implementation does not support, namely `@inline` statements and `@OCL` statements.
- Remove the `@parallel` annotation, but keep the annotated block, since the `@parallel` annotation is also not supported.
- Parse the file by running the parsing command in the command line.

The ALF-based BPEL specification successfully passed the parsing and constraint checking stage. In conclusion, the ALF-based BPEL specification is correct in terms of syntax.

9.1.2 Evaluating semantic correctness by testing

Semantic correctness concerns whether the specification truly represents the language developers' intentions and can be declared to be correct by the language developers. However, this is often not the case, as the real understanding of the DSL semantics relies on the language developers' explanation. Many DSL semantics are based on the

knowledge of domain experts. In such cases, the language developers do not create language semantics, but capture the domain experts' knowledge and formalise them into a machine processable format.

In both cases (evaluated either by domain experts or language developers), the semantic correctness of a language specification relies on people's justification. People can be language developers or domain experts. As the BPEL is a well-implemented technology, and its execution result is clearly defined in the standard, it is possible to inspect the correctness of the execution manually via testing and observing the result.

Testing by design test cases

36 testing files are created by the Ecore instance editor for testing the correctness of the BPEL specification. The test cases are designed to reflect various activities. The following steps are used for testing the BPEL processes.

- Predefine the expected execution behaviour of the testing model.
- Execute the testing model.
- Analyse the log file manually to check whether the result is the correct behaviour.

The BPEL specification successfully passed all these test cases. This suggests that the BPEL case study produces expected behaviours.

Testing by real BPEL processes

The BPEL specification is tested according to various test cases. However, the expected behaviour of the test cases is derived from our understanding of the language specification. It is possible that our understanding is wrong. In this case, the BPEL specification could produce a result that is different from that of the language standard's authors. In a real scenario of DSL development, the domain experts are involved in judging whether the language specification provides the expected result. However, in our project, this is infeasible because there is no access to a domain expert.

Fortunately, BPEL is a well-implemented language. There are open source implementations of BPEL, and they conform to the language standard, while being used for commercial projects. The result produced by a mature BPEL execution engine is assumed to be the correct behaviour.

Thus, the hypothesis of this experiment is that our generated BPEL prototype produces the correct behaviour when executing processes. This correct behaviour can be achieved

by executing the same process files in a BPEL implementation. This is achieved via the following steps.

Firstly, it is necessary to select certain BPEL processes for testing. It will be more convincing if the test cases are real test cases that are not designed by the authors, but by other developers.

Such BPEL process can be found in open source projects. Google code and github.com are searched for BPEL files and Many of them are found, but also that most are fairly simple and contain only certain activities (such as invoking two web services and combining the received data) that are enclosed in a sequential activity. The author attempted to search for BPEL files that are more complex, and which are defined as follows. This process should have more complex business logic and should use fault and compensation handlers. It is better to use nested scopes and correlations.

Then the BPEL files that contained a fault handler and an event handler are searched. 20 files are obtained, most of which are clearly test files for BPEL tools. As testing real process used in real projects is desired, these files were also eliminated. Finally, the Bookstore project is selected, which is a web service that orders purchases of books. Table 14 lists the files that are used for testing, of which three come from the bookstore project, and one is the standing example of the official primer document [106].

The Apache ODE is selected as the execution engine due to its integration with Eclipse. The testing environment is listed in Table 15. Apache ODE (Orchestration Director Engine) is an implementation of BPEL, which supports the execution of both BPEL 1.1 and 2.0 standards. After deploying a BPEL process file to Apache ODE, the process is visible as a normal web service. When it receives messages, the Apache ODE will perform the execution, evaluating data and invoking the external web services.

As testing the BPEL processes in Table 14 is desired, these needed to be deployed to an Apache ODE server. The process files interact with and are composed of web services, to which accesses are not gained. However, based on the WSDL files, the interfaces of the web service are revealed. It is possible to build dummy web services that have the same interface as the required web service, and which provide a meaningful response.

Test file	Source
ShippingService.bpel	BPEL primer [106]
OrderBook.bpel	http://wangyuhere.googlecode.com/svn/trunk/ID2208/OnlineBookStoreBpel/
ShipmentBook.bpel	http://wangyuhere.googlecode.com/svn/trunk/ID2208/OnlineBookStoreBpel/
onlineBookStore.bpel	http://wangyuhere.googlecode.com/svn/trunk/ID2208/OnlineBookStoreBpel/

Table 14: Testing files

Environment	Version
Operating system	Windows XP SP3
Eclipse	Helios with BPEL designer project
Apache Tomcat	v7.0
Apache ODE	v1.3

Table 15: Environments

Finally, relevant web services are built by Java and to deploy them to an Apache Tomcat server with ODE plugin, then simulated requests that triggered different execution paths of the process, such as a normal flow, or a response that causes a fault and runs a negative flow. The log files are analysed and the activities that were executed are recorded, the sequence of the execution and the data transfer between activities.

Then the Ecore instance editor is used to rebuild the same process, giving it the same request model and executing it. Its log file shows that the sequence of the execution of the activities and the transfer of the messages is as expected.

By passing these test cases, the generated prototype is demonstrated that the expected behaviours are processed. As the prototype is derived from the ALF-based BPEL specification, if the BPEL specification is incorrect, or if the code generation is incorrect, both will result in the prototype having unexpected behaviours.

Thus, it is possible to conclude that the language specification is consistent and correct.

9.1.3 Evaluating model quality by software metrics

Language specifications are also models. The quality thereof should have the same feature as other models. Mohagheghi et al. [103] summarised methods of assessing model quality. The first method is manual inspection. While manual inspection is a good way of evaluating understandability, it would be nice to design an experiment and to recruit volunteers to manually inspect the understandability thereof. However, limited time and resources makes finding the developers of language specifications difficult, because there are not many developers that are familiar with models and DSLs. Thus, an evaluation method that does not require manual inspection is needed.

The other way of assessing model quality is to collect metrics from models. Various software metrics can reveal the quality features of the software under study. This chapter is more interested in evaluating the understandability of the BPEL case study. Many factors can impact on the understandability of models, including the organisation of diagrams and

models, the familiarity of the users and the model's simplicity or complexity [103]. Of these three types, the complexity of the models can be revealed by software metrics, as agreed by Gruhn and Laue [59]. The understandability of the models is related to the complexity metrics. Since complexity metrics have been used successfully in predicting the understandability, as well as in the cost and error rate, the evaluation of the understandability of the ALF-based BPEL specification is carried out by evaluating the complexity metrics.

Software metrics have proved effectiveness in evaluating the traditional programme, including structurally oriented programming [97, 130] and object-oriented programming [152]. Many works have tried to adapt the same metrics to models. For example, Gruhn and Laue [59] discussed ways of evaluating the complexity of business process models using several metrics, including size, the control flow complexity, and the cognitive complexity. [95, 89] applied similar metrics for analysing various domain models. These works clearly show that it is feasible to apply traditional complexity metrics to models.

The ALF-based BPEL specification can be seen as an executable meta-model, and can also be seen as an object-oriented programme. As previously discussed, both views can feasibly apply complexity metrics to the specification. Thus, the task of the evaluation is to calculate the complexity metrics of the BPEL specification.

Our hypothesis is that, compared to other implementations of the BPEL semantic specification, the ALF-based BPEL specification has a lower metrics in software complexity.

Comparable specifications

Evaluating the complexity of the ALF-based BPEL specification requires the comparison of the specification with other BPEL specifications. Thus, it is possible to establish metrics and to compare them. The official semantic specification of BPEL is based on text, which makes it impossible to compare it. Fortunately, there are two other works that formalise BPEL semantics. Fahland and Reisig [38] specified the semantics of BPEL using an ASM language. The ASM specification is created by first mapping the BPEL meta-model manually to an abstract state machine and then using a combination of mathematical language and procedural execution rules to specify semantics. Stahl [145] used the Petri net for the same purpose. The resulting semantic specification consists of many Petri net diagrams. Both can be categorised as translational semantics. As a result,

the ASM specification and the Petri net specification are considered to be comparable implementations of the ALF specification.

Since the complexity of the three specifications will be calculated and will repeatedly be referenced in the remainder of the thesis, the ALF-based BPEL specification produced in the case study will be referred to as the ALF specification, Fahland and Reisig [37] (complete specification in [38]), will be referred to as the ASM specification, and Stahl [145] will be referred to as the Petri net specification.

Coverage of the official standard

The three BPEL semantics specifications are research works; therefore, not all parts of the official documents are covered. Of the three specifications, ALF and ASM have similar coverage. Activities, handlers, instances, correlations and message management are all defined. However, ASM specification has many abstract functions whereby only the interface is defined and the semantics are defined by the text. The ALF-based specification also contains certain abstract functions, but these are limited to the functions that the semantics intentionally left undefined, or the semantics that are dependent on the implementation platform. Hence, the amount of abstract functions in the ALF specification is significantly smaller. The Petri net specification did not define instances, correlations or message management; therefore it has significantly less coverage than the other two. Of the three specifications, only ALF specification defines both the abstract syntax and the behavioural semantics.

Size analysis

The most commonly used metrics for measuring the size of a programme are the lines of code. These are the most commonly cited software metrics [111]. The Lines of Code are measured by first remove any empty lines and comments. The line numbers were then calculated. However, since the ALF specification includes the syntax, the semantic definition and the definition of a WSDL meta-model, ASM specification only defines the semantics. Thus, to make the comparison fair, the definition of the BPEL meta-model, the WSDL meta-model is removed from the specification, but any other classes that are related to behaviours, such as the execution class, the instance class and managers, remain. By doing this, Table 16 lists the Lines of codes of the two specifications.

	ALF	ASM
LOC - complete	1588	1506
LOC - exclude syntax	1318	1498

Number of methods	121	147
--------------------------	-----	-----

Table 16: LOC comparison.

Table 16 shows the size of the ALF specification in terms of LOC and the number of methods. Although the size of the specification is not only affected by the technology for defining the specification, it can also be affected by the design of the specification, the skills of the developer, and the coverage of the specification. Considering that the coverage of the ALF specification is better than that of the ASM specification, and that the design of the specification is similar, both are derived from the same official language specification. Thus, it is concluded that the ALF specification is smaller in size than that of the ASM specification.

Cyclomatic Complexity analysis

Cyclomatic Complexity [97] believes that the complexity of the control flow graph could reflect the complexity of the programme. In addition, the metric is language independent and could even measure graphs. If a programme were to be translated into a control graph, assuming that the count of nodes is n , the count of edge would be e , Cyclomatic Complexity (CC), defined as

$$CC = e - n + 2$$

A practical method included in [97] for calculating CC is counting the number of control structures (if, for, while, logical and, or, etc.). The Cycomatic complexity can be calculated by the following formula.

$$CC = \text{count} + 1$$

This formula is used for calculating the Cyclomatic complexity of a single-entry-single-exit programme. In both the ALF and ASM specifications, there are certain methods that contain multiple-exit programmes. However, there are various ways of calculating the Cyclomatic Complexity of a multiple-exit programme. According to Henderson-Sellers and Tegarden [66], Cyclomatic complexity could be adapted to multiple exist programmes just by consisting as an existing as a control structure²⁵. Thus, the count of structures is defined as

$$\begin{aligned} \text{count} = & \text{sum}(\text{conditional statements}) + \text{sum}(\text{loop statements}) \\ & + \text{sum}(\text{logical operators}) + \text{sum}(\text{exits}) \end{aligned}$$

²⁵ It is also identify that [64] suggests a different way of calculating the complexity. However, the multiple exit points only appear a few times in the ALF specification, and do not exist in the ASM specification. Applying [64]'s work will not affect the conclusions, because this will only reduce the complexity of the ALF specification.

Here, the conditional statements are defined as normal conditional statements, including if/switch, and accept statements are also treated in the same way as switch statements. Loop statements include for/while/do while loops. Finally, The *Total Cyclomatic Complexity (TCC)* is defined as the summarised CC of the entire project:

$$TCC = \sum_{i=1}^k CCI - k + 1$$

The above method is used in order to collect the Cyclomatic complexity of ALF and ASM specification. The Petri net specification, however, is different from the other two. Since it is not structured programming but graphs, in terms of the calculation of the complexity of a Petri net, Mao's [95] approach is applied by calculating the nodes and edges.

Mao presents a technique to calculate the Cyclomatic complexity of a business process that is defined as a Petri net. The method simply treats places and transitions as nodes, and allows the link between them to be the edges. By calculating their numbers, the Cyclomatic complexity of the Petri net can be derived. This method was chosen due to the limited work in applying Cyclomatic complexity to Petri nets, and this is the exact method that solves the same problem as ours.

The method for gathering the complexity metrics is to first divide the language specification into several *code units*, and then to gather the Cyclomatic complexity manually. A code unit contains the portion of the specification that defines a specific topic. The specifications are split into comparable code units. A code unit in ALF specification may contain one class or several related classes, and a code unit of the ASM or the Petri net specification may contain the rules or graphs listed in one subsection. Since the architectures of the three specifications are different, the code units are not guaranteed to have the same semantics. However, each code unit reflects the semantics of the same BPEL structure, the semantics of which are similar.

Table 17 shows the result of the Cyclomatic complexity of the three specifications of BPEL. Each line of the table stands for a code unit of the language specification. Considering the coverage of the specification, the Petri net specification has the lowest coverage of the official standard but the highest complexity, which shows that a graph-based language definition could increase the complexity and thus affect the understandability of the language standard.

The ALF and ASM specification has similar coverage, and their architectures are also similar. It is clear that the ALF specification is the least TCC of them. When looking at each code unit, most of the CC of ALF is smaller than that of ASM. When going deeply into the code, if the code blocks are very simple, the CC will make no significant difference. For example, Reply, Terminate, Empty, Wait, has nearly the same CC. When the code unit become more complex, the ALF specification becomes simpler than the ASM specification. There are exceptions; for example, the Flow, Invoke unit of ALF specification is significantly more complex than that of ASM. The reason is that the ASM has defined the fault and event handlers as a separated unit, while in the ALF specification, the event handling is managed by the relevant activity instances that cannot be calculated individually.

	ALF	ASM	Petri Net
Reply	7	7	11
Flow	16	8	2
Flow Link²⁶			31
Terminate	4	5	4
While	11	11	4
Invoke	24	13	35
Empty	4	5	2
Wait	5	5	6
Throw	4	4	2
Receive	9	7	13
Assign	10	30	11
Sequence	11	10	2
Switch	16	10	7
Pick	18	31	17
Scope	47	52	99
handlers²⁷		23	42
process	10	9	8
Compensation	7	24	67
Message manager	26	36	0
Basic activity	24	26	0
Total	253	316	363

Table 17: Cychomatic complexity comparison.

²⁶ The ALF specification and ASM specification combines Flow link semantics with flow, and they are combined with Flow semantics. Thus, the complexity of flow of ALF and ASM is greater than that of Petri net.

²⁷ The ALF specification combined the semantics of fault handlers with scope, process and basic execution; thus, there is no separate class to deal with a fault handler.

Cognitive complexity

Cychromatic complexity can reflect the physical complexity of software; however, it considers only the control and conditional flows of the programme, whereas in the specifications that are evaluated, there are certain statements that are different from the normal definition of control flows, such as signals, sequence expressions and logical expressions in ASM. Shao and Wang [139] also argued that such metrics did not reflect the effort to understand the software. They proposed a metric based on cognitive weight that could evaluate programmes with more advanced structures. This could reflect the cognitive and psychological complexity of software by considering both the internal structures and the behaviours they process. In the case study, the cognitive complexity is calculated by first assigning a cognitive weight to all the control structures, which are shown in Table 18. The cognitive weights are categorised according to the type of control structure. Because ASM formalism and ALF language have special control structures, there is a column entitled “language”, which indicates the languages that such control structures.

Category	Language	Control Structures	Weight(W_i)
Sequence	Both	Sequence	1
Branch	Both	if-then-else	2
	Both	case, if-elseif	3
Iteration	Both	for	3
	Both	While	3
	Both	do while	3
Function calls	Both	function calls	2
	Both	Recursion	3
Concurrency	ALF	Parallel	4
	ASM	on signal	4
	ALF	Accept	4
Sequence operation	Both	select/forAll/exists	4
	ASM	other logic structure	4

Table 18. Cognitive weights of control structures.

Cognitive complexity adds the cognitive weights of its q linear blocks. If a block contains nested blocks, the cognitive weights are defined by multiplying the cognitive weights of the nested blocks. The total cognitive weight can be calculated by the formula

$$W_c = \sum_{j=1}^q \left[\prod_{k=1}^m \sum_{i=1}^n w_c(j, k, i) \right]$$

$w_c(j, k, i)$ is the complexity weight of the statement.

The following steps are used to calculate the total cognitive complexity.

- The complexity of a code block is the sum of the weight of the control flow.
- A statement that does not contain nested code blocks has the weight specified in Table 18.
- The complexity of a statement that has nested code blocks is the weight multiplied by the sum of the nested blocks.
- The total cognitive complexity of a code unit is the sum of the complexity of the code blocks.

In order to explain how this works, the code from the ALF specification is took and the complexity thereof is calculated. The code below is the classifier behaviour of `AssignExecution` and the complexity of each statement or code block is written as comments

```
//Total: 2+72=74
//2
this.setState(State.DISABLED);
let completed:Boolean = false;
//3*24=72
while (! completed){
    //4*(2+2+2)=24
    accept(SignalStart){
        //2
        this.enable();
    }
    or accept(SignalLinkActivited){
        //2
        this.running();
        completed = true;
    }
    or accept(SignalTerminate){
        //2
        this.terminate();
        completed = true;
    }
}
```

Table 19 listed the cognitive complexity of the ALF specification and the ASM specification. This table is similar to the table of Cychromatic complexity. The ALF specification is less complex than is the ASM specification in terms of total cognitive complexity.

When looking into individual code units, the distribution of the complexity is revealed. For simpler units, like `wait` and `throw`, the complexity of ALF and ASM is similar. In fact, the ALF specification is slightly higher than that of ASM. This is because the pattern of receiving incoming signals uses a while loop, which results in the block always having a nested loop that multiplies the complexity. In the ASM specification, there is a specially designed syntax for receiving the signal continuously, which results in the complexity of receiving a signal being lower. However, for more complex code blocks, such as `Assign`, `Compensate`, `Scope` and `Sequence`, the ALF specification has a much smaller complexity.

Name	ALF	ASM
assign	117	278
compensate	113	665
correlation	8	
empty	78	84
structured activity	300	236
flow	168	154
invoke	403	407
locus	360	
message	244	
pick	257	294
receive	201	156
reply	191	168
scope	553	959
sequence	163	428
switch	251	184
terminate	80	86
throw	99	70
wait	97	86
while	201	256
event handler		472
fault handler		460

Total	3884	5443
--------------	------	------

Table 19: Cognitive complexity comparison.

One reason that the ASM specification is more complex than that of ALF is that its nature makes the reuse of code difficult. Many codes are duplicated. The ASM specification also uses extremely complex logic to define the semantics as a post condition, which makes the control flow complexity large.

Discussion

This section evaluated the physical lines of code, the Cychomatic complexity thereof, and the cognitive complexity of the ALF-based BPEL specification of the ASM-based BPEL semantic specification and a Petri net BPEL semantic specification. The result summarised in the tables clearly show the lower complexity of the ALF specification, thus validating the hypothesis.

Our method of using software metrics for evaluating the complexity relies on the assumption that the complexity metrics used are independent of language. This claim is supported by other works; for example, McCabe [97] and Shao and Wang [139] believe that CC and cognitive complexity are language independent, while there are other metrics, like the first complexity metrics proposed by Halstead [62], which are language dependent.

Although differences in the complexities between ALF/ASM specifications are large (the total cognitive complexity of ALF specification is 28.6% less than that of ASM, and the total Cychomatic complexity is 19.9% smaller), there is a concern that may affect validity: the differences of the complexity are not proven to be statistically significant. However, evaluating the statistical significance of software metrics is still a challenging topic, and cannot be done if a large code database exists. Considering the age of ALF, although such a large database is still missing, it is nearly impossible to prove the statistical significance of the complexity evaluation.

9.2 Evaluating static checkers of the ALF Language Specification Framework

The consistency, correctness and complexity of the ALF-based BPEL specification are evaluated in the previous section. These features are the most important factors that affect the quality of a language specification. It demonstrates the effectiveness of producing a high quality language specification using FQLS because, in the process of developing the

BPEL case study, the framework helped to identify many errors in the earlier stages. Because some of the built-in static checkers were designed and developed while developing the BPEL case study, the true ability of the static checkers needs to be tested in a better way than using the author's report, so that the static checkers are not only useful to the BPEL case study, but are generally related to ALF programmes.

Thus, in this experiment, the robustness of the static checkers was evaluated. The robustness was defined as the built-in checkers of ASLF, which will still have expected behaviour when given ALF programme from other sources.

One widely used method for evaluating the static checkers is to use the static checker to analyse a widely known open source project and to analyse its result, as per Findbug [70]. The same principle holds when evaluating a static checker for ALF; however, ALF does not have such a large open-source project. Finally, the sample was collected from the testing file²⁸ for the ALF open-source reference project and, after extracting the source code, the test files were placed in the 'test-x' folder.

The test files of the ALF open-source reference implementation contain 25 test files and four helper files. The content varies from a simple hello world, to evaluating complex expressions and ALF-specific statements. These programmes can be treated as real examples of ALF programmes. These test ALF files were opened using the ALF code editor. By analysing the results, it can be seen that these test files contain several bad practices and possible errors.

- Many of the test files use variables without declaring the type. This is considered to be bad practice. Therefore, the result is a warning, or possibly an error, if the type cannot be inferred by the static checker.
- Many variables are used without initialisation, particularly when used as an output variable.
- In `Expressions_Assignment_Feature.alf`, line 34, there is a type error, where `c.y` has a type `Integer` and cannot be assigned to `null`.
- `Expressions_Assignment_Feature_Indexed.alf`, line 31, similar to the above.
- `Expressions_Constructor_Destructor.alf`. In lines 27, 28 and 34, the constructor has the same name as a public member. For example:

²⁸ <http://lib.modeldriven.org/MDLibrary/trunk/Applications/Alf-Reference-Implementation/dist/test-x.zip>

- `public transferred: Boolean = false;`
- `@Create public transferred(in employeeInfo: Employee) {`
- `Expressions_Sequence_Expansion_Reduction.alf,` from line 25: The parameter name is the same as a keyword.

The bad practices identified above clearly show the necessity of having basic static checkers to assure the correctness of the programmes, because the examples in the standards have potential errors.

9.3 Limitations

FQLS uses fUML as its semantic basis. This indicates that it is not possible to specify the semantics that fUML cannot specify. The fUML standard [55] and Selic [137] both summarise the limitations of fUML, in that it only supports event-driven, discrete behaviours. If a DSL that simulates the physical world has continuous behaviours, such semantics cannot be directly specifiable by fUML. Such behaviours may possibly be supported by certain profiles that remain unexplored.

Various arguments [111] imply that many complexity metrics (including Cyclomatic Complexity) are correlated to the size of the programme; in other words, the LoC. If this is true, an evaluation that uses metrics in the thesis would be less convincing. This is a limitation of the evaluation method. Another way of evaluating the understandability is to conduct an experiment that surveys language engineers. This is extremely challenging because the number of language engineers is limited, making the task almost impossible because of limited resources and time.

The generated prototype also has a limitation, because the generated prototype does not ensure thread safety. EMF “can safely be used in multi-threaded environments; however, EMF does not, itself, ensure thread-safety in application model implementations²⁹”. The prototype inherited the limitations of EMF; thus, attributes shared by multiple threads may have race conditions.

²⁹ <http://wiki.eclipse.org/EMF/FAQ>

9.4 Summary

This chapter evaluated the FQLS by confirming the hypothesis. The evaluation demonstrated that the BPEL specification developed by FQLS is a correct and consistent specification and, compared to other implementations of the language, has lower complexity that indicates better understandability. The effectiveness of the static checkers of ALF was also tested by examining the testing code of the ALF open source reference implementation.

Chapter 10.

Conclusion and further work

10.1 Conclusions

The journey of this thesis starts from the problems in domain specific language development. The current methods for developing domain specific language specification, especially a specification contains both abstract syntax and behavioural semantics, deliver language specifications that are in low quality. The problem of the quality of language specifications leads to the research objectives of the thesis.

The first research objective is “*To investigate the requirements of a high quality DSL specification.*” By analysing existing approaches and existing language specifications, Section proposes that the language specification should has consistency, correctness, executability, understandability, expressiveness, interoperability, and should be model-based.

The second research objective is “*To design an approach that could define a DSL in a unified manner, fulfil the requirements of a high quality DSL specification*”. The FQLS approach introduced in Chapter 4 and detailed in later chapters is the answer to this objective.

The third research objective is “*To create a software development process that applying the new definition approach*”. Such a software development process is presented in Section 4.3.

The last research objective is “*To develop a framework for supporting the software development process that assists the development of DSLs*”. A software tool chain that supports specification development, static checking and testing are presented in Chapter 5, Chapter 6, and Chapter 7.

10.1.1 Summary of contributions

The scientific contribution of the thesis is the method for defining domain specific language specification. Its purpose is to deliver high quality language specification. The goal is reached by using established technologies, i.e. the ALF, fUML, static analysis and

projects based on the Eclipse modelling framework. In contrast to other approaches, the FQLS approach eliminates inconsistencies between abstract syntax and semantics. It uses expressive language and produces interoperable and understandable language specifications.

The thesis also contributes three technical area. The first technical contribution is a software architecture that supports the language specification technique and the software development process. It proposes a tool chain that contains editor for the language specification, static checkers for identifying errors, model transformer which translates the ALF specification to fUML models, and code generator that translates the ALF specification to Java, which enables testing the language semantics.

The second technical contribution is a software development process that applies quality assurance on the language specification itself while developing the language specification. The products, the roles, the tasks and the guidelines of developing language specifications are identified and captured by SPEM meta-model. The process proposes to perform checks and testing directly on the specification rather than leave them until implementation stage, thus, makes identifying errors in an early stage.

The last technical contribution is that the case study demonstrates that fUML can be used to define the semantics of DSLs despite its original purpose is to define UML semantics. The case study creates the first model-based BPEL specification that defines abstract syntax and behavioural semantics in a unified way.

10.1.2 Summary of evaluation

To demonstrate that FQLS can produce a high quality language specification, a real DSL – the BPEL is chosen as a case study. In the case study, the abstract syntax and behavioural semantics are defined by ALF, while applying the software development process and using the software tool chain. While developing the language specification, the specification is statically checked for syntax and bad practices on going, and being tested in each iteration.

Then this case study is used as a source for evaluation. The experiments and results are summarised as:

- The syntactical correctness of the specification is evaluated by checking the syntactical correctness by the ALF open source implementation. The semantic

correctness is demonstrated by running test cases and real process examples. Both of the methods show no error in the ALF –based BPEL specification.

- The BPEL case study is evaluated by software metrics, and the result is compared to other approaches for defining BPEL. The ALF-based BPEL specification is less complex than the comparable specifications in all the selected metrics, which suggests the FQLS method is better in understandability.
- The static checkers are evaluated by checking the programs presented in the ALF open source implementation. Several bad practices that suggest errors are identified. This experiment shows the robustness of the static checkers. They are not only effective for checking errors in the BPEL specification.

The evaluation confirms that a language specification defined as the FQLS has better quality in terms of correctness, consistency, understandability. Since the FQLS used an interoperable, expressive language, the language specification that the FQLS produced unifies abstract syntax and behavioural semantics, the hypothesis of the thesis is confirmed.

10.2 Further work

The OMG is seeking the methods of defining the semantics of UML family by ALF. Currently, the fUML standard defines the semantics as restricted Java code. It will be much consist and precise, if the semantics are defined by ALF. The “structures of Precise Semantics of UML Composite Structures” [54] is a working on draft that describes the semantics of UML composite structures (e.g. the elements of the composite diagrams and the collaboration diagrams), using fUML as its vocabulary. Eventually, the same work can be extended to any language that is defined as a meta-model.

If ALF and fUML become a widely accepted way for defining languages, it is possible to reuse language definitions just like reuse software components. This direction can be further researched in several routes

- Language composition. One requirement in language development is to composite two different but similar languages into one language, thus the composited language contains the expressiveness and the concepts of both the languages. If these two languages are defined by ALF, it is likely that the process of composition

will be easier since they share many similarities. It is also worth seeking automatic ways to composite languages that are defined using ALF.

- Language evolution. DSLs change and evolve throughout their life cycle. The meta-model may be extended with new concepts, or changes to another model. The semantics may change as well. Hence, it is worth of research on how FQLS could support the development of an evolving language specification that still keeps the quality of the specification.

Another direction of further research is continuing to improve the FQLS to provide advanced supports for language developers. The possible works include:

- Adding automatic unit tests. Currently, testing is done by executing the test cases and then checks the result manually. It is worth to develop an automatic testing framework.
- Revising the ALF to Java code generation, generating from ALF sequence expression to Java lambda expression. Using Java lambda expression will avoid the complex mechanism of generating OCL and limitations of sequence expression.
- The Model Library of ALF is not fully implemented. Thus, some built in data type, such as BitString, Set, will report type not found error, which is not an error.
- Addressing known bugs. There are some known bugs, such as failing to deal with treaty ++ -- operators performed on a property access expression.
- Developing full support of the ALF model library. The Model Library of ALF is not fully implemented. Thus, some built in data type, such as BitString, Set, will report type not found error, which is not an error.

Appendix A. Complete specification of BPEL

```

/*****
 * BPEL META MODEL
 * *****/

    public enum State{DISABLED,ENABLED, RUNNING, COMPLETED,
FAULTED,TERMINATED, COMPENSATING, STOPPED}

    /* WSDL */
    public class WSDL{
        public messages:compose MessageType[*];
        public portTypes : compose PortType[*];
        public properties:compose Property[*];
        public serviceLinkTypes:compose ServiceLinkType[*];
    }

    public class ServiceLinkType{
        public name:String;
        public role:compose Role[*];
    }

    public class Role{
        public name:String;
        public portType:PortType;
    }

    public class MessageType{
        public name:String;
        public parts:compose Part[*];
    }

    public class Message{
        public type:MessageType;
        public value:String;
    }

    public class Part{
        public name:String;
        public typeName:String;
        public elementName:String;
    }

    public class Property{
        public name:String;
        public type:String;
    }

    public class PortType{
        public name:String;
        public operations:compose Operation[*];

        public invokeOperation(inout operation:Operation, inout
message:Message){

```

```

    }
}

public class Operation{
    public name:String;
    public input:compose Input;
    public output:compose Output;
    public fault:compose Fault;

    public testRespondMessage:compose MessageInfo[*];

    public invoke(in message:Message){
        log(" " + message);
        if (this.testRespondMessage->size()>0 && this.output!=null){
            let messageInfo:MessageInfo = this.testRespondMessage[0];
            this.testRespondMessage->remove(0);
            messageManagerInstance().SignalReceiveMessage(messageInfo);
        }
    }

    public receiveResponse(in message:Message){
        log(" "+message+" received");
    }
}

public abstract class MessageReference{
    public name:String;
    public messageType:MessageType;
}

public class Input specializes MessageReference{}
public class Output specializes MessageReference{}
public class Fault specializes MessageReference{}

/* ACTIVITIES */

public abstract class Expression{
    public body:String;
    public expressionLanguage:String;
}

public class BooleanExpression specializes Expression{
    private count:Integer;
    public eval():Boolean{
        //evaluate the expression, omitted
        if (this.body=="true"){
            return true;
        }
        else
        {return false;}
    }
}

public class DurationExpression specializes Expression{}

public class DeadlineExpression specializes Expression{}

public class Variable{
    public name:String;
    public value:String;
}

```

```

    public messageType:MessageType;

    // public setVariable(in message:Message){
    //     this.value = message.value;
    // }

    public copyFrom(in variable:Variable){

    }

    public copyPart(in inVar:Variable, in inPart:Part, in
selfPart:Part){

    }

    public copyLiteral(in literal:String, in selfPart:Part){

    }

    public getMessage():Message{
        let message:Message = new Message();
        message.value = this.value;
        message.type = this.messageType;
        return message;
    }

}

public class PartnerLink{
    public name:String;
    public serviceLinkType:ServiceLinkType;
    public myRole:Role;
    public partnerRole:Role;
}

public class BPELProcess specializes Scope {

}

public abstract class Activity{
    public name:String;
    public joinCondition:compose BooleanExpression;
    public suppressJointFailure:Boolean = false;
    public enclosedScope():AbstractScope{
        let parent:Activity = container(this);
        while (! (parent instanceof AbstractScope)){
            parent = parent.container;
        }
        return (AbstractScope)parent;
    }
}

// public abstract class CorrelationActivity{
//     public correlations:compose Correlation[*];
// }
//
// public class Correlation{
//     public set:CorrelationSet;
//     public initiate:Boolean;
//     public pattern:String;
// }

```

```

public abstract class AbstractScope specializes StructuredActivity{
    public partnerLinks:compose PartnerLink[*];
    public variables:compose Variable[*];
    public faultHandler:compose FaultHandler;
    public eventHandlers:compose EventHandler;
    public compensationHandler:compose CompensationHandler;
    public correlationSet:compose CorrelationSet[*];
}

public class CorrelationSet{
    public name:String;
    public properties:Property[*];
}

public class CompensationHandler{
    public activity_:compose Activity;
}

public class EventHandler{
    public events:compose OnEvent[*];
}

public class OnEvent{
    public partnerLink:PartnerLink;
    public portType:PortType;
    public operation:Operation;
    public messageType:MessageType;
    public scope:compose Scope;
}

public class Scope specializes AbstractScope{}

public class Compensate specializes Activity{}
//
// public class CompensateScope specializes Activity{
//     public target:Scope;
// }
//
//
public assoc Link_Activity_1{
    public sources:Link[*];
    public target:Activity;
}

public assoc Link_Activity_2{
    public targets:Link[*];
    public source:Activity;
}

public class Link{}

public assoc StructuredActivity_Activity{
    public container:StructuredActivity;
    public activities:compose Activity[*] sequence;
}

public abstract class StructuredActivity specializes Activity{
    public primaryActivity():Activity{
        return this.activities[0];
    }
}

```

```

public assoc Flow_Link{
    public links:compose Link[*];
    public flow:Flow;
}

public class Flow specializes StructuredActivity{
}

public class While specializes StructuredActivity{
    public condition:compose BooleanExpression;
}

public class FaultHandler{
    public catch_: compose Catch[*];
    public catchAll:compose CatchAll;
}

public class Catch{
    public faultName:String;
    public faultVariable:Variable;
    public faultMessageType:MessageType;
    public activity_:compose Activity;
}
public class CatchAll{
    public activity_:compose Activity;
}

public class Sequence specializes StructuredActivity{}

public class Switch specializes StructuredActivity{
    public case_:compose Case[*];
    public otherwise:compose Activity;
}

public class Case{
    public condition:compose BooleanExpression;
    public activity_:compose Activity;
}

public class Pick specializes Activity{
//    public onMessage:compose OnMessage[*];
    public onEvent:compose AbstractOnEvent[*];
}

public abstract class AbstractOnEvent {
    public activity_:compose Activity;
}

public class OnMessage specializes PartnerActivity,AbstractOnEvent{
//    public activity_:compose Activity;
}

public class OnAlarm specializes AbstractOnEvent{
    public for_:compose DurationExpression;
    public until_:compose DeadlineExpression;
}

public class Empty specializes Activity{}

public class Terminate specializes Activity{}

```

```

public abstract class PartnerActivity specializes Activity{
    public partnerLink:PartnerLink;
    public operation:Operation;
    public variable:Variable;
    public portType:PortType;
    public correlations:compose Correlation[*];
}

public class Receive specializes PartnerActivity{
    public createInstance:Boolean;
}

public class Reply specializes PartnerActivity{
    public faultName:String;
}

public class Invoke specializes PartnerActivity{
    public inputVariable:Variable;
    public outputVariable:Variable;
}

public class Throw specializes Activity{
    public faultName:String;
    public faultVariable:Variable;
}

public class Wait specializes Activity{
    public for_:compose DurationExpression;
    public until_:compose DeadlineExpression;
}

public class Assign specializes Activity{
    public copy:compose Copy[*];
}

public class Copy {
    public from_:compose FromSpec;
    public to_:compose ToSpec;
}

public abstract class FromSpec{}
public abstract class ToSpec{}

public class FromVariablePart specializes FromSpec{
    public variable:Variable;
    public part:Part;
}

public class FromLiteral specializes FromSpec{
    public literal:String;
}

public class FromExpression specializes FromSpec{
    public expression:String;
}

public class FromProperty specializes FromSpec{
    public variable:Variable;
    public property:Property;
}

```



```

public class ToVariablePart specializes ToSpec{
    public variable:Variable;
    public part:Part;
}

public class ToProperty specializes ToSpec{
    public variable:Variable;
    public property:Property;
}

/*****
* EXECUTIONS
*****/

public abstract active class Execution{
    public state:State;
    public receive signal SignalStart{}
    public receive signal SignalTerminate{}
    public receive signal SignalLinkActivated{}

    //send SignalCompleted to its container
    //    // OCL("
    //self.activity_.sources->size()==0
    //or
    //self.activity_.sources
    // ->collect(e|e.source.execution)
    // ->forall(e|e.state==State::COMPLETED)
    //")
    public allLinksActivated():Boolean{
    //    return this.activity_.sources
    //        ->collect e (e.source.execution)
    //        ->forall e (e.state==State.COMPLETED)
    //        || this.activity_.sources->isEmpty();
    //    if (this.activity_.sources->isEmpty()){
    //        return true;
    //    }else{
    //        return this.isAllDetermined();
    //    }
    }

    private isAllDetermined():Boolean{
        let allDetermined:Boolean = true;
        let flowExecution:FlowExecution =
        (FlowExecution)locus().findExecution(instance(this),(Flow)this.activity_.
        sources[0].flow);
        for (Link l:this.activity_.sources){
            let linkInstance:LinkInstance =
            flowExecution.linkInstanceOf(l);
            if (! linkInstance.isDetermined){
                allDetermined = false;
                break;
            }
        }

        return allDetermined;
    }

    public propagatedDPE():Boolean{
        if (this.activity_.sources->isEmpty()){

```

```

        return true;
    }
    if (this.activity_.joinCondition==null){
        //default semantics:if one linkInstance(isDetermined and
success )
        let flow:Flow = this.activity_.sources[0].flow;
        let flowExecution:FlowExecution =
        (FlowExecution)locus().findExecution(instance(this),flow);

        for (Link l:this.activity_.sources){
            let linkInstance:LinkInstance =
flowExecution.linkInstanceOf(l);
            if (linkInstance.isDetermined && linkInstance.success){
                return true;
            }
        }
        return false;
    }else{
        return this.activity_.joinCondition.eval();
    }
}

public abstract doAction(){}
public running():Boolean{
    if (this.state==State.ENABLED && this.allLinksActivited()){
        let linkSucc:Boolean = this.popagateDPE();
        if (linkSucc){
            this.doAction();
        }else{
            if (this.activity_.suppressJointFailure){
                this.complete(false);
            }else{
                this.setState(State.FAULTED);
                //suppressJointFailure
                let faultInfo:FaultInfo = new FaultInfo();
                faultInfo.faultName = "bpws:joinFailure";
                // faultInfo.faultMessage = "";
                this.throwFault(faultInfo);
                return true;
            }
        }
    }
    return !linkSucc;
}
return false;
}

public notifyTargets(in succ:Boolean){
    // for (Link link: this.activity_.targets){
    //     link.target.execution.SignalLinkActivited();
    // }
    // this.activity_.targets->collect e
    (e.target.execution.SignalLinkActivited());
    if (!this.activity_.targets->isEmpty()){
        let flow:Flow = this.activity_.targets[0].flow;
        let flowExecution:FlowExecution =
        (FlowExecution)locus().findExecution(instance(this),flow);
        //@parallel
        for (Link link: this.activity_.targets){
            let linkInstance:LinkInstance =
flowExecution.linkInstanceOf(link);
            linkInstance.sourceDetermined(true);

```

```

        linkInstance.success = succ;
    }
}

public setState(in s:State):Boolean{
    this.state = s;
    return true;
}

public throwFault(in faultInfo:FaultInfo){
    let scopeExecution:ScopeExecution =
(ScopeExecution)locus().findExecution(instance(this),this.activity_.enclosedScope());
    scopeExecution.SignalFaulted(faultInfo);
    //do something with the faultHandler.
}
public enable(){
    if (this.state==State.DISABLED){
        this.setState(State.ENABLED);
        if (this.activity_.sources->isEmpty()){
            this.SignalLinkActivited();
        }
    }
}
public terminate(){
    this.setState(State.TERMINATED);
}

public complete(in linkSucc:Boolean){
    this.setState(State.COMPLETED);
    //send parent
    this.parentExecution.SignalChildFinished(this);
    //send link targets
    if (! this.activity_.targets->isEmpty()){
        this.notifyTargets(linkSucc);
    }
}
}do{}

public assoc Execution_Activity_1{
    public execution:Execution[*];
    public activity_:Activity;
}

public active class EmptyExecution specializes Execution {

    public doAction(){
        this.setState(State.RUNNING);
        this.complete(true);
    }
}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (! completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivited){
            this.running();
        }
    }
}

```

```

        completed = true;
    }
    or accept(SignalTerminate){
        this.terminate();
        completed = true;
    }
}

}

public active class AssignExecution specializes Execution{
    public getAssign():Assign{
        return (Assign)this.activity_;
    }

    public doAction(){
        this.setState(State.RUNNING);

        for(Copy co:this.getAssign().copy){
            this.doAssign(co.from_,co.to_);
        }

        this.complete(true);
    }

    public doAssign(inout fromSpec:FromSpec, inout toSpec:ToSpec){
        //TODO assign value
        if (fromSpec instanceof FromVariablePart && toSpec instanceof
ToVariablePart){
            let frm:FromVariablePart = (FromVariablePart)fromSpec;
            let t:ToVariablePart = (ToVariablePart)toSpec;
            if (frm.part==null){
                t.variable.copyFrom(frm.variable);
            }
            else{
                t.variable.copyPart(frm.variable,frm.part,t.part);
            }
        }else if (fromSpec instanceof FromLiteral && toSpec instanceof
ToVariablePart){
            let frm:FromLiteral = (FromLiteral)fromSpec;
            let t:ToVariablePart = (ToVariablePart)toSpec;
            t.variable.copyLiteral(frm.literal, t.part);
        }
    }
}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (! completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivated){
            this.running();
            completed = true;
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}
}

```

```

public active class ReplyExecution specializes Execution{

    public getReply():Reply{
        return (Reply)this.activity_;
    }

    public doAction(){
        this.setState(State.RUNNING);
        let variable:Variable = this.getReply().variable;
        this.getReply().operation.receiveResponse(variable.getMessage());
        //init correlation

        if (!this.getReply().correlations->isEmpty()){
            //this.getReply().correlations
            // ->select cor (cor.initiate)
            // ->collect vi (...)
            for (Correlation cor:this.getReply().correlations){
                if (cor.initiate){
                    let
vi:VariableInstance=instance(this).findVariableInstance(variable);
                    instance(this).correlationManager.initiate(vi,cor);
                }
            }
            this.complete(true);
        }

    }do{
        this.setState(State.DISABLED);
        let completed:Boolean = false;
        while (! completed){
            accept(SignalStart){
                this.enable();
            }
            or accept(SignalLinkActivited){
                completed = this.running();
            }
            or accept(SignalTerminate){
                this.terminate();
                completed = true;
            }
        }
    }

}

public active class TerminateExecution specializes Execution{

    public doAction(){
        this.setState(State.RUNNING);
        this.setState(State.TERMINATED);
        //send parent
        this.parentExecution.SignalTerminate();
    }
}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (! completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivited){
            this.running();
        }
    }
}

```

```

        completed = true;
    }
    or accept(SignalTerminate){
        this.terminate();
        completed = true;
    }
}

public abstract active class MessageReceiver{
    public receive signal SignalReceive{
//        public message:Message;
        public info:MessageInfo;
    }
}do{}

public active class InvokeExecution specializes Execution,
MessageReceiver{
    public getInvoke():Invoke{
        return (Invoke)this.activity_;
    }

    public running():Boolean{
        //change the state to running if possible
        //send the input message to external by messageManagerInstance
        let completed:Boolean = false;
        if (this.state==State.ENABLED && this.allLinksActivited()){
            let linkSucc:Boolean = this.popagateDPE();
            if (linkSucc){
                completed = this.runningLinkSucc();
            }else{
                this.complete(false);
            }
        }
        return completed;
    }

    private initCorrelation(){
        for (Correlation cor:this.getInvoke().correlations){
            if (cor.initiate && (cor.pattern=="in" ||
cor.pattern=="in/out")){
                let
vi:VariableInstance=instance(this).findVariableInstance(this.getInvoke().
variable);
                instance(this).correlationManager.initiate(vi,cor);
            }
        }
    }

    private runningLinkSucc():Boolean{
        this.setState(State.RUNNING);
        let inputMessage:Message =
this.getInvoke().inputVariable.getMessage();
        let messageInfo:MessageInfo = new MessageInfo();
        messageInfo.message = inputMessage;
        messageInfo.operation = this.getInvoke().operation;
        messageInfo.portType = this.getInvoke().portType;
        messageManagerInstance().SignalSendMessage(this,messageInfo);

        //init Correlation instances

```

```

        if (!this.getInvoke().correlations->isEmpty()){
            this.initCorrelation();
        }
        //if the invoke expected responds, then do not complete, wait
        for incoming message
        //else complete the execution
        if (this.getInvoke().outputVariable==null){
            this.complete(true);
            return true;
        }else{
            //require message
            let info:MessageInfo = new MessageInfo();
            let outputMessage:Message =
this.getInvoke().outputVariable.getMessage();

            info.portType = this.getInvoke().portType;
            info.operation = this.getInvoke().operation;
            info.message = outputMessage;

            messageManagerInstance().SignalRequestMessage(this, info);

            return false;
        }
    }

    private isOutputMessage(in info:MessageInfo):Boolean{
        let outputMessageType:MessageType =
this.getInvoke().outputVariable.messageType;
        return info.message.type==outputMessageType;
    }

    private isFaultMessage(in info:MessageInfo):Boolean{
        let operation:Operation = this.getInvoke().operation;
        if (operation.fault==null){
            return false;
        }else {
            return operation.fault.messageType==info.message.type;
        }
    }

    private receivingNormalFlow(in info:MessageInfo){
        //then assign the message value to the output message
        this.getInvoke().outputVariable.setVariable(info.message);

        instance(this).findVariableInstance(this.getInvoke().outputVariable)
            .copyMessage(info.message);
        if (!this.getInvoke().correlations->isEmpty()){
            // this.getInvoke().correlations
            // ->select cor ()
            for (Correlation cor:this.getInvoke().correlations){
                if (cor.initiate && (cor.pattern=="out" ||
cor.pattern=="in/out")){
                    let vi:VariableInstance=

instance(this).findVariableInstance(this.getInvoke().variable);
                    instance(this).correlationManager.initiate(vi,cor);
                }
            }
        }
        this.complete(true);
    }
}

```

```

    public receiving(in info:MessageInfo){
        //if the message type is the output message
        if (this.isOutputMessage(info)){
            this.receivingNormalFlow(info);
        }
        //else if the message type is the fault message
        else if (this.isFaultMessage(info)){
            this.setState(State.FAULTED);
            let faultMessage:Message = info.message;
            let faultInfo:FaultInfo = new FaultInfo();
            faultInfo.faultName = faultMessage.type.name;
            faultInfo.faultMessage = faultMessage;
            this.throwFault(faultInfo);
        }
        //then report fault this.throwFault
        //set state to FAULTED
        //TODO else the execution received a wrong message, possibly a
        runtime fault will be thrown.
    }

    }do{
        this.setState(State.DISABLED);
        let completed:Boolean = false;
        while (! completed){
            accept(SignalStart){
                this.enable();
            }
            or accept(SignalLinkActivated){
                completed = this.running();
            }
            or accept(sig:SignalReceive){
                //if there is no faults
                this.receiving(sig.info);
                completed = true;
            }
            or accept(SignalTerminate){
                this.terminate();
                completed = true;
            }
        }
    }

}

public active class ReceiveExecution specializes Execution,
MessageReceiver{
    public getReceive():Receive{
        return (Receive)this.activity_;
    }

    public doAction(){
        this.setState(State.RUNNING);
        let info:MessageInfo = new MessageInfo();
        let message:Message = new Message();
        message.type = this.getReceive().operation.input.messageType;

        info.portType = this.getReceive().portType;
        info.operation = this.getReceive().operation;
        info.message = message;
        messageManagerInstance().SignalRequestMessage(this, info);
    }
}

```



```

        private initCorrelation(){
            for (Correlation cor: this.getReceive().correlations){
                if (cor.initiate){
                    let
vi:VariableInstance=instance(this).findVariableInstance(this.getReceive().
.variable);
                    instance(this).correlationManager.initiate(vi,cor);
                }
            }
        }

        public receiving(in info:MessageInfo):Boolean{

            //if it is a normal receive
            //copy the data from the message to the variable
            //if receive an fault, the standard seems not defined.
            //
            if
            (this.getReceive().operation.input.messageType==message.type){
            //
            this.getReceive().variable.setVariable(info.message);
            instance(this).findVariableInstance(this.getReceive().variable)
                .copyMessage(info.message);
            //inform locus to create another new instance that wait for the
            message.
            if (this.getReceive().createInstance){
                locus().SignalCreateInstance();
            }
            //init correlation
            if (!this.getReceive().correlations->isEmpty()){
                this.initCorrelation();
            }
            this.complete(true);

            return true;
        }
    }

}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (! completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivited){
            completed = this.running();
        }
        or accept(sig:SignalReceive){
            //if there is no faults
            completed = this.receiving(sig.info);
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}

}

public active class ThrowExecution specializes Execution{

    public getThrow():Throw{
        return (Throw)this.activity_;
    }
}

```

```

    }

    public doAction(){
        this.setState(State.RUNNING);
        //      let scopeExecution:ScopeExecution =
        //          (ScopeExecution)this.activity_.enclosedScope().execution;
        //      scopeExecution.SignalFaulted(this.getThrow().faultName,
        null);//
        let faultInfo:FaultInfo = new FaultInfo();
        faultInfo.faultName = this.getThrow().faultName;
        faultInfo.faultVariable = this.getThrow().faultVariable;
        this.throwFault(faultInfo);
        //the normal flow ends
        //@parallel{
            this.setState(State.FAULTED);
            //send parent
            //send link targets
            this.parentExecution.SignalChildFinished(this);
        }
    }

}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (! completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivited){
            this.running();
            completed = true;
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}

}

public active class WaitExecution specializes Execution{
    public getWait():Wait{
        return (Wait)this.activity_;
    }

    public doAction(){
        this.setState(State.RUNNING);
        if (this.getWait().for_!=null){
            waitFor(this.getWait().for_);
        }
        else{
            waitUntil(this.getWait().until_);
        }
        this.complete(true);
    }
}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (! completed){
        accept(SignalStart){
            this.enable();
        }
    }
}

```

```

        or accept(SignalLinkActivited){
            this.running();
            completed = true;
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}

public assoc StructuredExecution_Execution{
    public parentExecution:StructuredExecution;
    public childExecution:compose Execution[*];
}

public abstract active class StructuredExecution specializes
Execution{
    public receive signal SignalChildFinished{
        public execution:Execution;
    }

    //@OCL("
self.childExecution->forall(e|e.state=State::COMPLETED)
or self.childExecution->exists(e|e.state=State::FAULTED)
")
    public allChildFinished():Boolean{
        return this.childExecution->forall e(e.state==State.COMPLETED);
    }
    //send SignalCompleted to its container
    //send SignalTerminate to its childExecution

    public terminateChilds(){
        for (Execution execution:this.childExecution){
            execution.SignalTerminate();
        }
    }

    public terminate(){
        this.terminateChilds();
        this.parentExecution.SignalTerminate();
        this.setState(State.TERMINATED);
    }
}do{}

public class LinkInstance {
    public link:Link;
    public isDetermined:Boolean;
    public success:Boolean;
    public sourceDetermined(in isNeg:Boolean){
        //TODO transitionCondition
        this.isDetermined = true;
        this.success = isNeg;
        let targetActivity:Activity = this.link.target;
        let targetExecution:Execution =
locus().findExecution(instance(this.flowExecution),targetActivity);
        if (targetExecution!=null){
            targetExecution.SignalLinkActivited();
        }
    }
}

```

```

public assoc FlowExecution_LinkInstance{
    public links:compose LinkInstance[*];
    public flowExecution:FlowExecution;
}

public active class FlowExecution specializes StructuredExecution{

    public linkInstanceOf(in l:Link):LinkInstance{
//      this.links
//      ->select linkInstance (linkInstance==l)
//      ->any();
        for (LinkInstance linkInstance:this.links){
            if (linkInstance.link==l){
                return linkInstance;
            }
        }
        return null;
    }

    public getFlow():Flow{
        return (Flow)this.activity_;
    }

    public createExecutions(){
        for (Link l:this.getFlow().links){
            let linkInstance:LinkInstance = new LinkInstance();
            linkInstance.link=l;
            linkInstance.isDetermined = false;
            linkInstance.success = false;
            linkInstance.flowExecution = this;
        }

        for (Activity act: ((Flow)this.activity_).activities){
            let execution:Execution = createExecution(act);
            this.childExecution->add(execution);
            //start execution in java
        }
    }

    public doAction(){
        this.setState(State.RUNNING);
        this.createExecutions();
        this.childExecution.SignalStart();
    }

    public finishing(in execution:Execution):Boolean{
        let completed:Boolean = false;
        if (execution.state==State.FAULTED){
            this.setState(State.FAULTED);
            //inform parent execution
            this.terminateChilds();
            this.parentExecution.SignalChildFinished(this);
            return true;
        }

        if (this.allChildFinished() && this.state==State.RUNNING){
            completed = true;
            this.complete(true);
        }
    }
}

```

```

        return completed;
    }
}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (!completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivated){
            completed = this.running();
        }
        or accept(sig:SignalChildFinished){
            completed = this.finishing(sig.execution);
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}

public active class SequenceExecution specializes StructuredExecution{

    public getSequence():Sequence{
        return (Sequence)this.activity_;
    }

    public doAction(){
        this.setState(State.RUNNING);
        let execution:Execution =
createExecution(this.getSequence().activities[0]);
        execution.parentExecution = this;
        execution.SignalStart();
    }

    public finishing(in execution:Execution):Boolean{
        if (execution.state==State.FAULTED){
            this.setState(State.FAULTED);
            //inform parent execution
            this.parentExecution.SignalChildFinished(this);
            return true;
        }
        else if (this.childExecution-
>size()==this.getSequence().activities->size()
            && this.allChildFinished() && this.state==State.RUNNING){
            this.complete(true);
            return true;
        }
        else{
            let preIndex:Integer = this.getSequence().activities-
>indexOf(execution.activity_);
            let nextActivity:Activity=
this.getSequence().activities[preIndex+1];
            let childExecution:Execution = createExecution(nextActivity);
            childExecution.parentExecution = this;
            childExecution.SignalStart();
            return false;
        }
    }
} //finishing

```

```

}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (!completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivated){
            completed = this.running();
        }
        or accept(sig:SignalChildFinished){
            completed = this.finishing(sig.execution);
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}

public active class SwitchExecution specializes StructuredExecution{

    public getSwitch():Switch{
        return (Switch)this.activity_;
    }

    private createOtherwise():Execution{
        let execution:Execution = null;
        if (this.getSwitch().otherwise==null){
            //create an empty
            let em:Empty = new Empty();
            em.name = "default otherwise";
            execution = createExecution(em);
        }else{
            execution = createExecution(this.getSwitch().otherwise);
        }
        return execution;
    }

    private informNegativeLink(in cs:Case, in execution:Execution){
        if (cs.activity_!=execution.activity_){
            for (Link link:cs.activity_.targets){
                let flow:Flow = link.flow;
                let flowExecution:FlowExecution =
                (FlowExecution)locus().findExecution(instance(this),flow);
                let linkInstance:LinkInstance =
                flowExecution.linkInstanceOf(link);
                linkInstance.sourceDetermined(false);
            }
        }
    }

    public doAction(){
        this.setState(State.RUNNING);
        //select the first case which condition evaluates to true
        let execution:Execution = null;
        for (Case c:this.getSwitch().case_){
            if (c.condition.eval()){
                execution = createExecution(c.activity_);
                break;
            }
        }
    }
}

```

```

    }
}
//if no case eval to true
if (execution==null){
    execution = this.createOtherwise();
}

execution.parentExecution = this;
execution.SignalStart();
//When cs.activity_ is source to links, and if its condition
evaluates to false,
//all the target executions must be informed that the link is a
negative link
for (Case cs:this.getSwitch().case_){
    this.informNegativeLink(cs,execution);
}
//TODO otherwise
}
public finishing(in execution:Execution){
    if (this.state==State.RUNNING &&
execution.state==State.COMPLETED){
        this.setState(State.COMPLETED);
        //inform parent execution
        this.parentExecution.SignalChildFinished(this);
        //if there is any links, send link finish to the target of
the link
        if (! this.activity_.targets->isEmpty()){
            this.notifyTargets(true);
        }
    } else if (execution.state==State.FAULTED){
        this.setState(State.FAULTED);
        //inform parent execution
        this.parentExecution.SignalChildFinished(this);
    }
}

}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (!completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivated){
            this.running();
        }
        or accept(sig:SignalChildFinished){
            this.finishing(sig.execution);
            completed = true;
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}

public active class PickExecution specializes StructuredExecution,
MessageReceiver{

    public getPick():Pick{

```

```

        return (Pick)this.activity_;
    }

    public doAction(){
        this.setState(State.RUNNING);

//        for (OnMessage onmsg:this.getPick().onMessage){
//            for (AbstractOnEvent onevent:this.getPick().onEvent){
//                //request all the message
//                if (onevent instanceof OnMessage){
//                    let onmsg:OnMessage = (OnMessage)onevent;
//                    let info:MessageInfo = new MessageInfo();
//                    let message:Message = new Message();
//                    message.type = onmsg.operation.input.messageType;
//                    info.portType = onmsg.portType;
//                    info.operation = onmsg.operation;
//                    info.message = message;
//                    messageManagerInstance().SignalRequestMessage(this, info);
//                }
//            }
//        }

        private createExecutionAndInformNegative(in onmsg:OnMessage, in
messageInfo:MessageInfo){
            if (messageInfo.message.type==onmsg.operation.input.messageType
                && messageInfo.portType==onmsg.portType
                && messageInfo.operation==onmsg.operation )
            {
                let execution:Execution = createExecution(onmsg.activity_);
                execution.parentExecution = this;
                execution.SignalStart();
            }else{
                for (Link link: onmsg.activity_.targets){
                    let flow:Flow = link.flow;
                    let flowExecution:FlowExecution =
                        (FlowExecution)locus().findExecution(instance(this),flow);
                    let linkInstance:LinkInstance =
                        flowExecution.linkInstanceOf(link);
                    linkInstance.sourceDetermined(false);
                }
            }
        }

        public receiving(in messageInfo:MessageInfo){
            //accroding to the messageinfo, find the OnMessage, execute the
            activity

            for (AbstractOnEvent onevent:this.getPick().onEvent){
                if (onevent instanceof OnMessage){

                    this.createExecutionAndInformNegative((OnMessage)onevent,messageInfo);
                }
            }
        }

        public finishing(in execution:Execution){
            if (this.state==State.RUNNING &&
                execution.state==State.COMPLETED){
                this.complete(true);
            }else if (execution.state==State.FAULTED){

```



```

        this.setState(State.FAULTED);
        //inform parent execution
        this.parentExecution.SignalChildFinished(this);
    }
}

}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    let firstOccured:Boolean = false;
    while (!completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivited){
            completed = this.running();
        }
        or accept(sigReceive:SignalReceive){
            if (! firstOccured){
                this.receiving(sigReceive.info);
                firstOccured = true;
            }
        }
        or accept(sig:SignalChildFinished){
            this.finishing(sig.execution);
            completed = true;
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}

public active class WhileExecution specializes StructuredExecution{

    public getWhile():While{
        return (While)this.activity_;
    }

    public doAction(){
        this.setState(State.RUNNING);
        if (this.getWhile().condition.eval()){
            this.runWhile();
        }else{
            this.complete(true);
        }
    }

    public runWhile(){

        let execution:Execution =
createExecution(this.getWhile().primaryActivity());
        execution.parentExecution = this;
        execution.SignalStart();
    }

    public finishing(in execution:Execution):Boolean{
        if (this.state==State.RUNNING &&
execution.state==State.COMPLETED){
            if (this.getWhile().condition.eval()){

```

```

        this.runWhile();
        return false;
    }
    else{
        this.complete(true);
        return true;
    }
}
else{
    this.setState(State.FAULTED);
    //inform parent execution
    this.parentExecution.SignalChildFinished(this);
    return true;
}
}
}

}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (!completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivited){
            completed = this.running();
            completed = completed || (! this.getWhile().condition.eval());
        }
        or accept(sig:SignalChildFinished){
            completed = this.finishing(sig.execution);
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}

}

public class FaultInfo{
    public faultName:String;
    public faultMessage:Message;
    public faultVariable:Variable;
}

public assoc Variable_VariableInstance{
    public variable:Variable;
    public instance:VariableInstance[*];
}

public class VariableInstance{
    public value:String;
    public copyMessage(in message:Message){
        this.value = message.value;
        log(this);
    }
}

public class CompensationStack{

    public push(inout exe:ScopeExecution){
        /*@inline( java )"this.stack.push(exe);" */
    }
    public pop():ScopeExecution{

```

```

        return null;
        /*@inline(java)"
        if (stack.empty())
            return null;
        else
            return this.stack.pop();" */
    }
}

public class PortInstance{
    public operation:Operation;
}

public active class ScopeExecution specializes StructuredExecution,
MessageReceiver{
    public receive signal SignalFaulted{
        public faultInfo:FaultInfo;
    }

    public receive signal SignalCompensateScope{}

    public receive signal SignalCompensate{
        public compensateExecution:CompensateExecution;
    }

    public receive signal SignalCompleted{}

    private compensateExecution:CompensateExecution;
    public correlationManager:compose CorrelationManager;
    public variableInstances:compose VariableInstance[*];
    public portInstances:compose PortInstance[*];
    public compensationStack:compose CompensationStack;
    public getScope():Scope{
        return (Scope)this.activity_;
    }

    public getFaultHandler():FaultHandler{
        return this.getScope().faultHandler;
    }

    public findFaultActivity(in name:String):Activity{
        if (this.getFaultHandler()==null){
            return null;
        }

        for (Catch catch_:this.getFaultHandler().catch_){
            if (catch_.faultName==name){
                return catch_.activity_;
            }
        }
        if (this.getFaultHandler().catchAll!=null){
            return this.getFaultHandler().catchAll.activity_;
        }else{
            return null;
        }
    }

    public findVariableInstance(in variable:Variable):VariableInstance{
        //instance = this.variableInstances->select e
        (e.variable==variable)->first();
        let instance:VariableInstance = null;

```

```

        for (VariableInstance ins: this.variableInstances){
            if (ins.variable==variable){
                instance = ins;
                break;
            }
        }
        if (instance==null){
            instance = new VariableInstance();
            instance.variable = variable;
        }
        this.variableInstances->add(instance);
        return instance;
    }

    public doAction(){

        this.setState(State.RUNNING);
        this.compensationStack = new CompensationStack();
        this.correlationManager = new CorrelationManager();
        initialisePartnerLinks(this);
        //TODO install on alarm handlers
        let execution:Execution =
createExecution(this.getScope().primaryActivity());
        execution.parentExecution = this;
        execution.SignalStart();
        //install event handler
        //if event handler exists
        if (this.getScope().eventHandlers!=null){
            for (OnEvent event : this.getScope().eventHandlers.events){
                let info:MessageInfo = new MessageInfo();
                let message:Message = new Message();
                message.type = event.messageType;
                info.message = message;
                info.operation = event.operation;
                info.portType = event.portType;
                messageManagerInstance().SignalRequestMessage(this, info);
            }
        }

    }

    public finishing(in execution:Execution){
        //do something with the event handlers
        if (this.state==State.RUNNING &&
execution.state==State.COMPLETED){
            this.complete(true);
            //terminate all fault handlers
            //install compensation handler, put it to the stack
            let stack:CompensationStack =
instance(this).compensationStack;
            stack.push(this);
        }
        else if (this.state==State.RUNNING &&
execution.state==State.FAULTED){
            this.setState(State.FAULTED);
            //inform parent execution
            this.parentExecution.SignalChildFinished(this);
        }
        else if (this.state==State.FAULTED){
            //the fault activity finished
            this.setState(State.COMPLETED);
            //inform parent execution

```

```

        this.parentExecution.SignalChildFinished(this);
        //if there is any links, send link finish to the target of
the link
        if (! this.activity_.targets->isEmpty()){
            this.notifyTargets(true);
        }
    }

    public faulted(in faultInfo:FaultInfo){
        // discuss: if fault is threw by normal flow, or negative flow.
        if (this.state==State.RUNNING ){
            //terminate all childs and event handlers
            this.setState(State.FAULTED);
            this.terminateChilds();
            // this.faultHandlerExecution.SignalFault(name,variable);
            let faultActivity:Activity =
this.findFaultActivity(faultInfo.faultName);
            if (faultActivity==null){
                //report the fault to its upper scope
                let parentScopeExecution: ScopeExecution =
(ScopeExecution)locus().findExecution(instance(this),this.getScope().encl
osedScope());
                //
                (ScopeExecution)this.getScope().enclosedScope().execution;
                parentScopeExecution.SignalFaulted(faultInfo);
            }
            else{
                // create the instance of the activity in the fault
handler.
                let faultExecution:Execution =
createExecution(faultActivity);
                faultExecution.parentExecution = this;
                faultExecution.SignalStart();
            }
        }
        else if (this.state==State.FAULTED){
            //the fault is thrown by negative flow
            //the parent scope of the scope deal with the fault
            let scopeExecution:ScopeExecution
            =
(ScopeExecution)this.getScope().enclosedScope().execution;
            scopeExecution.SignalFaulted(faultInfo);
        }
    }

    public handleEvent(in info:MessageInfo){
        let event: OnEvent = this.findEventFromInfo(info);
        let execution:Execution = createExecution(event.scope);
        execution.parentExecution = this;
        execution.SignalStart();
        execution.SignalLinkActivited();
    }

    //OCL("self.getScope().eventHandlers.events
    // ->select(e|e.messageType=info.message.type and
e.operation=info.operation)->first()")
    public findEventFromInfo(in info:MessageInfo):OnEvent{
        for (OnEvent event:this.getScope().eventHandlers.events){
            if (event.operation==info.operation){
                return event;
            }
        }
    }

```

```

    }
}
return null;
}

public isFaultHandlerActivity(in act:Activity):Boolean{
    let result:Boolean = false;
    if (this.getFaultHandler()==null){
        return false;
    }
    for (Catch a:this.getFaultHandler().catch_){
        if (a.activity_==act){
            result=true;
            break;
        }
    }
    if (this.getFaultHandler().catchAll.activity_==act)
    {
        result = true;
    }
    return result;
}

public compensateScope(){
    if (this.state==State.COMPLETED){
        if (this.getScope().compensationHandler!=null){
            this.setState(State.COMPENSATING);
            let execution:Execution =
createExecution(this.getScope().compensationHandler.activity_);
            execution.parentExecution = this;
            execution.SignalStart();
        }
    }
}

public childFinish(in execution:Execution):Boolean{
    let completed:Boolean = false;
    //Only when the finished execution is the execution in of its
sub activity.

    //if the child is the primary activity of the scope, it finish
successfully or not
    if (execution.activity_== this.getScope().primaryActivity()
        && (execution.state==State.COMPLETED ||
execution.state==State.FAULTED)
    ){
        this.finishing(execution);
    }
    else if (this.state==State.COMPENSATING){
        this.setState(State.STOPPED);
        if (this.compensateExecution!=null){
            this.compensateExecution.SignalCompensateNext();
        }
        completed = true;
    }
    //if the execution is a successfully finished fault handler.
    else if (execution.state==State.COMPLETED
        && this.isFaultHandlerActivity(execution.activity_)
    ){
        this.setState(State.FAULTED);
        this.parentExecution.SignalChildFinished(this);
    }
}

```

```

        completed = true;
    }
    return completed;
}

}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (!completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivated){
            this.running();
        }
        or accept(sigFinish:SignalChildFinished){
            completed = this.childFinish(sigFinish.execution);
        }
        or accept(sigReceive:SignalReceive){
            this.handleEvent(sigReceive.info);
        }
        or accept(sigFault:SignalFaulted){
            this.faulted(sigFault.faultInfo);
        }
        or accept(sigOnAlarm:SignalAlarm){
            //start on alarm activity execution
        }
        or accept(sigCompensate:SignalCompensate){
            this.compensateExecution = sigCompensate.compensateExecution;
            this.compensateScope();
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
        or accept(SignalCompleted){
            completed = true;
        }
        // accept
    }//while
}

```

```

public active class CompensateExecution specializes Execution{

    public receive signal SignalCompensateNext{}

    public getCompensate():Compensate{
        return (Compensate)this.activity_;
    }

    public running():Boolean{
        //execute the scope's compensation handler's activity
        //and execute all the scope's sub-scope compensation handler
execute
        //only compensate completed scopes[standard p112]
        this.setState(State.RUNNING);
        this.compensateNext();
        return false;
    }
}

```

```

    public compensateNext():Boolean{
        let stack:CompensationStack = instance(this).compensationStack;
        let scopeExecution:ScopeExecution = stack.pop();
        if (scopeExecution!=null){
            scopeExecution.SignalCompensate(this);
            return false;
        }
        return true;
    }
}do{
    this.setState(State.DISABLED);
    let completed:Boolean = false;
    while (! completed){
        accept(SignalStart){
            this.enable();
        }
        or accept(SignalLinkActivited){
            this.running();
        }
        or accept(SignalCompensateNext){
            completed = this.compensateNext();
        }
        or accept(SignalTerminate){
            this.terminate();
            completed = true;
        }
    }
}
}

public active class Locus specializes StructuredExecution,
MessageReceiver{
    public process_:compose BPELProcess;
    public messageManager:compose MessageManager;
    public testReceivedMessage:compose MessageInfo[*];
    public wsdl:compose WSDL;
    public createMultipleInstance:Boolean = false;

    public receive signal SignalCreateInstance{}

    public getAllScopes():ScopeExecution[*]{
        return getAllElementsInCompensateStack();
    }

    public findExecution(in instance:ScopeExecution, in
act:Activity):Execution{
        return allContents(instance)
            ->select e (e instanceof Execution)
            ->select e (e.activity_==act)->first();
        /*@inline( java) "
        if (instance.getActivity_().equals(act)){
            return instance;
        }
        TreeIterator<EObject> iterator = instance.eAllContents();
        while (iterator.hasNext()){
            EObject eobj = iterator.next();
            if (eobj instanceof Execution &&
((Execution)eobj).getActivity_().equals(act)){
                return (Execution)eobj;
            }
        }
        */
    }
}

```



```

        GlobalActivity.log("\\\\"[LocusImpl::findExecution] ERROR: didnot
find\\\\"");
        return null;" */
    }

//    public findLinkStatus(in instance:ScopeExecution, in
link:Link):LinkInstance{
//
//    }

    public createInstanceMessageInfo():MessageInfo[*]{
        let infos:MessageInfo[] = new MessageInfo();

        /*@inline (java) "
EList<MessageInfo> infos = new BasicEList<MessageInfo>() ;" */

        for (Receive re:this.onCreateInstanceReceive()){
            let message:Message = new Message();
            message.type = re.operation.input.messageType;
            let messageInfo:MessageInfo = new MessageInfo();
            messageInfo.message = message;
            messageInfo.operation = re.operation;
            messageInfo.portType = re.portType;
            infos->add(messageInfo);
        }
        return infos;
    }

    /*@OCL("Receive.allInstances()->select(e|e.createInstance)-
>asOrderedSet()")
    public onCreateInstanceReceive():Receive[*]{
        /*@inline (java) "
TreeIterator<EObject> iterator =
this.getProcess_().eAllContents();
EList<Receive> receives = new BasicEList<Receive>();
while (iterator.hasNext()){
    EObject eobj = iterator.next();
    if (eobj instanceof Receive &&
((Receive)eobj).getCreateInstance()){
        receives.add((Receive)eobj);
    }
}
return receives;" */
        return null;
    }

    private createInstance(){
        let execution:Execution = createExecution (this.process_);
        execution.parentExecution = this;
        execution.SignalStart();
    }
}do{

    this.messageManager = new MessageManager();

    /*@inline (java) "
GlobalActivity.setGlobalMessageManager(this.messageManager);
GlobalActivity.setLocus(this);" */

    for (MessageInfo info:this.testReceivedMessage){

```

```

        this.messageManager.SignalReceiveMessage(info);
    }

    this.SignalCreateInstance();
    if (this.createMultipleInstance){
        let completed:Boolean = false;
        while (! completed){
            accept(SignalCreateInstance){
                this.createInstance();
            }
            or accept (sigChild:SignalChildFinished){
                //TODO compensation is not correct
                for (ScopeExecution exe:this.getAllScopes()){
                    exe.SignalCompleted();
                }
            }
        }
    }
    else{
        accept(SignalCreateInstance){
            let execution:Execution = createExecution (this.process_);
            execution.parentExecution = this;
            execution.SignalStart();
        }
        accept (SignalChildFinished){
            //TODO compensation is not correct
            for (ScopeExecution exec:this.getAllScopes()){
                exec.SignalCompleted();
            }
        }
    }
}

}

public class MessageInfo{
    public message:compose Message;
    public portType:PortType;
    public operation:Operation;
}

public class CorrelationManager{
    public instances:compose CorrelationInstance[*];

    public initiate(in variable:VariableInstance, in
correlation:Correlation){
        let instance:CorrelationInstance =
this.createInstance(variable,correlation);
        this.instances->add(instance);
        log("Correlation initialised: "+variable.value);
    }

    public createInstance(in variable:VariableInstance, in
correlation:Correlation):CorrelationInstance{
        let correlationInstance:CorrelationInstance = new
CorrelationInstance();
        correlationInstance.correlation = correlation;
        correlationInstance.value = variable.value;
        return correlationInstance;
    }
}
}

```

```

public assoc Correlation_CorrelationInstance{
    public correlation:Correlation;
    public instance:CorrelationInstance;
}

public class CorrelationInstance{
    public value:String;
    public correlationSatisfied(in messageInfo:MessageInfo):Boolean{
        return this.value==messageInfo.message.value;
        /*@inline( java )"return
this.getValue().equals(messageInfo.getMessage().getValue());" */
    }
}

public active class MessageManager{
    public receive signal SignalReceiveMessage{
        public messageInfo:MessageInfo;
    }
    public receive signal SignalRequestMessage{
        public execution:Execution;
        public messageInfo:MessageInfo;
    }
    public receive signal SignalSendMessage{
        public execution:Execution;
        public messageInfo:MessageInfo;
    }
    public receive signal SignalTerminateMessageManager{}

    public messageRequests: compose MessageRequest[*];
    public incomingMessage: compose MessageInfo[*];

    public findMessageRequest(in
messageInfo:MessageInfo):MessageRequest{
        for (MessageRequest request:this.messageRequests){
            if (request.operation==messageInfo.operation
                && this.correlationSatisfied(request.execution,messageInfo)
            ){

                // if
                (instance(request.execution).correlationManager.instances
                //      ->exists e (e.correlationSatisfied(messageInfo)))
                //      return request;
                return request;
            }
        }
        return null;
    }

    public correlationSatisfied(in execution:Execution, in
messageInfo:MessageInfo):Boolean{
        if (execution instanceof ReceiveExecution
            &&
            ((ReceiveExecution)execution).getReceive().createInstance){
            return true;
        }

        let correlationDefined:Boolean = false;

        for (CorrelationInstance
ins:instance(execution).correlationManager.instances){

```

```

        correlationDefined = true;
        if (ins.correlationSatisfied(messageInfo)){
            return true;
        }
    }
    return !correlationDefined;
}

//("self.incomingMessage
// ->select(e|e.portType=info.portType and
e.operation=info.operation)->first()")
public findMessageInfo(in info:MessageInfo):MessageInfo{
    for (MessageInfo e: this.incomingMessage){
        if (e.operation==info.operation){
            return e;
        }
    }
    return null;
}

public initialise(){
    //initialise messageRequests and incoming message;
    /*@ inline(java)
        "this.incomingMessage = new BasicELList<MessageInfo>();
        this.messageRequests = new BasicELList<MessageRequest>();
        this.messageQueue = new BasicELList<Signal_>();"
    */
}

public sendToExecution(in request:MessageRequest,in
messageInfo:MessageInfo){
    let execution:Execution = request.execution;
    if (execution instanceof ReceiveExecution){
        let receiveExecution:ReceiveExecution =
(ReceiveExecution)execution;
        receiveExecution.SignalReceive(messageInfo);
    } else if (execution instanceof ScopeExecution){
        let scopeExecution:ScopeExecution = (ScopeExecution)execution;
        scopeExecution.SignalReceive(messageInfo);
    }
    else if (execution instanceof InvokeExecution){
        let invokeExecution:InvokeExecution =
(InvokeExecution)execution;
        invokeExecution.SignalReceive(messageInfo);
    }
    else if (execution instanceof PickExecution){
        let pickExecution:PickExecution = (PickExecution)execution;
        pickExecution.SignalReceive(messageInfo);
    }
    else if (execution instanceof Locus){
        let locus:Locus = (Locus)execution;
        locus.SignalReceive(messageInfo);
    }
    else {
        log( "[MessageManager.sendToExecution]" + execution + "does not
implemented" );
    }
    this.incomingMessage->remove(messageInfo);
    this.messageRequests->remove(request);
    log( "External message:" + messageInfo.message + " send to " +
request.execution);
}

```

```

    }

    private receiveMessage(in messageInfo:MessageInfo){
        //receive message from external
        log("External message received:"+messageInfo.message);
        let request:MessageRequest=this.findMessageRequest(messageInfo);
        if (request==null){
            this.incomingMessage->add(messageInfo);
        }else{
            this.sendToExecution(request, messageInfo);
        }
    }
}

    private requestMessage(in execution:Execution, in
messageInfo:MessageInfo){
        //internal execution request message
        let info:MessageInfo = this.findMessageInfo(messageInfo);
        let request:MessageRequest = new MessageRequest
            (execution,
             messageInfo.message.type,
             messageInfo.portType,
             messageInfo.operation
            );
        request.init(execution,
                     messageInfo.message.type,
                     messageInfo.portType,
                     messageInfo.operation
                    );

        if (info==null || !
(this.correlationSatisfied(execution,info))){
            this.messageRequests->add(request);
        }else{
            this.sendToExecution(request,info);
        }
        log("Request received from "+request.execution);
    }
}do{
    this.initialise();

    let completed:Boolean = false;
    while (!completed){
        accept(sigReceive:SignalReceiveMessage){
            this.receiveMessage(sigReceive.messageInfo);
        }
        or accept (sigRequest:SignalRequestMessage){
            this.requestMessage(sigRequest.execution,
sigRequest.messageInfo);
        }
        or accept(sigSend:SignalSendMessage){
            //internal execution sends message to external
            //send to a phsycal address if

            sigSend.messageInfo.operation.invoke(sigSend.messageInfo.message);

            sendToPortInstance(instance(sigSend.execution),sigSend.messageInfo);
        }
        or accept(SignalTerminateMessageManager){
            completed = true;
        }
    }
}

```

```

    }
}

public class MessageRequest{
    public execution:Execution;
    public messageType:MessageType;
    public portType:PortType;
    public operation:Operation;

    @Create public MessageRequest(in e:Execution, in m:MessageType, in
p:PortType, in o:Operation){
        this.execution = e;
        this.messageType = m;
        this.portType = p;
        this.operation = o;
    }
    public init(in e:Execution, in m:MessageType, in p:PortType, in
o:Operation){
        this.execution = e;
        this.messageType = m;
        this.portType = p;
        this.operation = o;
    }
}

/*GLOBAL ACTIVITIES */

public activity createExecution(in act:Activity):Execution{
    let execution:Execution = null;
    if (act instanceof Empty){
        execution = new EmptyExecution();
    }
    else if (act instanceof Receive){
        execution = new ReceiveExecution();
    }
    //...
    execution.activity_ = act;
    return execution;
}

public activity log(in message:String){
}

public activity messageManagerInstance():MessageManager{
    return MessageManager.allInstances()[0];
}

public activity correlationManager():CorrelationManager{
}

public activity pushCompensateStack(in scopeExecution:ScopeExecution){
}

public activity popCompensateStack():ScopeExecution{
}

public activity getAllElementsInCompensateStack():ScopeExecution[*]{
}

```

```

public activity emptyCompensateStack(){}

public activity nextCompensation(){
}

public activity container(in model:any){}

public activity waitFor(in duration:DurationExpression){}
public activity waitUntil(in deadline:DeadlineExpression){}
public activity locus():Locus{
    return Locus.allInstances()->first();
}
public activity allContents(in ob:any):any{}

public activity instance(in execution:Execution):ScopeExecution{}

public activity logVariable(in vi:VariableInstance){}
public activity initialisePartnerLinks(inout
scopeExecution:ScopeExecution){}

    public activity sendToPortInstance(inout instance:ScopeExecution,in
info:MessageInfo){}

```

Appendix B. Guidelines for the FQLS process

Decide the semantic definition strategy

- The language engineers need to ask, what is the primary goal of the specification? Who will be the primary reader of the specification? By clarifying this, the language engineers will have these goals in mind while designing the language.
- How to deal with semantic variation points? Will the specification contain semantics that are intentionally left undefined? If so, they can be defined as an abstract activity. Such an activity defines an interface that specifies the input and output, but leaves the implementation details to a later stage.
- How to deal with the semantic variation points of fUML? fUML explicitly defines concurrency, time and inter-object communication as semantic variation points. This will not be a problem if such concepts are not involved in the language specification. Depending on how much detail the language specification needs to define, a language containing such concepts is not excluded in our approach. As mentioned, fUML's semantic variation points do not give the implementation details of these concepts, but many languages do not need to know how these concepts are implemented. For example, BPEL contains Flow activity, which executes its enclosed activities concurrently. The concurrency in BPEL has not been explicitly defined either, which means it does not constrain the concurrent execution model, which can be real concurrency in a multi-core machine, or it can just use a single core to schedule concurrency. Therefore, it can use the concepts in ALF to represent its concurrent concepts, and considers the implementation of concurrency as a low-level detail. Only those languages that are critical for the semantic variation points of fUML require special discussion. For example, the language engineers may need to explicitly define a thread schedule model and then use this model for realising concurrent execution.
- How to deal with semantics that are out of scope? This is common when a DSL can embed other DSLs; for example, an XML-based language can reuse XPath as its query language. However, the way in which the XPath expression is going to be evaluated and the way in which the query is executed are out of the scope. In such a

case, the developers need to develop a way of representing the semantics that are out of scope. For example, evaluating an XPath expression can be defined as an abstract activity, and it can then be used as a library without knowing how it works. Another occasion for using abstract activities is when some behaviour has already been defined in another technical space, and the language specification just needs to reuse the concept. In this case, the semantics of XPath may already be defined by FQLS, which means that the concepts in XPath can simply be imported as existing concepts.

Guideline: Behavioural semantics development

- Limit the use of inline statements. Inline statements enable the embedding other programming languages in ALF text. While this is convenient when generating code that is difficult to represent in ALF, inline statements can be detrimental to the quality of the standard. They disrupt the interoperability and direct executability, since a programme contains many inline statements that are platform specific to the inline language, and it is unlikely that an executor that can understand all types of inline statements will be available in the foreseeable future.
- Limit the editing of generated code. Editing of generated code is easy to lose when regenerating. Furthermore, manually editing generated code results in the models expiring, causing the entire development process to become code-centric [80]. To avoid this, Kelly and Tolvanen [76] suggested that generated code should not be edited or even looked at. When defining language specification in ALF, the only exception to this is those activities that are left intentionally undefined. These activities should be implemented as an extension to the generated reference implementation.
- Unify code style before implementation. To balance the opinions of the UML and Java programming society, ALF supports both UML-like syntax and Java-like syntax. While this gives developers flexibility, it also makes coding in two different styles possible. Since one specification that contains different coding style is not desirable, it is suggested that the language engineers agree on a unified coding style. Here follows a list of things to be considered:

- UML naming/general programme naming. UML naming supports the use of any string as a qualified name, while most GPL names have many constraints. Our experience is that the use of unrestrained UML names can make the generated code difficult to maintain, since long names and names that contain spaces need to be converted to a more restrained format in most programming languages. When using a special naming strategy for a GPL, the names must be checked to avoid ambiguity when generating code (See platform specific errors in Section 6.1).
- UML statements/Java statements. Local variable definitions and for loops both support the Java style and the UML style. Although this is not bad for a language specification, it is preferable to agree on one.
- Use sequence expression/prefer simple loops. Sequence expressions are short and expressive; thus, the use thereof is suggested in a language specification when possible. However, if the language engineers have special conditions, especially when the language specification will be used to generate code that does not support either lambda expressions or anonymous functions, they may agree to avoid the use of sequence expressions.
- Agree on a consistent way of naming prefixes. There are seven kinds of classifiers in ALF, including signals and associations, and some of them do not have a corresponding concept in other languages. Giving them a name with a special prefix can help to identify the original concept when debugging the generated code.

Guideline for developing behaviours

- Follow general good coding practices, such as preferring shorter operations to long ones and using meaningful variable names.
- Remember that the most important aim of a language specification is to be understood by its readers. Thus, the developers should prefer readability to efficiency. The programme should be well commented.
- Abstract activities should be minimised. Leaving many gaps in the language specification will make the specification nearly impossible to execute without making an effort to implement the gaps in the reference implementation.

- Abstract activities should be *atomic*. This means ensuring that abstract activity is a behaviour that is outside of the scope, and that these are behaviours that do not need to be specified in the language specification.

Guideline for creating semantics definition architecture

- How to attach behaviours to the meta-model. It is possible to add operations directly to the abstract syntax model, to add activities as supplements to the abstract syntax model, or to create a runtime meta-model. Each method has certain benefits and drawbacks. Methods for attaching behaviours are introduced in detail in Section 5.1.3.
- If some concepts of the language represent state concepts, state machines of the language need to be designed. In such a language execution model, there should be a class member of the concepts that represent the state of the object. The state machine should be designed by deciding the semantics of the states, and the transition signals' interfaces and conditions.

Guidelines for implements an external checking strategy

- It is possible that a tool that could load ALF, fUML or Ecore models is already available, and that language engineers have decided to use it. In such a situation, the FQLS framework has already provided transformations for fUML and Ecore; the language engineers need to create a mechanism to report errors. For example, Ecore2CSP is a tool that checks for various conflicts and unsatisfiable concepts in an Ecore model. The language engineers can treat this as a black box that loads an Ecore model and produces a report of its findings. They then need to process the findings, either by manually mapping the errors in the original model, or by using an automatic mechanism.
- It is possible that the language engineer could decide to use a particular tool, but that the input of that tool is not a format that our framework directly supported. In such a situation, the language engineers need to define a transformation, as well as a mechanism to report errors.

- It is possible that the language engineers do not have a tool, but that they know the algorithm or could find an algorithm to do the job. The checking is then completely reliant on the language developers' ability.

Appendix C. List of errors sourced from static checkers of other languages.

Class implements the same classes as super class

This is an error that can happen when the object inheritance design is wrong. For example, `Activity` is a base class and `A`, `B` and `C` are classes that inherit `Activity`. This is defined below:

```
A specializes Activity {...}
B specializes A {...}
C specializes B, Activity {...}
```

When `C` inherits `B`, `C` is already a subclass of `Activity`. Making `C` also inherit `Activity` is not correct syntax.

An Activity that has a return part must define a return statement

A syntax error is when an operation that has a return value has no proper return statement in the body of the operation. The simplest way of checking this kind of error is that when an operation has a return type, its enclosed statements must have at least one return statement. This method can create false negatives, because when multiple exit routes exist, the method must guarantee that each of the possible exits has a return statement.

Empty code block

Any statement that can include a block of statements (that is, if, while, for, accept, inline, switch) can be empty, and such empty blocks indicate a deficiency in semantics or improper practices. Such errors can be checked by adding an OCL constraint to the ALF meta-model, or can simply be checked by a model validator, which will test whether a statement block contains a statement or not.

Unused/unwritten class members or variables

When a class member (an attribute/a signal reception) or a local variable (includes the parameters of an operation) is defined, but never used within their scope, this indicates that such class members or variables are redundant and should be removed.

Unwritten class members or variables are the members or variables that are defined, but the value thereof is never changed. This means that the symbol that represents these variables has never appeared in an assignment expression or a `createLink` expression. An unwritten member is not always bad practice, but it can indicate design deficiency.

Compare two objects when they are not comparable

When checking the equality of two objects, for example:

```
if (a==b) {}
```

if the types of `a` and `b` are not comparable (they have different static types), this can indicate something wrong in the specification.

Class defines an attribute that masks a superclass attribute

An attribute masks a super-class attribute, which means that an attribute has the same name as a super-class attribute and causes the attribute of the super-class to be overridden.

Consider the following example:

```
public class Vehicle{
    public engine : Engine;
}
public class Lorry specializes Vehicle{
    public engine : LorryEngine;
}
```

The `Lorry.engine` attribute masks the same attribute with the same name as its super-class, which causes the same name to stand for different attributes of the class, hence causing confusion.

Ambiguous ! operator

The `!` Boolean operator has a higher calculation priority than does the `instanceof` operator. However, developers may forget this rule and write code like:

```
if (! parent instanceof AbstractScope)
```

This is equivalent to

```
if ((!parent) instanceof AbstractScope)
```

However, since the `parent` variable is not a Boolean expression, the developer is likely to mean:

```
if (! (parent instanceof AbstractScope))
```

This rule checks the type of the operand of the `!` operator. If the type is not Boolean, the checker issues a warning to the developer.

Unnecessary type check done using instanceof operator

This error happens when using the `instanceof` operator, and the result of the `instanceof` expression can be deducted by analysing the static type of its operands. For example:

```
Empty a;
if (a instanceof Empty) {...}
```

When using `instanceof`, it is unnecessary to check if an object is known to be an instance or a sub-class instance of a particular type. The type of the left-hand-side and right-hand-side operands can be decided statically by type checking if all the types of an object can be determined. When the left-hand-side is a sub-type of the right-hand-side, the expression will always be true.

ALF does not force the type of an object to be declared, which means that, as with other dynamic languages, the type of an object may only be determined at runtime. In addition, due to the effect of the `classify` statement, the type of an object can be changed during the execution. Thus, the simple method of checking unnecessary `instanceof` does not guarantee completeness or soundness.

Obvious infinite loop

A loop without a termination is obviously a flaw in the code. This simple rule only checks a particular type of infinite loop, namely a loop in which the exit condition is true and there is no `break` statement inside the code block. The code below is an example

```
while (true){  
    //no break statement inside the code block  
}
```

Impossible cast

An impossible cast is defined as a type cast expression, for example, `(CastedClass) c`, in which the reference that has been cast is impossible to be cast to the desired type. Type casts can occur successfully between basic data types. When casting an object to another type, the object must be a part of the target class hierarchy, which is a super class or a sub-class. In the example, if the checker knows the exact type of `c`, and if it is not possible to cast the type to the desired type, this cast is categorised as an impossible cast.

The static checker uses the type resolver to analyse the type of the expression that has been cast. Although it is not always possible to determine the runtime type of an expression, there are many special cases that can be checked, such as when the declaration of the instance is close to the cast expression.

Method names differ only in capitalisation.

Method names that are only differentiated by capitalisation, like `doSomething()` or `DoSomething()`, are confusing. This check issues a warning if such confusing names appear.

Bibliography

- [1] IEEE Standard for System and Software Verification and Validation, 2012.
- [2] Islam Abdelhalim, James Sharp, Steve Schneider, and Helen Treharne. Formal Verification of Tokeneer Behaviours Modelled in fUML Using CSP. In Jin Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *Lecture Notes in Computer Science*, pages 371–387. Springer Berlin / Heidelberg, 2010.
- [3] David H. Akehurst. Validating BPEL Specifications Using OCL. Technical report, University of Kent at Canterbury technical report: 15-04, 2004.
- [4] Scott Ambler. *Agile modeling: effective practices for extreme programming and the unified process*. John Wiley & Sons, 2002.
- [5] Jorge Aranda, Daniela Damian, and Arber Borici. Transition to Model-Driven Engineering. In RobertB. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 692–708. Springer Berlin Heidelberg, 2012.
- [6] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41(2):155–166, 2011.
- [7] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. *Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin / Heidelberg, 2010.
- [8] Thomas Baar. An OCL Semantics Specified with QVT. In *Proceedings, MoDELS/UML 2006*, pages 1–6. Springer, 2006.
- [9] Nils Bandener, Christian Soltenborn, and Gregor Engels. Extending DMM Behavior Specifications for Visual Execution and Debugging. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 357–376. Springer Berlin / Heidelberg, 2011.
- [10] R. Bendraou, B. Combemale, X. Cregut, and M.-P. Gervais. Definition of an Executable SPEM 2.0. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pages 390–397, Dec 2007.

- [11] Luca Berardinelli, Philip Langer, and Tanja Mayerhofer. Combining fUML and Profiles for Non-functional Analysis Based on Model Execution Traces. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, pages 79–88, New York, NY, USA, 2013. ACM.
- [12] Gerard Berry, Georges Gonthier, Ard Berry Georges Gonthier, and Place Sophie Laltte. *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*, 1992.
- [13] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruehl, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 53–67. Springer Berlin / Heidelberg, 2008.
- [14] Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In Wil van der Aalst and Eike Best, editors, *Applications and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer Berlin / Heidelberg, 2003.
- [15] Conrad Bock and Michael Gruninger. PSL: A semantic domain for flow models. *Software and Systems Modeling*, 4:209–231, 2005.
- [16] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2012.
- [17] M.G.J. Brand, A. Deursen, J. Heering, H.A. Jong, M. Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-environment: A Component-Based Language Development Environment. In Reinhard Wilhelm, editor, *Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer Berlin Heidelberg, 2001.
- [18] Lionel Briand, Clay Williams, Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. *Weaving Executability into Object-Oriented Meta-languages*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer Berlin / Heidelberg, 2005.

- [19] B.R. Bryant, J. Gray, M. Mernik, P.J. Clarke, R.B. France, and G. Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Computer Science and Information Systems*, 8(2):225–253, 2011.
- [20] Jordi Cabot, Robert Claris #243, and Daniel Riera. Verification of UML/OCL Class Diagrams using Constraint Programming. In *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, pages 73–80. IEEE Computer Society, 2008.
- [21] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ASE '07*, pages 547–548, New York, NY, USA, 2007. ACM.
- [22] Eric Cariou, Cyril Ballagny, Alexandre Feugas, and Franck Barbier. Contracts for Model Execution Verification. In *ECMFA*, pages 3–18, 2011.
- [23] Michel Chaudron, José Rivera, José Romero, and Antonio Vallecillo. *Behavior, Time and Viewpoint Consistency: Three Challenges for MDE*, volume 5421 of *Lecture Notes in Computer Science*, pages 60–65. Springer Berlin / Heidelberg, 2009.
- [24] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 35–43. ACM, 2005.
- [25] Tony Clark, Paul Sammut, and James Willans. *Applied metamodelling: A foundation for language driven development, Second edition*. CETEVA, 2008.
- [26] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. *Journal of Software*, 4(9):943–958, 2009.
- [27] T. Copeland. *PMD applied*. Centennial Books, 2005.
- [28] Simone André da Costa and Leila Ribeiro. Verification of graph grammars using a logical approach. *Science of Computer Programming*, In Press, Corrected Proof:–, 2010.
- [29] Chris Daly. Emfatic Language Reference. <http://www.eclipse.org/gmt/epsilon/doc/-articles/emfatic/>, 2004.
- [30] Juan de Lara and Hans Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages & Computing*, 15(3-4):309–330, 2004.

- [31] Frédéric Kurtev Ivan Bézivin Jean Pierantonio Alfonso Di Ruscio, Davide Jouault. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical report, Laboratoire d'Informatique de Nantes-Atlantique, 2006.
- [32] W.J. Dzidek, E. Arisholm, and L.C. Briand. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. *Software Engineering, IEEE Transactions on*, 34(3):407–432, may-june 2008.
- [33] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.
- [34] M. Elaasar and L. Briand. An Overview of UML Consistency Management. Technical report, Department of Systems and Computer Engineering 1125 Colonel-By Drive, Ottawa, Ontario, K1S 5B6 Canada, 2004.
- [35] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In *Proceedings of the 3rd international conference on The unified modeling language: advancing the standard*, UML'00, pages 323–337, Berlin, Heidelberg, 2000. Springer-Verlag.
- [36] R. Esser and J.W. Janneck. Moses-a tool suite for visual modeling of discrete-event systems. In *Human-Centric Computing Languages and Environments, 2001. Proceedings IEEE Symposia on*, pages 272–279, 2001.
- [37] D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative control flow. In *Proc. 12th International Workshop on Abstract State Machines*, pages 131–151, 2005.
- [38] Dirk Fahland. Complete Abstract Operational Semantics for the Web Service Business Process Execution Language. Informatik-Berichte 190, Humboldt-Universität zu Berlin, September 2005.
- [39] Eclipse Foundation. Graphical Modeling Project (GMP). <http://www.eclipse.org/-modeling/gmp/>, 2011.
- [40] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A semantic framework for metamodel-based languages. *Automated Software Engineering*, 16(3):415–454, 2009.
- [41] Amir Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer (STTT)*, -:1–26, 2011.

- [42] U. Glässer, R. Gotzhein, and A. Prinz. The formal semantics of SDL-2000: Status and perspectives. *Computer Networks*, 42(3):343 – 358, 2003.
- [43] R.C. Gronback. *Eclipse modeling project: a domain-specific language toolkit*. The Eclipse series. Addison-Wesley, 2009.
- [44] Object Management Group. UML 1.4 with Action Semantics. <http://www.omg.org/cgi-bin/doc?ptc/02-01-09>, 2002.
- [45] Object Management Group. MDA Specifications. <http://www.omg.org/mda/specs.htm>, 2003.
- [46] Object Management Group. Object Constraint Language, Version 2.0. <http://www.omg.org/spec/OCL/2.0/>, 2006.
- [47] Object Management Group. MOF Model To Text Transformation Language (MOFM2T), 1.0. <http://www.omg.org/spec/MOFM2T/1.0/>, 2008.
- [48] Object Management Group. Software & Systems Process Engineering Metamodel Specification (SPEM) Version 2.0. <http://www.omg.org/spec/SPEM/2.0/>, 2008.
- [49] Object Management Group. Ontology Definition Metamodel (ODM). <http://www.omg.org/spec/ODM/1.0/>, 2009.
- [50] Object Management Group. Action Language For Foundational UML (ALF) 1.0 - Beta 1. www.omg.org/spec/ALF/, 2010.
- [51] Object Management Group. OMG’s MetaObject Facility. <http://www.omg.org/mof/>, 2010.
- [52] Object Management Group. Business Process Model And Notation (BPMN) Version 2.0. <http://www.omg.org/spec/BPMN/2.0/>, 2011.
- [53] Object Management Group. Documents Associated With Meta Object Facility (MOF) 2.0 Query/View/Transformation, V1.1. <http://www.omg.org/spec/QVT/1.1/>, 2011.
- [54] Object Management Group. Precise Semantics of UML Composite Structures RFP. <http://www.omg.org/cgi-bin/doc?ad/11-12-07>, 2011.
- [55] Object Management Group. Semantics Of A Foundational Subset For Executable UML Models (FUML), Version 1.0. <http://www.omg.org/spec/FUML/1.0/>, 2011.
- [56] Object Management Group. UML Profile For MARTE: Modeling And Analysis Of Real-Time Embedded Systems. <http://www.omg.org/spec/MARTE/>, 2011.
- [57] Object Management Group. OMG Systems Modeling Language. <http://www.omgsysml.org/>, 2012.

- [58] W3C Working Group. Web Services Glossary. [http://www.w3.org/TR/2004/NOTE-
ws-gloss-20040211/#webservice](http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice), 2004.
- [59] Volker Gruhn and Ralf Laue. Complexity metrics for business process models. In *in: W. Abramowicz, H.C. Mayr (Eds.), 9th International Conference on Business Information Systems (BIS 2006), Lecture Notes in Informatics*, pages 1–12, 2006.
- [60] Y Gurevich, B Rossman, and W Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, 2005.
- [61] Christian Hahn and Klaus Fischer. The Formal Semantics of the Domain Specific Modeling Language for Multiagent Systems. In Michael Luck and Jorge Gomez-Sanz, editors, *Agent-Oriented Software Engineering IX*, volume 5386 of *Lecture Notes in Computer Science*, pages 145–158. Springer Berlin / Heidelberg, 2009.
- [62] Maurice Howard Halstead. *Elements of software science*, volume 19. Elsevier New York, 1977.
- [63] Kevin Hammond and Greg Michaelson. Hume: A Domain-Specific Language for Real-Time Embedded Systems. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 37–56. Springer Berlin Heidelberg, 2003.
- [64] Warren A Harrison. Applying McCabe’s complexity measure to multiple-exit programs. *Software: Practice and Experience*, 14(10):1004–1007, 1984.
- [65] Constance Heitmeyer. On the Need for Practical Formal Methods. In *In Formal Techniques in RealTime and Real-Time Fault-Tolerant Systems, Proc., 5th Intern. Symposium (FTRTFT’98*, pages 18–26. Springer Verlag, 1998.
- [66] B. Henderson-Sellers and D. Tegarden. The theoretical extension of two versions of cyclomatic complexity to multiple entry/exit modules. *Software Quality Journal*, 3(4):253–269, 1994.
- [67] Alan Hevner and Samir Chatterjee. *Design Research in Information Systems*. Springer, 2010.
- [68] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, March 2004.
- [69] Peter Ölveczky, José Rivera, Francisco Durán, and Antonio Vallecillo. *On the Behavioral Semantics of Real-Time Domain Specific Visual Languages*, volume 6381 of *Lecture Notes in Computer Science*, pages 174–190. Springer Berlin / Heidelberg, 2010.

- [70] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.
- [71] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.
- [72] S. C. Johnson. Lint, a C Program Checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.
- [73] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008.
- [74] Lennart C.L. Kats and Eelco Visser. The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. *SIGPLAN Not.*, 45(10):444–463, October 2010.
- [75] Steven Kelly and Risto Pohjonen. Worst Practices for Domain-Specific Modeling. *IEEE Software*, 26:22–29, 2009.
- [76] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.
- [77] Pierre Kelsen and Qin Ma. A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 690–704. Springer Berlin / Heidelberg, 2008.
- [78] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.
- [79] A.G. Kleppe. A Language Description is More than a Metamodel. In *Fourth International Workshop on Software Language Engineering*, Grenoble, France, October 2007. megaplanet.org.
- [80] Anneke G Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Addison-Wesley Professional, 2009.

- [81] Dimitrios Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, Department of Computer Science, The University of York, 2008.
- [82] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2008.
- [83] Fabrice Kordon and Yann Thierry-Mieg. Experiences in Model Driven Verification of Behavior with UML. In Christine Choppy and Oleg Sokolsky, editors, *Foundations of Computer Software. Future Trends and Techniques for Development*, volume 6028 of *Lecture Notes in Computer Science*, pages 181–200. Springer Berlin / Heidelberg, 2010.
- [84] T. Kosar, N. Oliveira, M. Mernik, V.J.M. Pereira, M. Crepinšek, C.D. Da, and R.P. Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2):247–264, 2010.
- [85] Sabine Kuske. *A Formal Semantics of UML State Machines Based on Structured Graph Transformation*, volume 2185 of *Lecture Notes in Computer Science*, pages 241–256–256. Springer Berlin / Heidelberg, 2001.
- [86] Formal Methods laboratory University of Milan. ASMETA. <http://asmeta.sourceforge.net/>, 2010.
- [87] Qinan Lai and Andy Carpenter. Defining and verifying behaviour of domain specific language with fUML. In *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications*, BM-FA '12, pages 1:1–1:7, New York, NY, USA, 2012. ACM.
- [88] Qinan Lai and Andy Carpenter. Static Analysis and Testing of Executable DSL Specification. In *Proceedings of 1st International Conference on Model-Driven Engineering and Software Development*, 2013.
- [89] Kristian Bisgaard Lassen and Wil M.P. van der Aalst. Complexity metrics for Workflow nets. *Information and Software Technology*, 51(3):610 – 626, 2009.
- [90] Yoann Laurent, Reda Bendraou, Souheib Baarir, and Marie-Pierre Gervais. Formalization of fUML: An Application to Process Verification. In Matthias Jarke, John Mylopoulos, Christoph Quix, Colette Rolland, Yannis Manolopoulos, Haralambos Mouratidis, and Jennifer Horkoff, editors, *Advanced Information*

- Systems Engineering*, volume 8484 of *Lecture Notes in Computer Science*, pages 347–363. Springer International Publishing, 2014.
- [91] C.L. Lazăr, I. Lazăr, B. Pârv, S. Motogna, and I.G. Czibula. Tool Support for fUML Models. *International Journal of Computers Communications & Control*, 5(5):775–782, 2010.
 - [92] António Menezes Leitão. From Lisp S-expressions to Java source code. *Computer Science and Information Systems/ComSIS*, 5(2):19–38, 2008.
 - [93] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.
 - [94] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. *SIGPLAN Not.*, 45(11):67–78, September 2010.
 - [95] Chengying Mao. Complexity Analysis for Petri Net-based Business Process in Web Service Composition. In *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, pages 193–196. IEEE, 2010.
 - [96] Tanja Mayerhofer, Philip Langer, and Manuel Wimmer. Towards xMOF: executable DSMLs based on fUML. In *Proceedings of the 2012 workshop on Domain-specific modeling*, DSM ’12, pages 1–6, New York, NY, USA, 2012. ACM.
 - [97] Thomas J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, 4(4):308–320, 1976.
 - [98] Steve McConnell. *Code complete*. O’Reilly Media, Inc., 2004.
 - [99] Stephen J Mellor. *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.
 - [100] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
 - [101] P. Mohagheghi and J. Aagedal. Evaluating Quality in Model-Driven Engineering. In *International Workshop on Modeling in Software Engineering (MiSE’07) ICSE Workshop*, page 6, may 2007.
 - [102] Parastoo Mohagheghi and Vegard Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture - Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 432–443. Springer Berlin / Heidelberg, 2008.

- [103] Parastoo Mohagheghi, Vegard Dehlen, and Tor Neple. Definitions and approaches to model quality in model-based software development - A review of literature. *Information and Software Technology*, 51(12):1646 – 1669, 2009.
- [104] Pierre-Alain Muller, Frédéric Fondement, and Benoît Baudry. Modeling Modeling. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 2–16. Springer Berlin Heidelberg, 2009.
- [105] H.James Nelson and DavidE. Monarchi. Ensuring the quality of conceptual representations. *Software Quality Journal*, 15(2):213–233, 2007.
- [106] OASIS. Web Services Business Process Execution Language Version 2.0 Primer. <http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.html>, 2007.
- [107] Technical University of Berlin. The Attributed Graph Grammar System. <http://user.cs.tu-berlin.de/~gragra/agg/>, 2010.
- [108] University of Illinois at Urbana-Champaign. The Maude System. <http://maude.cs.uiuc.edu/>, 2010.
- [109] University of Maribor. LISA. <http://labraj.uni-mb.si/lisa/index.html>, 2006.
- [110] Greg O’Keefe. Improving the Definition of UML. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 42–56. Springer Berlin / Heidelberg, 2006.
- [111] Andy Oram and Greg Wilson. *Making Software*. O’Reilly, 2010.
- [112] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. Metamodelling for Grammarware Researchers. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 64–82. Springer Berlin Heidelberg, 2013.
- [113] P.R. Panda. SystemC - a modeling platform supporting multiple design abstractions. In *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, pages 75–80, 2001.
- [114] M. Peleg and D. Dori. The model multiplicity problem: experimenting with real-time specification methods. *Software Engineering, IEEE Transactions on*, 26(8):742–759, 2000.
- [115] James L. Peterson. Petri Nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.
- [116] Marian Petre. UML in practice. In *35th International Conference on Software Engineering (ICSE 2013)*, 2013.

- [117] Elena Planas, Jordi Cabot, and Cristina Gómez. Verifying Action Semantics Specifications in UML Behavioral Models. In Pascal van Eck, Jaap Gordijn, and Roel Wieringa, editors, *Advanced Information Systems Engineering*, volume 5565 of *Lecture Notes in Computer Science*, pages 125–140. Springer Berlin / Heidelberg, 2009.
- [118] Elena Planas, Jordi Cabot, and Cristina Gomez. Lightweight Verification of Executable Models. In *30th International Conference on Conceptual Modeling (ER 2011)*, 2011.
- [119] Elena Planas, Jordi Cabot, Cristina Gomez, Esther Guerra, and Juan de Lara. Lightweight Executability Analysis of Graph Transformation Rules. *Visual Languages and Human-Centric Computing, IEEE Symposium on*, 0:127–130, 2010.
- [120] Elena Planas, David Sanchez-Mendoza, Jordi Cabot, and Cristina Gómez. Alf-Verifier: An Eclipse Plugin for Verifying Alf/UML Executable Models. In Silvana Castano, Panos Vassiliadis, LaksV. Lakshmanan, and MongLi Lee, editors, *Advances in Conceptual Modeling*, volume 7518 of *Lecture Notes in Computer Science*, pages 378–382. Springer Berlin Heidelberg, 2012.
- [121] G. D. Plotkin. A Structural Approach to Operational Semantics, 1981.
- [122] Andreas Prinz, Markus Scheidgen, and Merete Tveit. A Model-Based Standard for SDL. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007: Design for Dependable Systems*, volume 4745 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin / Heidelberg, 2007.
- [123] Raman Ramsin and Richard F. Paige. Process-centered review of object oriented software development methodologies. *ACM Comput. Surv.*, 40(1):3:1–3:89, February 2008.
- [124] W3C Recommendation. OWL Web Ontology Language. <http://www.w3.org/TR/owl-features/>, 2004.
- [125] Arend Rensink. *The Edge of Graph Transformation - Graphs for Behavioural Specification*, volume 5765 of *Lecture Notes in Computer Science*, pages 6–32–32. Springer Berlin / Heidelberg, 2010.
- [126] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 219–222. Springer Berlin / Heidelberg, 2004.

- [127] Elvinia Riccobene and Patrizia Scandurra. Weaving executability into UML class models at PIM level. In *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, BM-MDA '09, pages 1:1–1:9, New York, NY, USA, 2009. ACM.
- [128] José E. Rivera. *On the semantics of real-time Domain Specific Modeling Languages*. PhD thesis, Universidad de Málaga, 2010.
- [129] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. The Epsilon Generation Language. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture - Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2008.
- [130] Matti Rossi and Sjaak Brinkkemper. Complexity metrics for systems development methods and techniques. *Information Systems*, 21(2):209 – 227, 1996.
- [131] Daniel A. Sadilek and Guido Wachsmuth. Using Grammarware Languages to Define Operational Semantics of Modelled Languages. In Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, Clemens Szyperski, Manuel Oriol, and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, pages 348–356. Springer Berlin Heidelberg, 2009.
- [132] Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. In *Proceedings of the 3rd European conference on Model driven architecture-foundations and applications*, ECMDA-FA'07, pages 157–171, Berlin, Heidelberg, 2007. Springer-Verlag.
- [133] Ina Schieferdecker, Alan Hartman, Daniel Sadilek, and Guido Wachsmuth. *Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages*, volume 5095 of *Lecture Notes in Computer Science*, pages 63–78. Springer Berlin / Heidelberg, 2008.
- [134] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.
- [135] E. Seidewitz. What models mean. *Software, IEEE*, 20(5):26–32, 2003.
- [136] B. Selic. A Systematic Approach to Domain-Specific Language Design Using UML. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*, pages 2–9, 2007.
- [137] Bran Selic. The Theory and Practice of Modeling Language Design for Model-Based Software Engineering - A Personal Perspective. In João Fernandes, Ralf Lämmel,

- Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 290–321. Springer Berlin / Heidelberg, 2011.
- [138] Traian Florin Serbanuta. *A REWRITING APPROACH TO CONCURRENT PROGRAMMING LANGUAGE DESIGN AND SEMANTICS*. PhD thesis, University of Illinois at Urbana-Champaign, 2011.
- [139] Jingqiu Shao and Yingxu Wang. A new measure of software complexity based on cognitive weights. *Electrical and Computer Engineering, Canadian Journal of*, 28(2):69–74, 2003.
- [140] Keng Siau and M. Rossi. Evaluation of information modeling methods-a review. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 5, pages 314 –322 vol.5, jan 1998.
- [141] Balazs Simon, Balazs Goldschmidt, and Karoly Kondorosi. A Human Readable Platform Independent Domain Specific Language for BPEL. In Filip Zavoral, Jakub Yaghob, Pit Pichappan, and Eyas El-Qawasmeh, editors, *Networked Digital Technologies*, volume 87 of *Communications in Computer and Information Science*, pages 537–544. Springer Berlin Heidelberg, 2010.
- [142] Christian Soltenborn and Gregor Engels. *Towards Test-Driven Semantics Specification*, volume 5795 of *Lecture Notes in Computer Science*, pages 378–392. Springer Berlin / Heidelberg, 2009.
- [143] Model Driven Solutions. Action Language for UML (Alf) Open Source Implementation. <http://modeldriven.org/alf/>, retrived on 06/06/2013, 2013.
- [144] Jonathan Sprinkle, Marjan Mernik, Juha-Pekka Tolvanen, and Diomidis Spinellis. Guest Editors’ Introduction: What Kinds of Nails Need a Domain-Specific Hammer? *IEEE Software*, 26:15–18, 2009.
- [145] Christian Stahl. A Petri Net Semantics for BPEL. Technical report, Humboldt-Universität zu Berlin, 2005.
- [146] OASIS standard. RELAX NG Specification. <http://relaxng.org/spec-20011203.html>, 2001.
- [147] OASIS standard. Web Services Business Process Execution Language Version 2.0. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>, 2007.
- [148] Dave Steinberg. *EMF : Eclipse Modeling Framework*. Addison-Wesley, Boston, Mass. ; London, 2nd ed., rev. and updated. edition, 2009.

- [149] Yu Sun, Jules White, and Jeff Gray. Model Transformation by Demonstration. *Model Driven Engineering Languages and Systems*, 5795:712–726, 2009.
- [150] Gerson Sunyé, François Pennaneac’h, Wai-Ming Ho, Alain Le Guennec, and Jean-Marc Jézéquel. Using UML Action Semantics for Executable Modeling and Beyond. In Klaus Dittrich, Andreas Geppert, and Moira Norrie, editors, *Advanced Information Systems Engineering*, volume 2068 of *Lecture Notes in Computer Science*, pages 433–447. Springer Berlin / Heidelberg, 2001.
- [151] Hideaki Takeda, Paul Veerkamp, and Hiroyuki Yoshikawa. Modeling design process. *AI magazine*, 11(4):37, 1990.
- [152] D.P. Tegarden, S.D. Sheetz, and D.E. Monarchi. Effectiveness of traditional software metrics for object-oriented systems. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume iv, pages 359 –368 vol.4, jan 1992.
- [153] International Telecommunication Union. ITU-T Recommendation Z.100 Annex F SDL Formal Definition, 2000.
- [154] V. Vaishnavi and W. Kuechler. Design Science Research in Information Systems. January 20, 2004, last updated November 11, 2012. URL: <http://www.desrist.org/design-research-in-information-systems/>, 2004.
- [155] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.
- [156] Markus Voelter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.
- [157] Guido Wachsmuth. Modelling the Operational Semantics of Domain-Specific Modelling Languages. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 506–520. Springer Berlin / Heidelberg, 2008.
- [158] Tabinda Waheed, Muhammad Iqbal, and Zafar Malik. Data Flow Analysis of UML Action Semantics for Executable Models. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture - Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 79–93. Springer Berlin / Heidelberg, 2008.
- [159] Jon Whittle, John Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *Software, IEEE*, 31(3):79–85, May 2014.

- [160] Claas Wilke and Birgit Demuth. UML is still inconsistent! How to improve OCL Constraints in the UML 2.3 Superstructure. *Electronic Communications of the EASST*, 44:1–19, 2011.
- [161] Yingzhou Zhang and Baowen Xu. A survey of semantic description frameworks for programming languages. *SIGPLAN Not.*, 39(3):14–30, 2004.