BABEŞ-BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Towards a Formal Foundation of Software Reuse

PhD Thesis Abstract

PhD student: **Vladiela Petraşcu**

Scientific advisor: Professor **Militon Frenţiu**

2011

Acknowledgements

I would like to thank all people which have made this work possible.

I express my sincere gratitude to my supervisor, Professor Militon Frenţiu, for the chance of pursuing a PhD in Software Engineering, as well as for his support, guidance and trust throughout the writing of this thesis.

I am deeply grateful to Dr. Dan Chiorean for the research opportunities offered, for all the knowledge he has shared with me, as well as for the insightful debates, constructive criticism, and continuous help throughout my doctoral study.

Last, but not least, I warmly thank my family, for all the emotional support they have provided me with.

The financial assistance received is also gratefully acknowledged.

The work described in Chapters 4 and 5, as well as Section 6.2 of this thesis has been supported by CNCSIS–UEFISCSU, project number PNII–IDEI 2049/2008 "Extensive Metamodeling-based Framework for Specifying, Implementing and Validating Languages and Applications".

The research reported in Section 6.1 has been leaded in the context of the ECO-NET Project Nr. 16293RG/2007 "Behaviour Abstraction from Code: Filling the Gap between Component Specification and Implementation", sponsored by Égide, France.

Contents

Acknowledgements **i**

1 Introduction **1**

2 Background **4**
- 2.1 Formal Methods and Languages 4
  - 2.1.1 Overview 4
  - 2.1.2 The B Method 4
  - 2.1.3 Design by Contract 4
  - 2.1.4 Object Constraint Language 4
- 2.2 Software Reuse and Reusable Assets 5
  - 2.2.1 Overview 5
  - 2.2.2 Object-Oriented Design Patterns 5
  - 2.2.3 Component-Based Software Development 5
  - 2.2.4 Model-Driven Engineering 5

3 Formalization of Design Patterns **6**
- 3.1 Motivation and Related Work 6
- 3.2 An Approach to Formalizing the State Pattern in B 6
  - 3.2.1 The State Design Pattern 7
  - 3.2.2 State Pattern Definition in B 7
  - 3.2.3 State Pattern Reuse in B 7
  - 3.2.4 Validation. Proof Activity Analysis 8
- 3.3 Summary 9

4 Constraint Patterns in Object-Oriented Modeling **10**
- 4.1 Motivation 10
- 4.2 Related Work 10
- 4.3 A New Approach: MDE-Driven OCL Specification Patterns 11
  - 4.3.1 Approached Patterns. Existing Solutions 11
  - 4.3.2 Proposed Solutions 12
    - 4.3.2.1 The *For All* Constraint Pattern 12
    - 4.3.2.2 The *Unique Identifier* Constraint Pattern - Invariant Solutions 12
    - 4.3.2.3 The *Unique Identifier* Constraint Pattern - Pre/Post-condition Solutions 13
  - 4.3.3 Tool Support and Validation 14
    - 4.3.3.1 Model Compilability Checks 14

4.3.3.2 Model Testing 15
4.4 Summary 15

5 The Static Semantics of (Meta)Modeling Languages 16
5.1 Motivation 16
5.1.1 The Model Compilability Problem 16
5.1.2 Diagnosing the State of Facts Regarding Model Compilability 16
5.2 Towards a Conceptual Framework Supporting Model Compilability - Principles of a Static Semantics Specification 17
5.3 Enhancements to the Static Semantics of (Meta)Modeling Languages 18
5.3.1 Enhancements to the Static Semantics of UML/MOF 18
5.3.1.1 State of Facts. Related Work 18
5.3.1.2 Proposed Enhancements 18
5.3.2 Enhancements to the Static Semantics of Ecore 21
5.3.2.1 State of Facts 21
5.3.2.2 Proposed Enhancements 21
5.3.2.3 Related Work 23
5.3.3 Enhancements to the Static Semantics of XCore 24
5.3.3.1 State of Facts 24
5.3.3.2 Proposed Enhancements 25
5.4 Summary 26

6 The Specification of Software Components 27
6.1 Towards Reverse Engineering Component Specifications from Code 27
6.1.1 Motivation and Related Work 27
6.1.2 General Approach 28
6.1.3 The Common Component MetaModel (CCMM) 28
6.1.4 Tool Support and Validation 29
6.2 ContractCML - A Contract-Aware Component Modeling Language 30
6.2.1 Motivation and Related Work 30
6.2.2 The ContractCML Metamodel 31
6.2.2.1 Metamodel Overview 31
6.2.2.2 Basic Contracts. Syntactic Specification of Interfaces 31
6.2.2.3 Behavioral Contracts. Semantic Specification of Interfaces . 32
6.2.2.4 A Model Weaving Approach to Representing Information Models 33
6.2.3 ContractCML Modeling Example 33
6.2.4 Component Models' Simulation Using ContractCML 33
6.2.4.1 Proposed Simulation Method 34
6.2.4.2 Validation 34
6.3 Summary 34

7 Conclusions 36

Bibliography 37

Abbreviations 44

Chapter 1

Introduction

Reuse - the use of existing artifacts within the development of new ones, is common practice in all engineering disciplines. Among its major benefits are higher productivity, thus lower time to market, lower production costs, and increased products' quality. In Software Engineering, the necessity of a mature approach to reuse has been first acknowledged openly at the 1968 NATO Conference, in the context of the so called "software crisis". Reuse has been proposed as a potential solution for the problems underneath it. Ever since, the reuse techniques in Software Engineering have evolved along with each newly emerged development paradigm. This evolution has occurred both in terms of granularity and abstraction level. Regarding granularity, things have advanced from code reuse in the form of procedures within the structured paradigm, to classes and objects within the object-oriented one, reaching components and frameworks reuse with the advent of Component-Based Software Development (CBSD). As to the abstraction level, there has been a major move from code reuse (procedures, classes, objects, components) to reuse of higher level abstractions such as object-oriented design patterns and architectures, ending with extensive reuse of entire models and metamodels, as promoted by the latest Model-Driven Engineering (MDE) approaches. The two previously-mentioned levels have also intermixed with a third one, concerning the type of reuse techniques employed, such as instantiation, composition or generative reuse.

Reuse, however, can only deliver its promises if provided with an appropriate formal foundation. This should address both the issue of an adequate formal specification of reusable artifacts and that of a formalized reuse process. *Formal methods* (covering languages, techniques and tools based on mathematical knowledge, which enables formal reasoning) have been proposed as a means of increasing software's reliability. Reliability is to be broadly understood as the absence of errors, including both correctness and robustness. In case of reusable assets, that are to be employed in a variety of other contexts, reliability is a must.

Reuse being a subject so vast, there are various open issues in the field of formalizing reuse and reusable assets. Within this thesis, we have approached four of its subdomains, concerning *object-oriented design and constraint patterns*, *(meta)modeling* and *software components*. In case of design patterns, the lack of formalism in specifying both their solutions and the associated reuse process makes it impossible to check their consistency, the correctness of their instantiations and limits their applicability to the development of critical systems. The OCL-based solutions currently provided for some of the object-oriented constraint patterns are inappropriate with respect to the role of assertions within an MDE development process (which is mainly that of supporting correctness checks of models and applications). In addition, the (meta)modeling languages employed by the MDE approaches have various drawbacks in their static semantics' specification, with a negative effect on model compilability checks. Two of the current problems in component-based software development are related to the existing gap between industrial component models (centered on implementation) and academic ones (focused on specification), as well as to the lack of component models allowing a full contractual specification of software components. Based on this state of facts, our general aim within this thesis has been that of recording a contribution to the solutioning of each of the previously-mentioned open issues.

Thesis structure. The present thesis is structured in seven chapters (an introduction, a background chapter, four chapters containing original contributions and a concluding one), has a bibliography including a number of 141 references and three appendices.

Chapter 1 introduces the context, motivation and goals of the thesis, summarizes the contributions brought within it and provides an outline of its contents.

Chapter 2 provides background information with respect to the formal approaches and reuse areas addressed by the contributions in this thesis. Concerning the formal approaches, we briefly overview the B formal method, the Design by Contract methodology and the OCL language. In the field of reusable assets and reuse-oriented paradigms, we provide a short introduction to object-oriented design patterns, Component-Based Software Development and Model-Driven Engineering.

Chapter 3 describes the first contribution reported by this thesis, which fits in the area of design pattern's formalization. The contribution consists in a full formalization of the GoF State design pattern in B. The presentation covers a short description of the pattern, its formal definition in B, a formalization of its reuse process illustrated by means of an example, as well as a detailed analysis of the proof activity performed with AtelierB.

Chapter 4 details our contribution with respect to the definition of constraint patterns for object-oriented models. The presentation starts with a description of the constraint patterns approached, together with their currently available solutions in the literature. Our proposals consist in a number of OCL specification patterns provided as solutions for the *For All* and *Unique Identifier* constraint patterns. A proof of concepts regarding the relevance of our approach is given by means of two meaningful cases studies, using OCLE.

Chapter 5 illustrates our proposals concerning the formalization of the static semantics of (meta)modeling languages. We start by emphasizing the compulsoriness of the model compilability requirement in the context of MDE and diagnosing the current state of facts in the field. Based on that, we argue on the need of a rigorous conceptual framework supporting the specification of the static semantics of (meta)modeling languages and enabling efficient model compilability checks. Further, we expose the principles that we see at the basis of such a framework and we propose improvements to the static semantics of UML/MOF, Ecore and XCore, in accordance to these principles.

Chapter 6 reports on our results in the field of software components' specification. It starts by introducing a contribution to a reverse engineering approach aimed at extracting structural and behavioral abstractions from component system implementations. This contribution has been established as part of an ECO-NET international project [1, 10]. The presentation covers the project motivation, the general approach proposed, the core of our contribution, as well as an overview of the provided tool-support and validation activity performed. Further in this chapter, we present a contribution intended to set the bases of a framework able to support an appropriate contractual specification of software components, with a special emphasis on semantic contracts. First, we introduce ContractCML, a domain specific modeling language ensuring the backbone of our proposal. The core contribution concerns the method proposed for representing components' semantic contracts within the language metamodel. The use of the language is illustrated by means of a component modeling example. Following, we describe a simulation approach regarding the execution of component services, which relies on the previously proposed method for representing semantic contracts.

Each of the chapters 3 - 6 starts with a motivation for approaching the field in question, includes an overview of related work and emphasizes the advantages of our proposals with respect to it. In addition, each such chapter ends with a summary of contributions and directions of future work on the topic.

Chapter 7 concludes our work and offers an overview of all contributions reported within it.

Keywords. reuse, formal methods, design patterns, constraint patterns, metamodeling, software components, MDE, WFR, CBSD, OCL, B

Chapter 2

Background

2.1 Formal Methods and Languages

2.1.1 Overview

In a broad sense, formal methods cover languages, techniques and tools based on mathematical modeling and formal logic, that are used throughout the specification, development and verification of software systems. Within this section, we have defined the concept of *formal language* and that of *formal method*, we have emphasized the value of formal methods in the development of critical systems and we have discussed about their classification. Most taxonomies distinguish among property-oriented and model-oriented formal methods.

2.1.2 The B Method

B [6] is a model-oriented formal method that supports the entire lifecycle of a software product. The main structuring unit of B models is the *abstract machine*, the notation employed by the method being called AMN (Abstract Machine Notation). System development in B starts with the creation of a mathematical model driven by the user requirements, expressed in terms of one or several abstract machines. This model is further refined or specialized, until reaching a complete system implementation. Both the consistency of the initial model and the correctness of all refinement steps are guaranteed by mathematical proofs. Among the most powerful B prover tools is AtelierB [32].

Within this section, we have described the AMN notation, together with the mechanisms ensuring incremental model development and refinement.

2.1.3 Design by Contract

Design by Contract (*DBC*) [54] is a methodology proposing a contractual approach to the development of object-oriented software, based on the use of assertions. The final aim is that of increasing software's reliability. Within this section, we have approached the main assertion types (pre/post-conditions and invariants), we have provided a definition of class correctness with respect to them and we have emphasized the purpose of writing assertions.

2.1.4 Object Constraint Language

OCL (*Object Constraint Language* [61, 87]) is a formal language used to define expressions on UML (Unified Modeling Language [62, 63]) models. Within this section, we have approached the OCL language features, OCL's role in metamodeling and the OCL dialects (e.g. XOCL

[31]), as well as the OCL tool support. Regarding metamodeling, we have defined the concepts of *Well-Formedness Rule* (*WFR*) and *Additional Operation* (*AO*); with respect to tools, we have summarized the facilities offered by OCLE (OCL Environment [49]) in specifying and evaluating assertions.

2.2 Software Reuse and Reusable Assets

2.2.1 Overview

Roughly speaking, *reuse* concerns the use of existing artifacts within the development of new ones [48]. Within this section, we have provided definitions of the concepts *reuse*, *reusability* and *reusable artifact*, we have argued on the advantages of adopting a reuse-based development process and we have illustrated the evolution of Software Engineering reuse techniques.

2.2.2 Object-Oriented Design Patterns

In software development, *design patterns* provide general recommended solutions to recurrent design problems. Within this section, we have addressed the roots and the evolution of the *pattern* concept, we have introduced a granularity-based pattern taxonomy, and we have discussed the pattern specification style adopted by the GoF catalog [43], underlining its informal nature.

2.2.3 Component-Based Software Development

This section provides a short overview of the *Component-Based Software Development* (*CBSD* [73]) paradigm. Within it, we have defined the concept of *software component*, we have discussed about industrial (COM (Component Object Model [19]), .NET [88], Web Services [55], CCM (CORBA Component Model [56]), JavaBeans [72], EJB (Enterprise JavaBeans [35])) and academic (Fractal [21], SOFA (SOFtware Appliances [22]), KobrA [13], Kmelia [9]) component models and we have addressed the component specification problem. With regard to the latter, we have summarized the *UML Components* methodology introduced in [24].

2.2.4 Model-Driven Engineering

Within this section, we have offered a short introduction to the model-driven software development paradigm. We have defined the basic concepts at its roots: *model*, *metamodel*, *meta-metamodel*, *Domain Specific Modeling Language* (*DSML*). We have summarized its different approaches: Model Driven Architecture (MDA [57]), Model-Driven Engineering (MDE [16, 17, 70]) and Language-Driven Development (LDD, [31]). MDA is based exclusively on OMG (Object Management Group) standards, its meta-metamodel being represented by MOF (Meta Object Facility [60]). MDE is the name used to refer the model-driven approaches including formalisms and technologies which are independent of the OMG standards. The best known MDE framework is EMF (Eclipse Modeling Framework [36]); this is based on the Ecore meta-metamodel and integrates powerful tools, among which MDT OCL (Model Development Tools OCL [38]) and oAW (openArchitectureWare [5]). LDD is based on the XCore meta-metamodel, this approach being implemented within the XMF Mosaic tool [3].

Chapter 3

Formalization of Design Patterns

Object-oriented design patterns are nowadays acknowledged as one of the most popular approaches in the landscape of software engineering reuse techniques. This chapter summarizes our work of investigating design patterns' formalization in general, and their formalization using the B formal method, in particular. The main contribution reported here concerns a full formalization of the GoF State pattern in B. This has allowed us to explore the limits of an existing approach regarding the reuse of patterns with the B method [45] and to propose a potential improvement.

3.1 Motivation and Related Work

The research reported in this chapter is motivated by the informal nature of the original description given to (GoF) design patterns. This lack of formalism is twofold: on the one side, the pattern themselves are described using a mixture of natural language, UML/ OMT diagrams and code samples, on the other, their reuse process is not formalized either. Both facts have had a major contribution to the success design patterns enjoy of today. Namely, the comprehensive, human-readable description of patterns ensures high understandability, enabling a straightforward identification of the solutions that best fit a particular design problem, while the unformalized reuse process provides a certain flexibility in adapting the patterns to the specifics of a new application [18]. However, the lack of formalism makes it impossible to check either the consistency of the patterns themselves or the correctness of their instantiations, thus limiting their applicability to the development of safety critical systems [52]. Moreover, the selection and reuse processes are hard to automate.

In this context, formalizing design patterns and their reuse has become a challenging research issue. Several approaches have been proposed in this area. Among the most relevant ones, we mention the creation of a specialized formal language called LePUS (Language for Patterns Uniform Specification [39, 40]), a formalization approach based on the RSL (RAISE Specification Language) language [23, 41], the formal specification of frameworks in Catalysis [50], as well as a few formalization proposals with the aid of the B formal method [18, 45, 51, 52].

3.2 An Approach to Formalizing the State Pattern in B

Our contribution to this field consists of a full formalization of the State design pattern using the B method. This covers both the formal definition of the pattern itself and the formalization of its associated reuse process.

3.2.1 The State Design Pattern

State [43] is a behavioral GoF pattern that provides an answer to the problem of designing a class (`Context`) whose objects have complex, state-dependent behavior. The solution's structure is given by the diagram in Figure 3.1. The core idea is that of creating an abstract class (`State`) to encapsulate state-specific behavior, which is then implemented in each of the concrete classes inheriting `State`. Each `Context` instance holds a reference to its current state and delegates to the latter's `Handle`$_i$ method (part of) each `Request`$_i$ received.

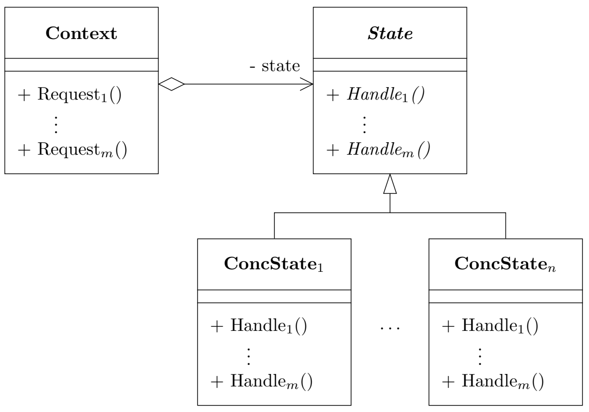

Figure 3.1: State pattern's structure

3.2.2 State Pattern Definition in B

The formal definition of the pattern has been manually created, starting from its informal description, as provided by the GoF catalog and driven by a set of UML-to-B translation rules available in the literature [71]. A core aspect here is related to the fact that the polymorphic behavior and the delegation idea have not been formalized at this level, being delayed to the first step of the reuse process (instantiation with renaming). This has allowed us to avoid a severe limitation regarding the number of concrete states involved[1]. Experimentation has shown that this delay proposal does not affect the number of proofs reused while instantiating the pattern in B.

The outcome of this step is represented by the `StatePattern` abstract machine. The resulting machine has been checked for correctness using the AtelierB tool [32]. All the generated proofs have been automatically discharged, acknowledging its consistency.

3.2.3 State Pattern Reuse in B

The State pattern's reuse process is illustrated by means of an average-complexity case study, involving the B formal modeling of an LCD Wallet Travelling Clock [74, 82]. While formalizing the reuse process, we have roughly followed the framework proposed in [45]. Consequently, the proposed formalization in achieved in two steps: an instantiation, followed by an extension. The former is achieved by means of the B inclusion mechanism, while the latter is accomplished using B refinement. Apart modularity, such a separation of concerns has the advantage of producing the instantiation machine at zero proof cost.

Step 1 - Machine inclusion. This is an intermediate reuse step, the output of which is represented by the `RenamedStatePattern` abstract machine. At this level, the pattern has been specialized by means of appropriate renamings, information related to the concrete states and new operations. This machine formalizes the polymorphic and delegation behaviors. Moreover, it is built so that to avoid the generation of additional (non-obvious) proof obligations. Therefore, the proof of its consistency has been trivially achieved.

As we have shown, this first reuse step is entirely automatable, based on user-provided information covering: name of the context-type class, number and names of concrete states, initial state of a context-type object, number and names of requests, textual form of a

[1]This limitation concerns working with a predefined number of concrete states. It is one of the main drawbacks of the Composite pattern formalization proposed in [45]

state transition diagram involving all concrete states and requests, as well as names of new operations to be added. Therefore, our delay proposal does not introduce extra manual overhead when reusing the pattern.

Step 2 - Refinement. The outcome of this step is represented by the `Clock` refinement machine, which entirely formalizes the LCD clock behavior under consideration.

An overview of all B components that make up the clock's formal model and their iterrelationships is given by the diagram in Figure 3.2.

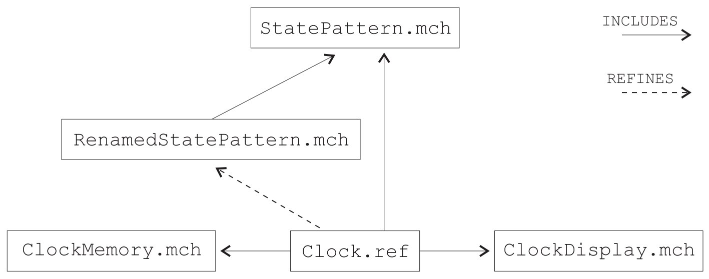

Figure 3.2: Structure of the B model corresponding to the LCD clock

3.2.4 Validation. Proof Activity Analysis

The entire B development has been supported by the AtelierB prover tool. Table 3.1 gives an overview of the proof activity performed.

Machine	TC	POG	nPOs	nAut	nInt	%Pr
`StatePattern`	OK	OK	7	7	0	100%
`RenamedStatePattern`	OK	OK	0	0	0	100%
`ClockMemory`	OK	OK	72	70	2	100%
`ClockDisplay`	OK	OK	29	29	0	100%
`Clock`	OK	OK	564	412	152	100%

Table 3.1: Overview of the proof activity

All machines have been successfully typechecked and all proof obligations (POs) have been successfully generated, as shown by the second and third columns of the table. The remaining columns indicate, from left to right, the number of non-obvious proof obligations generated (nPOs), the amount of proofs discharged automatically (nAut) and interactively (nInt) by the prover, as well as the proof success rate (%Pr).

Following, we have performed a detailed per-operation analysis of the proof obligations corresponding to the `StatePattern` machine and `Clock` refinement. For each operation, we have indicated the total number of POs generated, the number of such obligations per category (invariant-preservation vs. precondition-fulfillment ones), as well as the amount of proofs discharged interactively and automatically in each category. This analysis has revealed that, from the 564 POs generated for `Clock`, only three are directly connected to the instantiation of the pattern, all of them being automatically proved[2]. The remaining 561 proof are closely related to the specifics of the LCD case study. Although the number

[2]Taking into account also the facts that the `StatePattern` machine has been proven entirely automatically and that the `RenamedStatePattern` machine does not generate any non-obvious obligation, it follows that all the proof work concerning the definition and instantiation of the State pattern in B has been automatically performed.

of required interactive proofs seems high (152), the proof strategies that we have applied in order to discharge them have been quite similar. Manual decisions, such as performing proof by cases or by contradiction, instantiating universally quantified predicates, adding hypotheses or applying modus ponens, have been needed in order to direct the proofs.

As argued by [18], one of the advantages of applying patterns with the B method resides in the fact that not only the pattern definition itself is being reused, which leads to better modularity and improved safety, but so are the formal proofs associated with it. In order to assess the amount of proofs reused as a consequence of applying our proposal, we have defined the `ClockNoPattern` machine, which models the same behavior as `Clock`, only without explicitly using the State pattern. A comparative analysis of the associated proof activity has allowed us to make an assessment with respect to the total number of proofs reused as a result of applying the proposed formalization of the State pattern in a B project. The number in question has been estimated to $4 + m * n$, where n stands for the number of concrete states and m for the number of requests.

3.3 Summary

Within this chapter we have provided a full formalization of the State design pattern using the B formal method. This covers:

- a definition of the State design pattern in B;
- a highly automatable formalization of its associated reuse process;
- an illustration of its reuse in B, through an average-complexity case study;
- a full correctness proof (using the AtelierB tool) of all B machines involved in the case study, with a significant number of interactive proofs;
- a detailed analysis of the proof activity performed, ended with an assessment with respect to the number of proofs that are saved with each reuse of the proposed pattern formalization in a B project.

While formalizing the pattern's reuse process, we have roughly followed the framework proposed in [45]. However, as compared to the example presented there (a B formalization of the Composite design pattern), our approach to formalizing the State pattern exhibits the following advantages:

- it formalizes the delegation idea, that is the core behavior proposed by the pattern;
- it does not constrain in any way the number of concrete classes to which polymorphism applies (the number of concrete states, in this case); this is due to our proposal of delaying the formalization of the polymorphic behavior to the first step of the reuse process.

Further work is mainly aimed at automating the proposed approach, as well as at investigating our "delay" proposal and its believed advantages in relation to other design patterns.

This contribution has been published in [81]. It relies on previous experimentations with respect to proving the consistency of object-oriented models using the B method, reported in [82].

Chapter 4

Constraint Patterns in Object-Oriented Modeling

In the previous chapter, we have discussed about object-oriented design patterns as reusable assets and how to represent their solutions using the B formal method. Within this chapter, the reusable entities approached are the constraint patterns from object-oriented class modeling. The main contribution reported here consists in the proposal of a new approach concerning the definition of these patterns, driven by the new requirements imposed by MDE on the use of assertions.

4.1 Motivation

The emergence of MDE has imposed the necessity and laid the groundwork for automatic correctness verifications of models and model-based generated applications. Such verifications rely on the use of model assertions. Assertions, such as pre/post-conditions and invariants, are needed to compensate for the narrow expressivity power of diagrammatic modeling languages. In traditional software development, which used to employ models primarily for documentation purposes, correctness and clearness were the only quality attributes required for assertions. In the context of MDE however, which strengthens the need for automatic model correctness checks, assertions should be designed so as to provide efficient support for error diagnosis.

The ever-growing use of assertions following the advent of MDE has led to the identification of several constraint patters, while the necessity of spending less time and avoiding syntax errors in their writing has motivated a few approaches aimed at formalizing and automating the instantiation of these patterns. However, we argue that the constraint patterns' solutions currently available in the literature fail in providing the level of debugging support requested for assertions in the context of MDE.

4.2 Related Work

The most relevant related work on constraint patterns, from which we have started and which we have used as a comparison base for our proposals, is given by the approaches presented in [7, 8] and [84–86].

4.3 A New Approach: MDE-Driven OCL Specification Patterns

The approach proposed within this chapter (to which we refer as being *MDE-driven*) builds on two fundamental principles, namely:

1. In accordance with the model correctness-focused role of assertions within MDE, the solutions of constraint patterns should be designed so as to encapsulate increased debugging (error diagnosing) support;
2. In accordance with the ultimate purpose of models and model-level assertions in MDE (translation into code), as well as with the principles of Design by Contract, the solutions of constraint patterns should be given not only in terms of invariants (as currently the case), but also in terms of matching preconditions.

4.3.1 Approached Patterns. Existing Solutions

Being a rather new research topic compared to their design counterparts, constraint patterns do not currently benefit from a generally-agreed naming or definition in the literature[1]. Therefore, we have proposed and relied on the following definitions within this chapter.

Definition 4.1. A *constraint pattern* embodies a recurrent logical restriction imposed on class models, together with a recommended general specification for it.

Definition 4.2. An *OCL specification pattern* denotes the recommended OCL-based solution of a constraint pattern.

Our proposals are concerned with three constraint patterns previously identified in the literature - *Attribute Value Restriction*, *Unique Identifier*, and *For All*.

Attribute Value Restriction [84] (referred as *Invariant for Attribute Value* in [8]) is an atomic pattern used to abstract various constraints on the value of a given class attribute. Its associated OCL specification pattern[2], as provided in [84], is listed below.

```
pattern AttributeValueRestriction(property:Property, operator,value:OclExpression)=
 self.property operator value
```

Unique Identifier [84] (referred as *Semantic Key* in [8]) captures the situation in which an attribute (a group of attributes) of a class plays the role of an identifier for the class, i.e. the class' instances should differ in their value for that attribute (group). Following, there are its corresponding OCL templates, as proposed in [84]

```
pattern UniqueIdentifier(property:Tuple(Property))=
 self.allInstances()->isUnique(property)
```

and [7]

```
pattern SemanticKey(class:Class, property:Property)=
 class.allInstances()->forAll(i1, i2 | i1 <> i2 implies i1.property <> i2.property).
```

Finally, the *For All* constraint pattern [84] requires every object of a certain collection to fulfill a number of specified restrictions. Below, there is the OCL template-like description of its associated specification pattern.

```
pattern ForAll(collection:OclExpression, properties:Set(OclExpression))=
 collection->forAll(y | oclAND(properties, y))
```

[1]The phrases *constraint patterns* and *OCL specification patterns* are used interchangeably.

[2]All pattern solutions are given in terms of *OCL parameterized templates*, as described in [84].

4.3.2 Proposed Solutions

4.3.2.1 The *For All* Constraint Pattern

Our approach is rooted in a pair of solutions that we have proposed for the *For All* constraint pattern, whose corresponding OCL templates are provided below.

```
pattern ForAll_Reject(collection:OclExpression, properties:Set(OclExpression))=
 collection->reject(y | oclAND(properties, y))->isEmpty()

pattern ForAll_Select(collection:OclExpression, properties:Set(OclExpression))=
 collection->select(y | not oclAND(properties, y))->isEmpty()
```

There are two meta-constraints needed for the patterns above to generate syntactically correct OCL specifications by instantiation, namely:
(FA1) *collection* should be a valid OCL expression which evaluates to a collection type;
(FA2) each of the *properties* should be a valid boolean OCL expression.

We have argued on the advantages of our new solutions over the existing one by means of a relevant modeling example. Moreover, the semantical equivalence of our proposals with the one in use has been confirmed by translation into a B abstract machine, whose proof obligations have been discharged by AtelierB.

4.3.2.2 The *Unique Identifier* Constraint Pattern - Invariant Solutions

In case of the *Unique Identifier* constraint pattern, we have distinguished (for the first time in the literature, to the best of our knowledge) among two possible contexts in which such constraints can occur, providing appropriate invariant solutions for each case. The first refers to the so-called "global" uniqueness - certain models or applications may require all possible instances of a class to differ in their value for a particular attribute. The second captures a "container-relative" uniqueness - a model/application constraint may state that each instance of a class accessible starting from a given container should be uniquely identifiable by the value of a particular attribute, among all instances of the same class from within the same container. We have emphasized that, despite their frequent use for situations matching the "container-relative" case, the existing solutions for *Unique Identifier* correspond to the "global" case exclusively. Moreover, we have pointed to some drawbacks of these solutions, concerning both one's failure in obeying the semantics of invariants and their lack of appropriate debugging support.

"Global" uniqueness case (GUID). For the "global" context, we have proposed the following OCL specification pattern

```
pattern inv_GloballyUniqueIdentifier(class:Class,attribute:Property)=
 class.allInstances()->select(i | i.attribute = self.attribute)->size() = 1,
```

meant to be instantiated as an invariant in the context of *class*. The necessary conditions for ensuring syntactical correctness of the resulting OCL expression are thus the following:
(invGUID1) the pattern instantiation should be performed in the context of *class*;
(invGUID2) *attribute* should be among the attributes of *class*.

"Container-relative" uniqueness case (CUID). In case of "container-relative" uniqueness, the proposed solution is the one below.

```
pattern inv_ContainerRelativeUniqueIdentifier(container,contained:Class,
                      navigationToContained:Property, attribute:Property)=
 let bag:Bag(OclAny) = self.navigationToContained.attribute in
 ForAll_Reject(self.navigationToContained, Set{UniqueOccurrenceInBag(bag,attribute)})
```

Its corresponding meta constraints are as follows:
(invCUID1) the pattern instantiation should be performed as an invariant in the context of

container;
(invCUID2) *navigationToContained* should be a reference in *container* having the type *contained*;
(invCUID3) *attribute* should be an attribute of *contained*.

The OCL specification pattern above employs a newly-proposed constraint pattern - *Unique Occurrence in Bag* (*UOB*), requiring that "A given class attribute has exactly one occurrence in a given bag of elements of the same type". For UOB, we have proposed the following OCL specification pattern.

```
pattern UniqueOccurrenceInBag(bag:OclExpression, class:Class, attribute:Property)=
 bag->count(self.attribute) = 1
```

Its associated meta-constraints state that:
(invUOB1) the pattern instantiation is supposed to be performed as an invariant in the context of *class*;
(invUOB2) *attribute* should be among the attributes of *class*;
(invUOB3) *bag* should be a valid OCL expression that evaluates to a Bag type;
(invUOB4) *attribute* and the elements from *bag* should have the same type.

It is possible to generalize the `inv_ContainerRelativeUniqueIdentifier` pattern, such that the uniqueness constraint, instead of applying to all contained elements, would rather apply to a conveniently filtered subset. Below, we give the OCL pattern that we have proposed in this respect.

```
pattern inv_GenContainerRelativeUniqueIdentifier(container,contained:Class, attribute:Property,
                                                  navigation:Feature, properties:Set(OclExpression)) =
 let subset:Set(contained) = self.navigation->select(e | oclAND(properties,e)) in
 let bag:Bag(OclAny) = subset.attribute in
 ForAll_Reject(subset,Set{UniqueOccurrenceInBag(bag,attribute)})
```

The pattern instantiation is constrained by the following necessary conditions:
(invGCUID1) the instantiation should be performed as an invariant in the context of *container*;
(invGCUID2) *navigation* should be a feature of *container* having the type *contained*;
(invGCUID3) *attribute* should be an attribute of *contained*;
(invGCUID4) each expression from *properties* should stand for a valid OCL boolean expression.

4.3.2.3 The *Unique Identifier* Constraint Pattern - Pre/Post-condition Solutions

Until now, the solutions provided in the literature for constraint patterns such as *Unique Identifier* have only been stated in terms of class invariants. However, we have argued that, in the context of MDE and in accordance to the principles of Design by Contract, these solutions should be enhanced by the addition of appropriate OCL specification patterns for the preconditions of operations that might break the constraints in question. Consequently, we have provided such OCL specification patterns for the preconditions of model operations that could violate an unique identification constraint, considering both the "global" and the "container-relative" types of uniqueness contexts.

"Global" uniqueness case (GUID). For the "global" case, we have proposed the following precondition specification pattern, to be instantiated in the context of *class*`::set`*attribute*`()`. The generated precondition preserves the uniqueness of *attribute*'s values among all *class* instances.

```
pattern preSet_GloballyUniqueIdentifier(class:Class, attribute:Property, parameter:Parameter)=
 ForAll_Reject(class.allInstances(), Set{AttributeValueRestriction(attribute,<>,parameter)})
```

To ensure validity of the generated OCL expression, the following meta-constraints must hold:
(preGUID1) the instantiation is performed in the context of *class*`::set`*attribute*;
(preGUID2) *attribute* is an attribute of *class*;
(preGUID3) *parameter* is the only parameter of `set`*attribute*, having the same type as *attribute*.

"Container-relative" uniqueness case (CUID). For this case, we have proposed the following precondition patterns:

```
pattern preAdd_ContainerRelativeUniqueIdentifier(container,contained:Class,
      navigationToContained:Property, attribute:Property, parameter:Parameter) =
 ForAll_Reject(navigationToContained,
      Set{AttributeValueRestriction(attribute,<>,parameter.attribute)})

pattern preSet_ContainerRelativeUniqueIdentifier(container,contained:Class,
      navigationToContainer,navigationToContained:Property,
      attribute:Property, parameter:Parameter) =
 ForAll_Reject(navigationToContainer.navigationToContained,
      Set{AttributeValueRestriction(attribute,<>,parameter)}).
```

The instantiation of the first pattern above generates a precondition for the `add`*contained*`()` operation of *container*, meant to preserve the uniqueness of *attribute*'s values among all instances of *contained* from within *container*. These instances are accessible by means of the *navigationToContained* reference of *container*. The following meta-constraints should be fulfilled, so as to ensure the validity of the generated OCL expression.
(preAddCUID1) the pattern instantiation context is *container*`::add`*contained()*;
(preAddCUID2) *navigationToContained* is a reference in *container* of type *contained*;
(preAddCUID3) *attribute* is an attribute of *contained*;
(preAddCUID4) *parameter* is the only parameter of `add`*contained*, having *contained* as type;

The second pattern may be instantiated to generate a precondition for the `set`*attribute* operation of the *contained* class, with the purpose of preserving the same "container-relative" uniqueness constraint. Following, there are the meta-constraints corresponding to its parameters.
(preSetCUID1) the pattern instantiation context is *contained*`::set`*attribute*`();`
(preSetCUID2) *navigationToContained* is a reference in *container* of type *contained*;
(preSetCUID3) *navigationToContainer* is a reference in *contained* of type *container*, having mandatory-one multiplicity;
(preSetCUID4) *navigationToContained* is the opposite of *navigationToContainer*;
(preSetCUID5) *attribute* is an attribute of *contained*;
(preSetCUID6) *parameter* is the only parameter of `set`*attribute*, having the same type as *attribute*.

4.3.3 Tool Support and Validation

We have validated our approach and proved its relevance in establishing model correctness by means of the OCLE tool. So as to emphasize its compulsoriness, model correctness has been discussed by analogy to program correctness. From this perspective, establishing model correctness involves both model compilability checks and model testing.

4.3.3.1 Model Compilability Checks

Model compilability assessments involve the evaluation of metamodel WFRs on the model in question; failure to fulfill any of them indicates a bug in the model. Writing the WFRs with model debugging support in mind (as promoted by our proposed specification patterns) considerably facilitates this task, thus speeding up the development process. A proof of concepts has been provided, using a sample UML model for components and an UML 1.5 [58] WFR concerning the uniqueness of names within namespaces. We have illustrated the advantages of writing the WFR in question as an instantiation of the `inv_GenContainerRelative-UniqueIdentifier` specification pattern, as opposed to expressing it as an instantiation of `ForAll`, as originally specified in [58].

4.3.3.2 Model Testing

Model testing involves the evaluation of model-level invariants (business constraint rules or BCRs) on snapshots (domain model instantiations). Assuming the model itself as correct, the detection of any false-positive (wrong snapshot that is accepted, i.e. all BCRs evaluate to true) or false-negative (good snapshot that is denied, i.e. fails to fulfill a particular BCR) points to a logical bug in the BCR expressions. Designing them with debugging support in mind may ease the task considerably[3]. Using a sample model, we have provided a proof of concepts with respect to the advantages derived from applying the `ForAll_Reject` pattern, instead of the classical `ForAll`, in writing a particular BCR.

4.4 Summary

We have hereby proposed a novel approach concerning the definition of constraint patterns for object-oriented class models. This enhances the state of the art with the following:

- a proposal for a clarification of terminology, distinguishing among the concept of *constraint pattern* and that of *OCL specification pattern*;
- a pair of solutions for the *For All* constraint pattern (given as OCL specification patterns) enabling efficient error diagnosis;
- an equivalence proof of the solutions provided for the *For All* pattern with the existing one, conducted by means of the AtelierB prover;
- a new approach with respect to the definition of the *Unique Identifier* constraint pattern, distinguishing among the “global” and the “container-relative” contexts in which the constraint could occur;
- appropriate OCL specification patterns for the *Unique Identifier* constraint pattern, in each of the above-mentioned cases, given in terms of invariants;
- a pair of solutions for the *Unique Identifier* constraint pattern given in terms of pre-conditions;
- proposal of a new atomic constraint pattern, *Unique Occurrence in Bag*;
- a proof of concepts illustrating the validity and usefulness of our approach by means of the OCLE tool, covering both model compilability checks and model testing.

One may argue that the constraints generated by instantiating the proposed OCL specification patterns lack the clearness of the ones previously available in the literature. However, our approach has a great automation potential (either by instantiating the proposed solutions from scratch or by using them through automatic refactorings of old specifications), so this is by no means a limitation.

In fact, further work targets primarily at automating the use of the proposed approach in OCLE. In addition, we further aim at identifying new constraint and OCL specification patterns, along with bringing improvements to the existing ones.

The proposals made in this chapter have been disseminated by means of [27] and [28].

[3]This is imperative when the model to test is, in fact, a metamodel, since, given their reuse potential, metamodels require extensive testing on sizable models.

Chapter 5

The Static Semantics of (Meta)Modeling Languages

While arguing our contribution from the previous chapter, we have presented it as an error diagnosing means, which serves the purpose of achieving model compilability. This chapter further elaborates on the model compilability issue, by emphasizing its compulsoriness, the state of facts in the field and the reasons beneath it, as well as our proposals for improving this state of facts.

Our contribution here is related to the set up of a framework supporting an accurate specification of the static semantics of (meta)modeling languages and enabling efficient model compilability checks. This contribution is twofold. First, we have proposed a set of underlying principles concerning the specification of a static semantics. Second, we have provided enhancements to the definition of the static semantics of the UML metamodel and of three of the best known meta-metamodels - MOF, Ecore and XCore.

5.1 Motivation

5.1.1 The Model Compilability Problem

The MDE paradigm has triggered a major change of focus in the field of software engineering: from programs and programming languages to models and modeling languages. As the artifacts driving a highly automated development process, we argue that the first mandatory requirement imposed on models in the context of MDE should concern compilability. By analogy to program compilability, we have defined *model compilability* as conformance of a model to the abstract syntax and static semantics of its modeling language. The abstract syntax is given by the language metamodel, while the static semantics is expressed by means of the metamodel WFRs.

However, despite the imperative nature of this requirement in the context of MDE, current practice shows that model compilability is rather a goal than a reality. This state of facts is deemed to have both human and technological roots.

5.1.2 Diagnosing the State of Facts Regarding Model Compilability

We have argued that, behind the technological factor pointed out by most papers, the real issues triggering the current state of facts in the area of model compilability are represented by the inadequate specification and deficient validation of the static semantics of (meta)modeling languages. The previous statement is motivated by a detailed analysis that we have performed on the specification and use of assertions within the UML metamodel and three of the best known meta-metamodels - MOF, Ecore and XCore.

We have argued that the solution to the above-mentioned problems consists in the adoption of a rigorous conceptual framework supporting an accurate definition of the static semantics of (meta)modeling languages and enabling efficient model compilability checks. Such a framework should build on a set of well-defined principles regarding the specification of a static semantics.

5.2 Towards a Conceptual Framework Supporting Model Compilability - Principles of a Static Semantics Specification

In order to fully serve its intended purpose of supporting efficient model compilability checks, there are a number of requirements that any set of WFRs should comply with.

The first one is *completeness*; the WFRs should entirely cover the static semantics rules of the language. This entails an intimate understanding of all metamodel-level concepts and how they may be suitably related.

A second mandatory requirement concerns the *availability of an OCL or OCL-like formalization of the entire set of rules.* At least two alternatives to this are offered by the existing approaches: the explicit implementation of rules within the metamodel repository code (the Ecore way) and an attempt at preserving their fulfillment at any time through appropriate implementations of the repository modifiers (the XCore way).

In addition to the above, each WFR specification should itself fulfill a number of quality criteria. The following three are among the most important, the first two being also among the least addressed in the literature.

1. *Detailed, test-driven informal specification.* Preceding the formal WFR expression with a detailed and rigorous informal equivalent is the basic requirement for ensuring correct understandability of the rule. At its turn, the informal specification should be based on meaningful test snapshots needed for its validation (both positive and negative). By analogy to the programming approach known as *test-driven development*, this *test-driven specification* approach provides for a deeper reasoning with respect to the rules, with a positive effect on the correctness/comprehensiveness of their final statements.

2. *Testing-oriented formal specification.* The OCL WFRs should be stated so as to facilitate efficient error diagnosis in case of assertion failure. In this respect, the previous chapter argues on the use of appropriate OCL specification patterns.

3. *Correct and efficient formal specification.* We qualify an OCL WFR as being incorrect in case it fails to satisfy one of the following two criteria. The first criterion concerns compilability, therefore conformance to the OCL standard; the second asks for a full conformance between the OCL specifications and their natural language counterparts.

Another aspect to consider when specifying WFRs refers to *choosing the most appropriate context and shape for each.* This involves understanding the differences between a WFR and a “classical invariant”, as introduced by object-oriented programming (OOP) techniques.

Driven by the above-stated principles, we have evaluated the state of facts regarding the static semantics specification of UML/MOF, Ecore and XCore and we have made specific proposals meant to improve it.

5.3 Enhancements to the Static Semantics of (Meta)Modeling Languages

5.3.1 Enhancements to the Static Semantics of UML/MOF

5.3.1.1 State of Facts. Related Work

UML is now acknowledged as the *de facto* object-oriented modeling language. Moreover, its 1.4.2 release [59] has been adopted as an ISO standard. In turn, MOF is the meta-metamodel standing at the core of the most popular model-driven approach, which is OMG's MDA.

Through the last decade, there have been various papers (e.g. [69], [42], [25]) concerned with the adequacy of the WFRs from the OMG documents. However, although most of the work signaling problems in the static semantics' definition of UML and MOF has focused on the uncompilability of WFRs with respect to OCL, a closer look at the standard specifications (both WFRs and (AOs)) reveals that, apart from compilability issues, the specifications in question enclose several other drawbacks, such as incompleteness, inconsistency, logical errors, as well as shortcomings caused by their deficient testing.

5.3.1.2 Proposed Enhancements

Our proposals for improving the static semantics of the UML/MOF metamodels have been centered around two core metamodeling issues, one related to the semantics of composition and the other to name uniqueness within namespaces.

On the UML/MOF Composition Relationship
As inferable from the OMG documents ([59], [62]) and papers such as [20], composition is a stronger form of association, whose semantics may be captured by the following constraints:

[C1] *Only binary associations can be compositions*;
[C2] *At most one end of an association may specify composition* (*a container cannot be itself contained by a part*);
[C3] *An association end specifying composition must have an upper multiplicity bound less or equal to one* (*a part is included in at most one composite at a time*);
[C4] *Since the composite has sole responsibility for the disposition of its parts, the parts should be accessible starting from the container* (*navigation from container to parts should be enforced*).

We have investigated coverage of this semantics through WFRs in both the 1.x and 2.x releases of the UML/MOF metamodels. Our study reveals the incompleteness of the WFRs set enclosing the semantics of composition in both UML/MOF 1.x and 2.x, and emphasizes some inconsistencies among the informal statements and the OCL WFRs related to composition in UML/MOF 2.x. The solutions proposed stem from an analysis supported by the use of the *test-driven specification* principle. The possibility of expressing the same informal constraint in different contexts and under different shapes, as well as the criteria involved in choosing the right ones have been also discussed and exemplified.

UML 1.x. The UML 1.x metamodel excerpt defining associations is illustrated by the diagram in Figure 5.1. With respect to enforcing the rules [C1] to [C4], the standard specification only covers the first three of them. The OCL WFRs for [C1] and [C2] are stated in the context of `Association`,

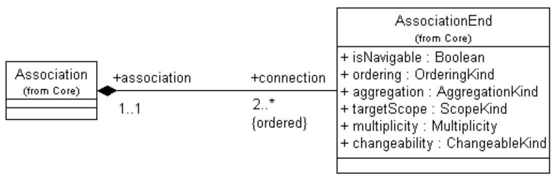
Association
(from Core)
+association
1..1
+connection
2..*
{ordered}
AssociationEnd
(from Core)
+ isNavigable : Boolean
+ ordering : OrderingKind
+ aggregation : AggregationKind
+ targetScope : ScopeKind
+ multiplicity : Multiplicity
+ changeability : ChangeableKind

Figure 5.1: UML 1.4 `Associations`

while the one for [C3] is written in the context of `AssociationEnd`.

Our proposals for formalizing the missing constraint, [C4], are provided in Listing 5.1. Favoring one over the others is a decision that depends on both language semantics and available tool facilities.

```
context AssociationEnd inv validCompositionNavigability1:
 self.aggregation = #composite implies
 self.association.connection->any(ae | ae <> self).isNavigable

context AssociationEnd inv validCompositionNavigability2:
 self.association.connection->exists(ae | ae <> self and
  ae.aggregation = #composite) implies self.isNavigable

context Association inv validCompositionNavigability3:
 self.connection->exists(ae | ae.aggregation = #composite) implies
 self.connection->any(ae | ae.aggregation <> #composite).isNavigable
```

Listing 5.1: Proposed WFR expressions for [C4] in MOF and UML 1.x

From the three alternative WFRs above, the only one fully complying with the UML 1.x composition semantics is the last one (written in the context of `Association`). According to this, the standard WFR for [C3] can be itself rephrased in the `Association` context, as follows.

```
context Association inv validCompositionUpperBound:
 self.connection->exists(ae | ae.aggregation = #composite) implies
 self.connection->any(ae | ae.aggregation = #composite).multiplicity.max = 1
```

Listing 5.2: Proposed WFR for [C3] in MOF and UML 1.x

The WFR from Listing 5.2 and the last WFR from Listing 5.1 may also be combined within a single OCL expression, as shown below. However, this has the disadvantage of requiring partial evaluation in case of assertion failure, so as to identify precisely which expression in the conjunction has caused the failure.

```
context Association inv validCompositionUpperAndNavigability:
 self.connection->exists(ae | ae.aggregation = #composite) implies
 (self.connection->any(ae | ae.aggregation = #composite).multiplicity.max = 1 and
  self.connection->any(ae | ae.aggregation <> #composite).isNavigable)
```

Listing 5.3: Proposed WFR for both [C3] and [C4] in MOF and UML 1.x

UML/MOF 2.x. As illustrated by Figure 5.2, the UML 2.x Infrastructure brings some changes in the definition of associations, changes that are also reflected in the MOF 2.0 specification. Unfortunately, concerning the semantics of composition, things seem to have worsened compared to the 1.x specifications. From those four constraints expressing the semantics of composition stated at the beginning of this section, only [C1] has a correct OCL equivalent within the specification documents. As for the others, [C4] seems to be missing, [C3] has some consistency-related drawbacks, and [C2] appears in the MOF 2.0 specification rather as an informal precondition of the `create` operation from the `Reflection::Factory` package. We have proposed appropriate OCL WFRs for each of the constraints [C2] to [C4].

The natural context for [C2] is represented by the `Association` metaclass. Its corresponding OCL invariant is given below.

```
context Association inv atMostOneCompositeEnd:
 self.memberEnd->select(p | p.isComposite)->size() <= 1
```

Listing 5.4: Proposed WFR for [C2] in MOF and UML 2.x

The rules [C3] and [C4] can be stated both in context of `Association` and `Property`, as shown in Listings 5.5 and 6.1.

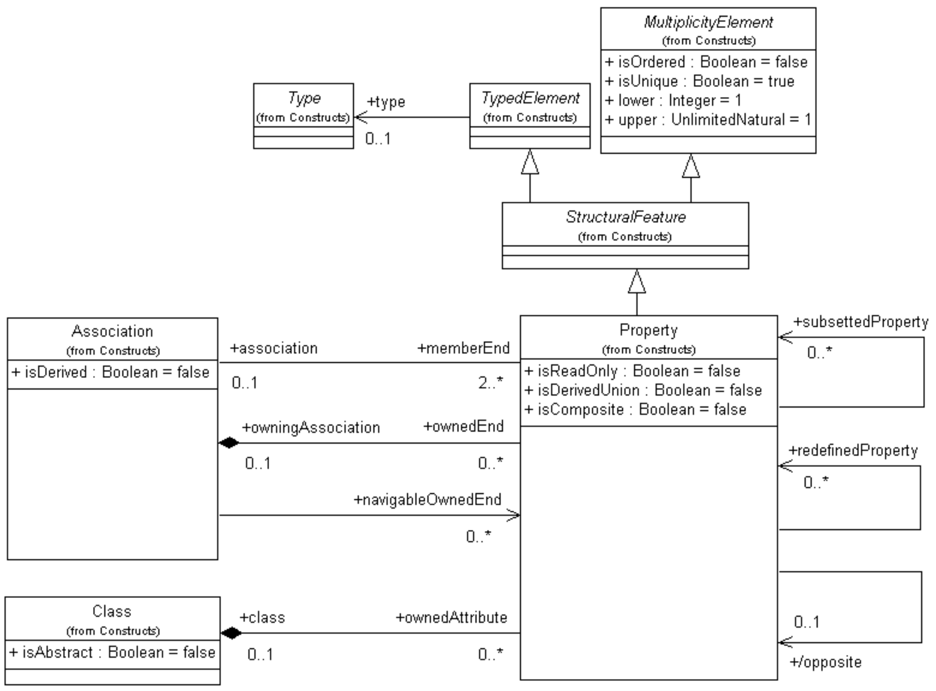

Figure 5.2: MOF 2.0 and UML 2.3 `Associations`

```
context Association inv validCompositionMultiplicity1:
 self.memberEnd->exists(p | p.isComposite) implies
 self.memberEnd->any(p | not p.isComposite).upper = 1

context Property inv validCompositionMultiplicity2:
 self.isComposite and self.association->notEmpty() implies
 self.association.memberEnd->any(p | p <> self).upper = 1
```

Listing 5.5: Proposed WFRs for [C3] in MOF and UML 2.x

```
context Association inv validCompositionNavigability1:
 self.memberEnd->exists(p | p.isComposite) implies
 self.memberEnd->any(p | p.isComposite).isNavigable()

context Property def: isNavigable() : Boolean =
(self.class->notEmpty()) xor (self.owningAssociation->notEmpty() and
 self.owningAssociation.navigableOwnedEnd->includes(self))

context Property inv validCompositionNavigability2:
  self.isComposite and self.owningAssociation->notEmpty() implies
  self.owningAssociation.navigableOwnedEnd->includes(self)
```

Listing 5.6: Proposed WFRs for [C4] in MOF and UML 2.x

On Forbidding Name Clashes within Namespaces

We have uncovered three types of errors occurring within the standard WFR and AOs prohibiting name clashes within namespaces: syntactic errors, logical ones, as well as faults coming from failure to provide the information required for error diagnosis in case the assertion gets violated. The solution proposed for the latter case involves the use of an appropriate OCL specification pattern.

5.3.2 Enhancements to the Static Semantics of Ecore

5.3.2.1 State of Facts

Ecore is the meta-metamodel of EMF and the best known EMOF (Essential MOF) implementation. However, Ecore does not match EMOF exactly. On the one side, the approach taken with Ecore is more pragmatic and implementation-oriented. On the other side, starting with EMF 2.3, Ecore includes constructs for modeling with generics [53]; this is considered to be a departure from EMOF, which does not currently provide such support.

The Ecore repository includes a set of WFRs implemented directly in Java, within the EcoreValidator class. However, even though EMF integrates an OCL plugin (MDT-OCL [38]) and there is a functional approach available enabling the automatic translation of OCL assertions into Java code [34], we have not found any OCL equivalent of the implemented constraints. Moreover, to the best of our knowledge, there is a single paper in the literature approaching the OCL formalization of Ecore WFRs. The paper in question [44] deals solely with a few rules regarding generics.

5.3.2.2 Proposed Enhancements

Driven by the previously reported state of facts and in accordance to one of the principles exposed in Section 5.2, regarding the necessity of an OCL(-like) formalization of a static semantics, we have defined, tested and validated in OCLE a comprehensive set of OCL WFRs for the Ecore meta-metamodel. The entire set of rules can be found at [4]. Within this section, we have approached some WFRs related to the Ecore generics. Choosing these particular constraints for exemplification purpose is due to both their complexity level (since they are non-trivial WFRs) and the fact that they allow a close comparison with related work described in [44].

The Ecore Generics. Figure 5.3 shows that part of the Ecore meta-metamodel that ensures the generic modeling support it provides. Similar to Java (whose generics model has inspired the one in Ecore), Ecore enables generic type and operation declarations, as well as generic type instantiations (also known as parameterized types). The supporting metamodel concepts are `ETypeParameter` and `EGenericType` respectively. Assuming a closer familiarity of the reader with Java than Ecore, we have explained both concepts by means of relevant model examples, starting from their Java equivalents. An `ETypeParameter` instance stands for a type parameter used by either a generic classifier or a generic operation declaration. An `EGenericType` instance may denote one of the following: a type parameter reference, a (generic) type invocation, or a wildcard. `EGenericType` instances can play various roles in an Ecore model, each kind of usage being constrained by corresponding WFRs. Such an instance can be exactly one of the following: a generic supertype

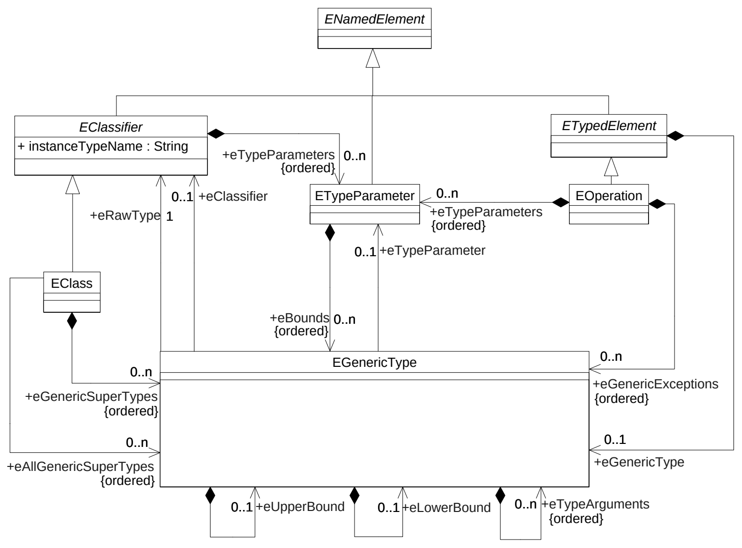

Figure 5.3: Ecore generics

of a class, the type of a typed element (attribute, reference, operation, parameter), a bound of a type parameter, one of the type arguments of a generic type invocation, the upper or lower bound of a wildcard, or an exception type.

On a WFR for Generics in Ecore

Within this section, we have provided an OCL specification for the following informal constraint: "*Assuming that a generic type denotes a type parameter reference, the referenced type parameter must be in scope and must not be a forward reference. The type parameter is in scope if its container is an ancestor of this generic type within the corresponding Ecore containment tree*". The proposed OCL WFRs are illustrated in Listing 5.7; the AOs involved are provided in Listings 5.8 and 5.9.

```
context EGenericType
 -- The referenced type parameter must be in scope, i.e.,
 -- its container must be an ancestor of this generic type ...
 inv InScopeTypeParameter:
  self.isTypeParameterReference() implies
  self.ancestors()->includes(self.eTypeParameter.eContainer())

context EGenericType
 -- ... and must not be a forward reference.
 inv NotForwardReference:
  (self.isTypeParameterReference() and self.isUsedInATypeParameterBound())
  implies
  (let refParameter : ETypeParameter = self.eTypeParameter
   let boundedParameter : ETypeParameter = self.boundedTypeParameter()
   let paramSeq:Sequence(ETypeParameter)=
     (if refParameter.eContainer().oclIsKindOf(EClassifier)
      then refParameter.eContainer().oclAsType(EClassifier).eTypeParameters
      else refParameter.eContainer().oclAsType(EOperation).eTypeParameters
      endif)
   let posRefParameter : Integer = paramSeq->indexOf(refParameter)
   let posBoundedParameter : Integer =
     (if paramSeq->includes(boundedParameter)
      then paramSeq->indexOf(boundedParameter)
      else -1
      endif)
   in
   ((posBoundedParameter <> -1) implies
    ((posRefParameter < posBoundedParameter) or
     ((posRefParameter = posBoundedParameter) and (not boundedParameter.eBounds->includes(self)))
    )
   )
  )
```

Listing 5.7: Proposed OCL WFRs for `EGenericType` prohibiting invalid type parameter references

```
context EGenericType def: isTypeParameterReference() : Boolean =
 not self.eTypeParameter.isUndefined()

context EObject def: ancestors() : Set(EObject) =
 let empty : Set(EObject) = Set{} in
 if self.eContainer().isUndefined() then empty
 else Set{self.eContainer()}->union(self.eContainer().ancestors())
 endif

context EObject def: eContainer() : EObject = oclUndefined(EObject)

context EParameter def: eContainer() : EObject = self.eOperation
--analogous definitions of eContainer() for EPackage, EClassifier, EStructuralFeature, EOperation

context ETypeParameter def: eContainer() : EObject =
 let classifier = EClassifier.allInstances()->any(c | c.eTypeParameters->includes(self))
 in (if not classifier.isUndefined() then classifier
     else EOperation.allInstances()->any(o | o.eTypeParameters->includes(self))
     endif)
--analogous definition of eContainer() for EGenericType
```

Listing 5.8: Query operations used by `InScopeTypeParameter`

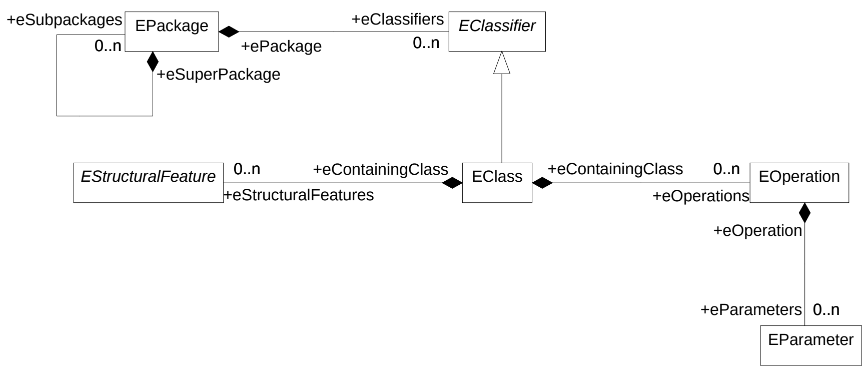

Figure 5.4: Ecore containment relationships

```
context EGenericType def: isUsedInATypeParameterBound() : Boolean =
 self.ancestors()->exists(o | o.oclIsTypeOf(ETypeParameter))

context EGenericType def: boundedTypeParameter() : ETypeParameter =
 self.ancestors()->any (o | o.oclIsTypeOf(ETypeParameter)).oclAsType(ETypeParameter)
```

Listing 5.9: Query operations used by `NotForwardReference`

5.3.2.3 Related Work

As previously pointed out, the only available benchmarks for comparing our work with have been the EMF implementation of the EcoreValidator and the paper [44].

The EMF EcoreValidator. We have argued that the availability of an OCL formalization of the Ecore WFRs has at least two major advantages. On the one side, OCL is the standard language for expressing such rules, the OCL assertions being, by nature, more compact and intelligible compared to their equivalents in a programming language. On the other, in the context of MDE, there is tool-support enabling the automatic translation of OCL expressions into corresponding programming-language code (OCLE and [34] are two notable examples).

In addition, even though the Ecore generics declare to closely mirror their Java correspondents, there are some discrepancies among the Java specification of generics and the rules implemented by the EcoreValidator. As an example, the following rule concerning the correct declaration of generic types and methods is enforced by the Java Language Specification [46] (pp. 50), while missing from the EcoreValidator implementation: "*Type variables have an optional bound, T & I_1 ... I_n. The bound consists of either a type variable, or a class or interface type T possibly followed by further interface types I_1, ..., I_n. ... It is a compile-time error if any of the types I_1 ... I_n is a class type or type variable. The order of types in a bound is only significant in that ... and that a class type or type variable may only appear in the first position.*". For this rule, we have proposed the following OCL WFR.

```
context ETypeParameter
 inv ValidBounds:
  -- If a type parameter has bounds and the first bound is a
  -- type parameter reference, then there are no other bounds.
  (self.eBounds->notEmpty() and self.eBounds->first().isTypeParameterReference() implies
   self.eBounds->size() = 1)
  and
  -- If there are at least two bounds, then all
  -- except (maybe) the first one should refer to interface types.
  (self.eBounds->size() >= 2 implies Sequence{2..self.eBounds->size()}->reject(i |
                                     self.eBounds->at(i).hasInterfaceReference())->isEmpty())
```

Listing 5.10: Proposed OCL WFR for `ETypeParameter` prohibiting invalid type parameter bounds

The above WFR makes use of the following query operations:

```
context EGenericType
 def: hasClassifierReference() : Boolean = not self.eClassifier.isUndefined()

 def: hasClassReference() : Boolean =
  self.hasClassifierReference() and self.eClassifier.oclIsTypeOf(EClass)

 def: hasInterfaceReference() : Boolean =
  self.hasClassReference() and self.eClassifier.oclAsType(EClass).interface

 def: isTypeParameterReference() : Boolean = not self.eTypeParameter.isUndefined
```

Listing 5.11: Query operations used by `ValidBounds`

The Approach Taken in [44]. This paper proposes a number of OCL WFRs meant to check the well-formedness of generic type declarations and that of parameterized types. We have analyzed these rules with respect to both their intended purpose and our goal of defining a complete set of OCL WFRs for Ecore. We argue that, even though they are a good starting point and comparison base, the WFRs in question have various shortcomings, triggered by incompleteness, redundancy and the use of `forAll`. As an example, the WFRs meant to check the well-formedness of a generic type declaration only constrain the bounds of a type parameter to reference parameters from within the same type declaration, without prohibiting forward referencing (as opposed to our proposal from Listing 5.7). Moreover, the rules are only focused on the correct definition and instantiation of generic classifiers; generic operations are not taken into account and nor are the various possible usages of a generic type.

5.3.3 Enhancements to the Static Semantics of XCore

5.3.3.1 State of Facts

XCore is the bootstraping kernel of XMF, a MOF-like metamodeling facility focused on capturing all aspects of a language definition - abstract syntax, concrete syntax and semantics. Unlike MOF though, XMF is completely self-defined and provides platform-independent executability support by means of an executable OCL dialect named XOCL.

The official XMF reference [31] acknowledges the value of WFRs and promotes their use in defining the static semantics of modeling languages. Still, the document does not describe (neither informally, nor formally) any WFR for the XCore meta-metamodel. As regarding the XMF implementation, this does only include two explicit XOCL constraints. Apart from these, there are a number of

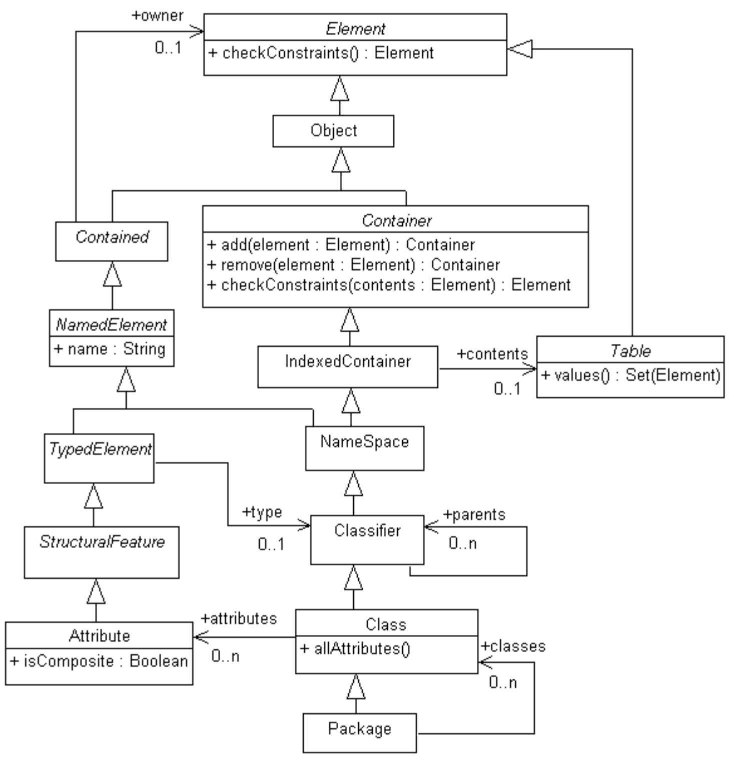

Figure 5.5: An excerpt of the XCore meta-metamodel

other rules intended to be preserved by means of a suitable implementation of modifiers. We have identified various deficiencies of this latter technique. Moreover, even with this approach, the XMF implementation does not cover some of the elementary WFRs that are compulsory for object-oriented concepts, such as avoiding name conflicts among features of the same class/classifier, or the proper management of contained-container dependencies.

5.3.3.2 Proposed Enhancements

As a solution to the above-mentioned problems, we have proposed a set of XOCL WFRs for the XCore meta-metamodel, which we have tested on relevant model examples. The entire set of rules, together with the corresponding tests, can be consulted at [4]. Within this section, we have only discussed two relevant examples, related to name uniqueness and containment respectively.

On Avoiding Name Conflicts Among Owned and Inherited Members

As previously stated, one of the WFRs not covered by the XMF implementation concerns the name conflict among an attribute owned by the current class and attributes inherited from its ancestors. In order to identify such invalid models, we have proposed a WFR with the following informal statement: *There should not be any name conflicts among the attributes owned and inherited by a class.* Listing 5.12 provides its formal XOCL equivalent. The referenced part of the XCore metamodel is illustrated in Figure 5.5.

```
context Attribute @Constraint uniqueName
 let allAtts = self.owner.allAttributes() then
     sameNameAtts = allAtts->excluding(self)->select(att |
                    att.name.asSymbol() = self.name.asSymbol())
 in sameNameAtts->isEmpty()
 end

fail
  let sameNameAtts = self.owner.allAttributes()->excluding(self)->
                     select(att | att.name.asSymbol() = self.name.asSymbol()) then
      msg = "Attribute name duplication! Inherited/owned attributes of " +
            self.owner.toString() + " with the same name: "
  in @While not sameNameAtts->isEmpty() do
       let att = sameNameAtts->sel
       in msg := msg + att.owner.toString() + "::" + att.toString() + "; ";
          sameNameAtts := sameNameAtts->excluding(att)
       end
     end;
     msg
  end
end
```

Listing 5.12: Proposed XOCL WFR prohibiting name conflicts among owned and inherited attributes of a class

On the XCore Containment Relationship

As shown by the metamodel excerpt in Figure 5.5, XCore represents containments explicitly, by means of the `Contained` and `Container` abstract metaclasses. According to the commonly-agreed semantics of containments, we argue that there are two fundamental rules that any XCore model should fulfill in this respect. These rules correspond to the UML composition constraints [C3] and [C2] respectively, as stated in Subsection 5.3.1.2.

[C1'] *A part should belong to a single container at a given time.*

[C2'] *A container cannot be itself contained by one of its parts.*

For the first rule, we have proposed the XOCL WFR from Listing 5.13, with the following informal statement: "*All* `Contained` *instances that belong to the contents table of an* `IndexedContainer` *should have that container as owner.*". In fact, the proposed constraint captures anomalies of a more general nature than just parts simultaneously belonging to at least two different containers (e.g. parts belonging to the `contents` table of a container and having no `owner` set at all).

```
context IndexedContainer @Constraint validOwnerForContents
  self.contents.values()->select(v | v.oclIsKindOf(Contained) and
         v <> null)->reject(v | v.owner = self)->isEmpty()

  fail "The elements from " + self.contents.values()->select(v | v.oclIsKindOf(Contained)
         and v <> null)->select(v | v.owner <> self).toString() +
       " should have " + self.toString() + " as the owner!"
end
```

Listing 5.13: Proposed XOCL WFR for containment constraint [C1']

For the second rule, we have proposed the XOCL WFR below, which applies to all indexed containers, except for the `Root` namespace (in XMF, `Root` is the global namespace in which everything is contained, itself included). Its informal equivalent states that "*No* `IndexedContainer` *different from the* `Root` *namespace can be owned by one of its parts.*".

```
context IndexedContainer @Constraint notOwnedByPart
  (self <> Root and self.oclIsKindOf(Contained)) implies
   self.contents.values()->select(v | self.owner = v)->isEmpty()

   fail "This container is owned by each of its parts from " +
         self.contents.values()->select(v | self.owner = v).toString()
end
```

Listing 5.14: Proposed XOCL WFR for containment constraint [C2']

5.4 Summary

Within this chapter, we have contributed to the set up of a framework supporting an accurate specification of the static semantics of (meta)modeling languages and enabling efficient model compilability checks. This improves the state of the art with the following:

- a detailed analysis of model compilability by analogy to program compilability;
- a set of underlying principles concerning the specification of a static semantics;
- a number of enhancements to the static semantics specification of the UML metamodel and MOF meta-metamodel;
- a comprehensive set of OCL WFRs and AOs (additional operations) meant to enhance the static semantics specification of the EMF Ecore meta-metamodel;
- a comprehensive set of XOCL WFRs and AOs meant to enhance the static semantics specification of the XMF XCore meta-metamodel.

Further work aims primarily at extending the set of corrected/added rules, so as to cover the entire UML/MOF 2.x metamodels. A complementary issue would be that of investigating the problems of consistency and redundancy in a given set of constraints (WFRs in particular). The translation into a formal language, such as B, and the use of the associated prover tools may help in this respect. Another direction of future work could be the identification and formalization of a core set of constraint patterns occurring in meta-metamodels.

The principles, together with the enhancements to the static semantics of UML and MOF have been published in [26], the proposals concerning the static semantics of Ecore in [80], while those for XCore in [76] and [75]. The contributions in this chapter build on previous results, reported in [29].

Chapter 6

The Specification of Software Components

In the previous chapters, we have focused on high-level software abstractions (such as design patterns, constraint patterns and metamodels) and means of providing appropriate formal foundations for their reuse. In this chapter, we report on two contributions in the area of software components. The first is a contribution to a reverse engineering approach aimed at extracting structural and behavioral abstractions from component system implementations. This has been established as part of an ECO-NET international project [1, 10]. The second is intended to set the bases of a framework able to support a full contractual specification of software components, with a special emphasis on semantic contracts.

6.1 Towards Reverse Engineering Component Specifications from Code

6.1.1 Motivation and Related Work

The motivation for the research reported in this section is given by the existing gap among academic and industrial component-based approaches [10]. Namely, the academic ones (e.g. Fractal, SOFA, Kmelia, KADL) focus on specification; they define abstract, hierarchical component models, enriched with formalisms for representing behaviors and means of checking various system properties, such as safety or liveness. Some also cover refinement and code generation, but many of them do not address implementation issues at all. In turn, the industrial approaches (e.g. CCM, EJB, OSGI, .NET) are focused on implementation; they offer strong and mature run-time infrastructures, but define only flat components and lack the model checking support needed for ensuring safe component reuse.

Such a discrepancy triggers a lack of traceability among component specifications and implementations. In turn, this makes it hard to ensure at the implementation level the fulfillment of properties proved for the associated abstract models and leads to maintenance problems (such as *architectural erosion* or *architectural drift*). Under MDE, the solutioning of such a gap can be approached by either direct ([65, 67]) or reverse engineering techniques ([14, 64]). In this context, the goal of the ECO-NET project [1][1] has been to contribute

[1]An international project to which we have taken part. The project, joining four research groups - the french teams COLOSS and OBASCO from LINA, University of Nantes, the czech group DSRG from Charles University in Prague, and the romanian team LCI from Babes-Bolyai University of Cluj-Napoca - has been funded by the French Ministry of Foreign Affairs.

to the reverse engineering way, by developing techniques and tools for extracting structural and behavioral abstractions from component code [10].

6.1.2 General Approach

The project has been aimed at establishing a link between component code (*the concrete* or *the source model*) and component specifications (*the abstract* or *the target model*). In order to properly cope with the complexity of such a process, the concrete model has been limited to Java source code and the abstract one to instantiations of component models covering both structural and behavioral features, such as SOFA [22], Kmelia [9], KADL [65], or Fractal [21].

The main contribution reported by the project consists in the set up of a reverse-engineering framework providing the following features:

- A *Common Component MetaModel* (*CCMM*) that addresses both the issue of handling several component models (SOFA, Kmelia, KADL, Fractal) in a generic way and that of storing the required traceability links among the source and target models. This metamodel is used to generate the API (Application Programming Interface) shared by the following two abstraction processes;
- A *structural abstraction process* (process *SA*) and a corresponding tool used to extract architectural information from the source model;
- A *behavioral abstraction process* (process *SB*) and a corresponding prototype tool, which uses the source model and the output of process SA in order to extract behavioral specifications of components.

Within the envisioned engineering process, the required traceability links are stored both in the target model (by means of specialized attributes defined for the CCMM metaclasses) and the source one (by means of dedicated Java annotations that have been defined).

6.1.3 The Common Component MetaModel (CCMM)

Our contribution in the project has been related to the definition of the CCMM metamodel, as well as its validation and generation of the associated API. This metamodel had to satisfy at least three basic constraints. The first constraint concerns genericity; the metamodel had to abstract over different concrete component models (SOFA, Kmelia, KADL, Fractal), by gathering a common set of concepts and postponing specific concepts to concrete model mappings. Second, it had to include means for managing the tight connections between model elements and the corresponding (Java) features implementing them. Third, the metamodel had to be fully specified, including all the necessary well-formedness rules and useful query

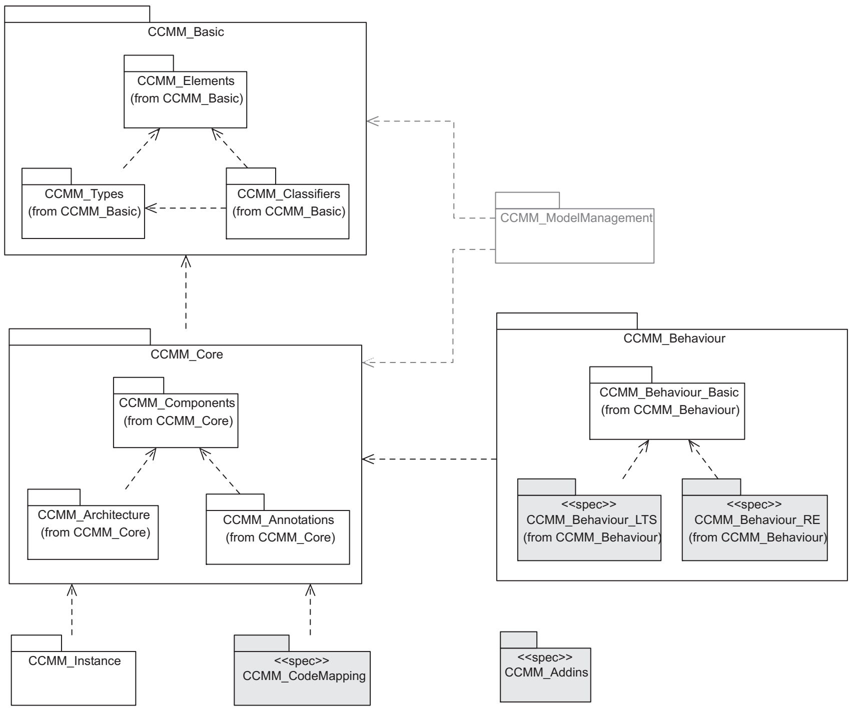
CCMM_Basic
CCMM_Elements (from CCMM_Basic)
CCMM_Types (from CCMM_Basic)
CCMM_Classifiers (from CCMM_Basic)
CCMM_ModelManagement
CCMM_Core
CCMM_Components (from CCMM_Core)
CCMM_Architecture (from CCMM_Core)
CCMM_Annotations (from CCMM_Core)
CCMM_Behaviour
CCMM_Behaviour_Basic (from CCMM_Behaviour)
<<spec>> CCMM_Behaviour_LTS (from CCMM_Behaviour)
<<spec>> CCMM_Behaviour_RE (from CCMM_Behaviour)
CCMM_Instance
<<spec>> CCMM_CodeMapping
<<spec>> CCMM_Addins

Figure 6.1: CCMM v1.1

operations. Moreover, appropriate tool support for metamodel testing and repository code generation had to be ensured.

The above-mentioned requirements have led to the definition of the Common Component MetaModel (CCMM), whose overall architecture is illustrated in Figure 6.1. The thesis includes a detailed description of all packages used for the generation of the CCMM API, including concepts, relationships, WFRs and AOs.

As an example, Figure 6.2 illustrates the contents of the `CCMM_Components` package, which is the core CCMM package, providing a black-box definition of components. According to the diagram, a `ComponentType` is a black-box entity, defined as a special-

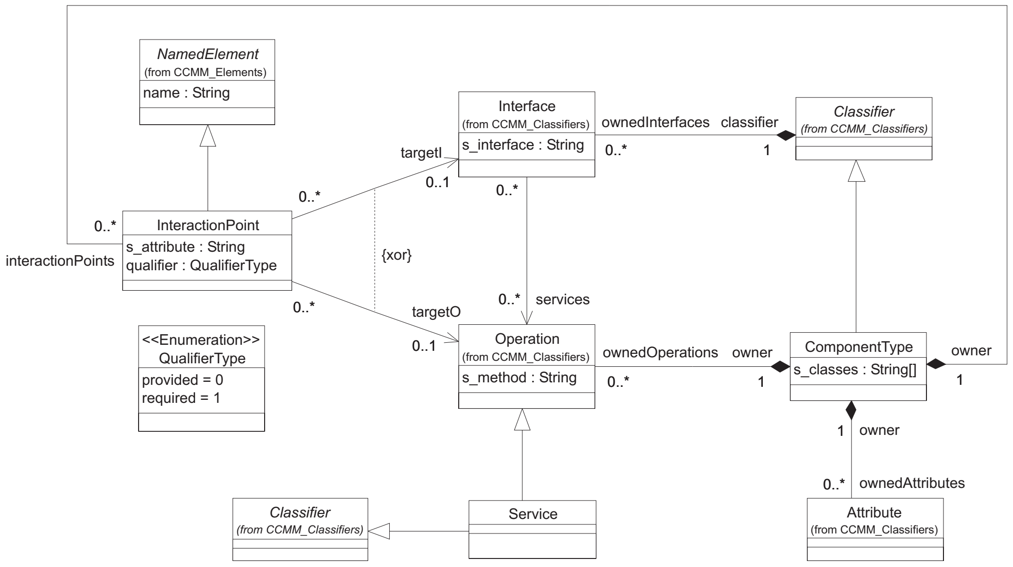

Figure 6.2: `CCMM_Components` package

ization of `Classifier`. Any `ComponentType` interacts with the environment through a number of `InteractionPoints`. Each of these expresses either a provision or a requirement, and may target either an `Interface` or an `Operation`. The type of the target depends on the concrete component model considered; it is an interface in case of SOFA, and an operation in case of Kmelia. Nevertheless, all `interactionPoints` owned by a certain `ComponentType` should have the same target type, fact expressed by means of the `consistentInteractionPoints` OCL WFR below.

```
context ComponentType
 inv consistentInteractionPoints:
  -- if at least one interaction point targets an interface,
  self.interactionPoints->exists(ip:InteractionPoint | ip.targetsInterface()) implies
  -- all interaction points should target interfaces
  self.interactionPoints->reject(ip:InteractionPoint | ip.targetsInterface())->isEmpty()

context InteractionPoint::targetsInterface():Boolean
 body: self.targetO.oclIsUndefined()
```

6.1.4 Tool Support and Validation

We have implemented CCMM as an Ecore metamodel, in order to be able to benefit from the strong EMF tool support. Based on it, we have generated the associated repository code (including the full implementation of WFRs and AOs), as well as a corresponding tree-like component model editor. Both repository and editor have been made available as Eclipse plugins. Validation of the proposed approach has been performed on the non-trivial CoCoME benchmark [68]. This includes tests and validation of the CCMM metamodel, with all its associated WFRs and AOs. Full details regarding the processes, toolset and experimentations leaded can be found at the ECONET SVN repository [2].

6.2 ContractCML - A Contract-Aware Component Modeling Language

6.2.1 Motivation and Related Work

A safe black-box reuse of any software component requires it to be accompanied by a comprehensive contractual specification of its required and provided services. Four contract levels have been identified to apply to software components [15], namely: *basic*, *behavioral*, *synchronization*, and *quality-of-service*. A first level basic contract introduces the so-called *syntactic specification* [33] of a software component; this merely includes signatures of required and provided services. The second level behavioral contract enriches the previous one, by adding a *semantic specification* [33] of services; this comprises a precise functional description of each service, including legal conditions under which it should be invoked (pre-conditions), as well as expected effects of its execution (post-conditions). The second level contracts specify behavior in terms of individual services, regarded as atomic operations executing in a sequential context. As opposed to this, level three synchronization contracts describe the global behavior of component objects. This includes dependencies between services pertaining to a component, such as sequence, parallelism or shuffle, in a distributed concurrent environment. Finally, level four contracts cover non-functional component properties (e.g. maximum response delay, average response, precision of results).

Although the compulsoriness of a semantic specification for software components is unanimously acknowledged, the only form of specification employed by dedicated industrial component models like EJB, COM, or CCM remains the syntactic one [33]. Some improvements have been brought by academic component models, some of which have introduced facilities for describing component behaviors. The Fractal and SOFA component models, for example, use behavior protocols in this purpose [66]. Nevertheless, this kind of specification rather fits level-three contracts, while missing level-two semantic information. Nothing can be said about the effects of invoking a service from an interface, except for what might be assumed from its own name, or the names and types of its parameters.

As the four types of contracts provide complementary information, a thorough component specification should include them all. Moreover, since we see them as interdependent (accomplishing a certain contract level stands as a precondition for the ones above it), skiping an intermediary level would result in essential loss of information.

In this context, our general aim has been that of setting the bases of a framework supporting a full (four-level) contractual specification of software components, thus enabling component interoperability checks. Such a framework should be based on a corresponding domain-specific modeling language (DSML). The motivation for this is twofold. Firstly, since DSMLs are tailored to particular problem domains, the models that use them are easier to understand and manage compared to those created with a general purpose modeling language (e.g. UML). Secondly, our choice enables us to benefit from a powerful tool support. Indeed, EMF [36], GMF [37], oAW [5] and XMF-Mosaic [3] are MDE meta-tools providing rich functionalities. Namely, these meta-tools assist users in specifying, testing and validating DSMLs at all language levels: abstract syntax, concrete syntax, and semantics. Moreover, having a validated DSML, the above-mentioned frameworks provide support in developing dedicated (domain specific) modeling tools.

6.2.2 The ContractCML Metamodel

Within this section, we have introduced ContractCML (Contract Component Modeling Language), the DSML that we have proposed as the backbone of our approach. ContractCML is a hierarchical component modeling language (as opposed to a flat one), since it allows defining and managing not only primitive components, but also composed ones. But most important, as its name shows, it is a contract-aware component modeling language, allowing the specification of component-related contracts. At the moment, it covers the first two levels[2], but its extensible architecture facilitates the adding of new levels in a non-invasive way.

6.2.2.1 Metamodel Overview

The ContractCML metamodel has been designed in a modular style, starting with basic syntactic component concepts, on which semantic and architectural aspects have been added. This has resulted in a loosely coupled, highly extensible architecture. The metamodel packages, as well as their inter-dependencies, are illustrated in Figure 6.3. The `Basic` package contains general-purpose, elementary modeling concepts. Depending on `Basic`, the `InterfaceSpec` package wraps metaclasses that allow defining the syntactic and semantic contracts published by software components. Further, `BlackBoxComponent` includes concepts offering a client's view of components. From this perspective, each component has a corresponding component type, that may be thought of as the collection of all ports required and provided by the component in question; each port defines an interaction point with the environment and is typed by an interface. The `Architecture` package bundles concepts used for describing component assemblies; an architecture owns a set of component instances and a set of assembly bindings among them. Finally, having the black box and architectural concepts defined, the `WhiteBoxComponent` overpasses the client's perspective, offering a deeper, architectural view of components. Components are classified as primitive or composed, composed components owning an architecture and a number of delegation bindings.

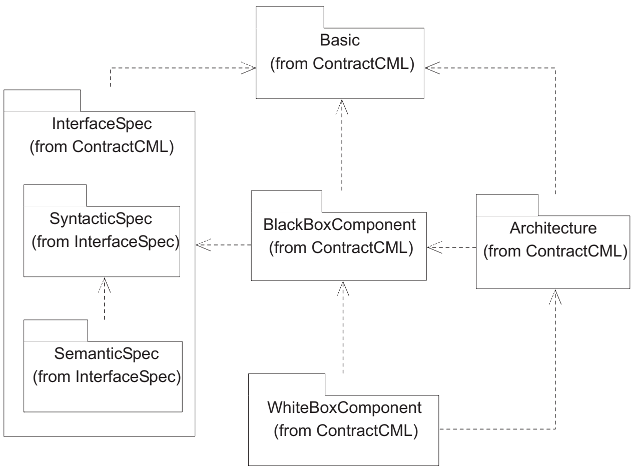

Figure 6.3: ContractCML architecture

This section details the semantics of the metaconcepts contained in these packages, including all WFRs and AOs. Among the most relevant WFRs are those expressing the semantics of assembly and delegation bindings.

6.2.2.2 Basic Contracts. Syntactic Specification of Interfaces

The elements from within `SyntacticSpec`, one of the two `InterfaceSpec` packages, describe component interfaces from a syntactic perspective. An interface is a named element that consists of a collection of operations, each operation having itself a name, an ordered list of parameters and, possibly, a return type; each parameter has a name, a type and a sort, with the latter specifying the dataflow direction: input, output, both, or unknown. By means of the metaclasses defined in this package (`Interface`, `Operation`, `Parameter`,

[2]We have focused on the second level, that is missing from the current component models.

ParameterSort), ContractCML is allowed to express basic contracts. Such contracts stand at the basis of component type checking and component interoperability verifications. The syntactic compatibility of operations and interfaces (in terms of exact matching) may be checked by means of appropriate AOs that we have defined here.

Publishing a syntactic specification of all provided and required interfaces of a component is mandatory in order to make a system that uses it work. However, in order to ensure that it works right, delivering the intended functionality, behavioral information regarding the respective component should be also provided. Such behavioral information may be either interface-specific (capturing aspects regarding the functionality of an interface independently of other interfaces belonging to the same component) or global (involving several component interfaces). In addition to the previously mentioned concepts, we have introduced in this package a BehaviorSpec metaclass for abstracting interface-specific behavioral information; each BehaviorSpec instance is owned by its corresponding Interface object. Concrete behavioral specifications (either at level two - expressing the semantics of services by means of pre/post-condition pairs, or at level three - constraining the order of service calls), should subclass BehaviorSpec and be described in separate packages. This approach ensures a good metamodel manageability and extensibility.

6.2.2.3 Behavioral Contracts. Semantic Specification of Interfaces

Built on SyntacticSpec, the SemanticSpec package adds second-level contract support to our component language. It follows a Design by Contract (DBC) approach in order to provide a semantic specification of interfaces. The metaclasses represented in Figure 6.4 and the relationships among them have been inspired by the specification concepts introduced in [24].

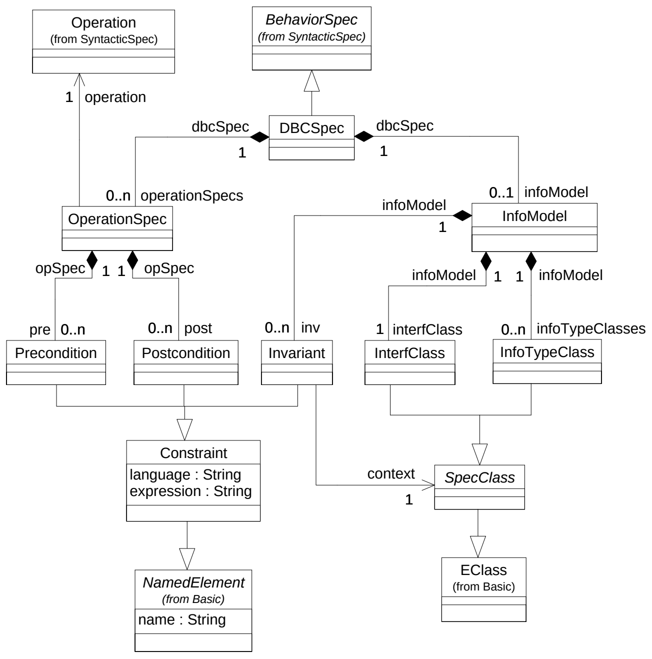

Figure 6.4: Semantic specification of interfaces

A DBCSpec is a kind of behavioral specification that may be attached to an interface. It consists of a collection of operation specifications (one for each service/operation exposed by the interface), together with an interface information model (or state model). As defined in [24], the information model corresponding to an interface is an abstraction of that part of a component's state that affects or may be affected by the execution of operations in the interface. It does not expose implementation details. It is merely an abstraction that helps in defining operations' behavior.

The OperationSpec metaclass allows representing the behavior of its associated Operation, stated in terms of pre/post-condition pairs. A Precondition is a predicate expressed in terms of the input parameters and the state model; a Postcondition is also a predicate, involving both input and output parameters, as well as the state just before the invocation and immediately after.

We have emphasized the challenges we had when transposing into our metamodel the concepts used in [24] for the semantic specification of interfaces. The main one is related to the need of introducing class modeling concepts into ContractCML, in order to properly represent information models.

6.2.2.4 A Model Weaving Approach to Representing Information Models

We have taken a model weaving approach in order to provide a solution to the previously mentioned problem. Model weaving can be regarded as a special kind of model transformation, the latter being generically defined as the process of creating an output model based on one or more input models [47]. By model weaving, different (but related) models can be composed into a consistent whole. Two types of weaving can be distinguished: symmetric and asymmetric. An asymmetric weaving works with a base model and one or several aspect models which it integrates into the base in a user controllable way, as opposed to a symmetric one, where there is no designated base model [47].

We have used XWeave [47], an asymmetric model weaving tool based on the EMF Ecore meta-metamodel. When performing the weaving, we have taken ContractCML as the base and Ecore itself as the aspect. The weaving has been based on name matching[3] among the analogous concepts highlighted in Figure 6.5. We have provided the oAW workflow used to perform the weaving.

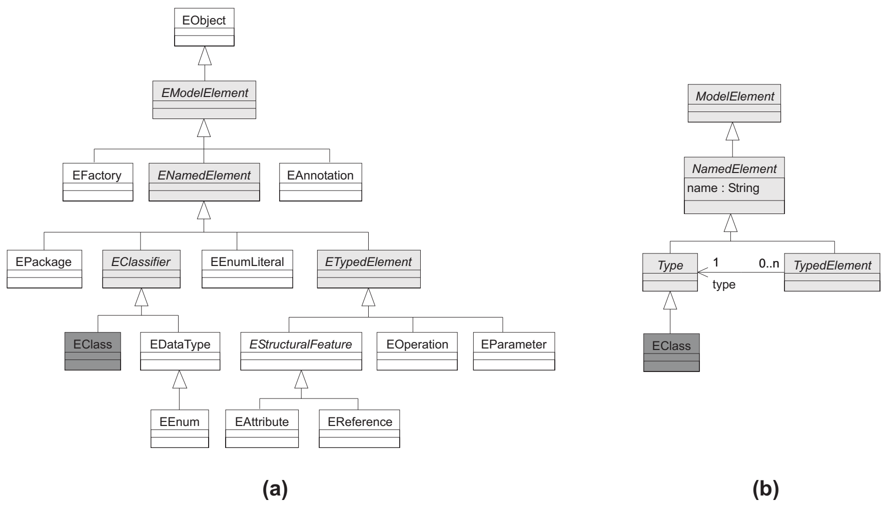

Figure 6.5: `Ecore` (a) and `ContractCML::Basic` (b) concepts correspondence

6.2.3 ContractCML Modeling Example

So as to provide a modeling example using ContractCML, we have considered a simplified variant of the hotel reservation system case study used in [24] and [30]. This has offered the possibility to show some concrete syntax elements of our language, as well as to emphasize the advantages of having a component modeling language that includes semantic specification facilities. Thus, following the presentation of the model, we have conducted a reasoning with respect to establishing the plug-in compatibility of two interfaces (a required and a provided one), based on their semantic specification. Both information model and operations' specification have been taken into account.

6.2.4 Component Models' Simulation Using ContractCML

Within this section, we have proposed a simulation approach for ContractCML component services, that may be employed in component interoperability tests. This method relies on our proposal for representing interface information models. The simulation takes place in

[3]The Ecore concepts in question have been previously remaned, so as to enable the weaving

the context of the XMF-Mosaic framework, being based on an XCore representation of the ContractCML metamodel and the use of XOCL.

6.2.4.1 Proposed Simulation Method

In Section 6.2.2.3, we have emphasized the role of an interface information model in the semantic specification of services pertaining to the interface in question. Within this section, we have illustrated the value of such an information model in simulating the execution of these services, using an executable OCL dialect (XOCL) and its associated execution framework (XMF Mosaic). The proposed simulation method has required enriching the ContractCML metamodel, enabling it to represent not only usage contracts, but also realization[4] contracts [24]. A realization contract attached to a component type contains information regarding the way in which its provided services should be designed in terms of the required ones. The rules related to the definition of such contracts have been formalized as XOCL constraints.

The simulation logic has been implemented in XOCL, within the `ContractCML::Simulator` metaclass. Its `simulate()` method allows simulating the execution of a component service, both the service and its call arguments being sent as parameters to the method. The component in question is assumed to be a part of an architecture in which its required sevices are provided by other components. The core aspect of the proposal consists in the strategy used to configure the object which ensures the simulation infrastructure.

6.2.4.2 Validation

The proposed simulation approach has been validated using an extended version of the Reservation System case study introduced in Subsection 6.2.3, illustrating its usefulness in components interoperability tests.

6.3 Summary

Our contribution in the area of software components' specification is twofold.

First, we have contributed to a reverse engineering approach aimed at bridging the gap between component implementations and specifications. This approach has been achieved as a result of an international collaboration, in an ECO-NET project. Specifically, we have been involved in:

- defining a common component metamodel (CCMM) to be used as the target of the envisioned reverse engineering process. The metamodel has been tested and validated on a non-trivial component benchmark;
- generating an associated repository and model editor, both made available as Eclipse plugins. The repository plugin integrates the required functionality for checking the compilability of CCMM models.

The entire CCMM metamodel specification, including all WFRs and AOs, is available in [11]. The toolchain developed to support the proposed reverse engineering approach has been reported in [12].

Second, we have proposed an integrated approach for handling components' contracts and components' composition. Our proposal improves the state of the art in the field of checking components' interoperability by means of:

[4]The previously discussed contracts are generically referred as *usage contracts*

- ContractCML, a hierarchical component DSML (domain-specific modeling language) focused on representing components' usage contracts. At the moment, ContractCML supports the first two contract levels, syntactic and semantic, but its extensible architecture allows the remaining levels (synchronization and quality of service) to be added in a natural manner. The main advantage of ContractCML over existing industrial and academic component models consists in its ability to represent semantic contracts;
- a simulation method for ContractCML component services within the XMF execution framework. The simulation approach relies on the method proposed for representing semantic contracts and enables reasoning on components' semantic interoperability.

These proposals have been disseminated through the papers [77] and [78, 79] respectively, while relying on previous work with respect to the specification of software components, reported in [83].

Further work is needed to complete the definition of the envisioned component modeling framework. As a first step, we aim at integrating the remaining two contract levels into the language metamodel. Then, we plan to provide a visual concrete syntax to the language and develop the required tool support for using it. This should allow creating component assemblies and reasoning on component interoperability issues. Investigating the possibility of using a theorem prover (such as AtelierB) for establishing the semantic interoperability of components (a far more reliable alternative to simulation) is also considered.

Chapter 7

Conclusions

Reaching a high level of reuse of its artifacts and processes is a maturation proof of software development as an engineering discipline. However, no reuse technique can deliver its promises in the absence of an adequate formal framework. This thesis gathers a number of contributions with respect to the use of formal approaches for ensuring an appropriate formal foundation to software reuse.

The first contribution reported in this thesis fits in the domain of *design pattern's formalization.* Our proposal consists in a full formalization of the GoF State design pattern using the B formal method. This covers both the formal definition of the pattern itself and the formalization of its associated reuse process. The correctness of the whole approach has been established by means of the AtelierB prover.

The second contribution brought by this thesis concerns *an appropriate definition of constraint patterns for object-oriented class models.* Namely, it consists of a new approach (that we refer as *MDE-driven*) to stating OCL-based solutions of such patterns (named *OCL specification patterns*). The rationale is provided by the primary role of assertions within an MDE development process (that of enabling efficient model correctness verifications), which requires them to ensure adequate error diagnosing support. The whole approach has been validated using the OCLE tool, by illustrating its value with respect to both model compilability checks and model testing.

Our third contribution concerns the *formalization of the static semantics of (meta)modeling languages.* This contribution is twofold. On the one side, we have proposed a set of principles regarding an appropriate specification of a static semantics. On the other, we have provided enhancements to the static semantics of UML, MOF, Ecore and XCore, in accordance to these principles. All proposed well-formedness rules and additional operations have been tested and validated using OCLE, EMF, and XMF-Mosaic respectively.

In the last part of this thesis, we have reported on two contributions in the area of *software components' specification.* The first contribution is part of a reverse engineering approach meant at extracting structural and behavioral descriptions from Java component code, with the aim of ensuring the required traceability among component specifications and implementations. The second contribution aims at setting the bases of a framework allowing a full contractual specification of software components and enabling component interoperability verifications. At the basis of such a framework we have proposed ContractCML, a hierarchical DSML focused on representing components' semantic contracts. Based on it, we have further proposed a simulation method for ContractCML component services within the XMF execution framework.

All proposals made have been appropriately motivated and compared to related work in the literature, so as to emphasize their relevance to the field.

Bibliography

[1] ECO-NET Project "Behavior Abstraction from Code: Filling the Gap between Component Specification and Implementation". http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:start.

[2] ECO-NET SVN Repository. svn://aiya.ms.mff.cuni.cz/econet.

[3] eXecutable Metamodeling Facility (XMF) Home Page, Ceteva Ltd. 2007. http://itcentre.tvu.ac.uk/~clark/xmf.html.

[4] Frame Based on the Extensive Use of Metamodeling for the Specification, Implementation and Validation of Languages and Applications (EMF_SIVLA). http://www.cs.ubbcluj.ro/~chiorean/CUEM_SIVLA.

[5] openArchitectureWare (oAW). http://www.openarchitectureware.org/.

[6] Abrial, J.-R. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[7] Ackermann, J. Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. In *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005*, T. Baar, Ed., Technical Report LGL-REPORT-2005-001. EPFL, 2005, pp. 15–29.

[8] Ackermann, J. Frequently Occurring Patterns in Behavioral Specification of Software Components. In *COEA* (2005), pp. 41–56.

[9] André, P., Ardourel, G., and Attiogbé, C. Defining Component Protocols with Service Composition: Illustration withe Kmelia Model. In *6th International Symposium on Software Composition, SC'07* (2007), vol. 4829 of *LNCS*, Springer.

[10] André, P., Chiorean, D., Plasil, F., and Royer, J.-C. Behavior Abstraction from Code: Filling the Gap between Component Specification and Implementation, 2006. ECO-NET 16293RG/2007 Project Proposal.

[11] André, P., and **Petraşcu, Vladiela**. ECONET Project - CCMM Specification v.1.1, 2008. http://www.cs.ubbcluj.ro/~vladi/ThesisReferences/ECONET/ECONET_CCMM_v1_1.pdf.

[12] Anquetil, N., Royer, J.-C., André, P., Ardourel, G., Hnetynka, P., Poch, T., Petraşcu, D., and **Petraşcu, Vladiela**. JavaCompExt: Extracting Architectural Elements from Java Source Code. In *Proceedings of 16th Working Conference on Reverse Engineering - WCRE'09* (2009), IEEE Computer Society, pp. 317–318. Tool Demo [DBLP, IEEE Xplore, IEEE CSDL].

[13] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J., and Zettel, J. *Component-Based Product Line Engineering with UML*. Addison-Wesley, 2001.

[14] Barros, T., Henrio, L., and Madelaine, E. Model-Checking Distributed Components: The Vercors Platform. In *Proceedings of Formal Aspects of Component Software (FACS'06)* (2006), ENTCS.

[15] Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. Making Components Contract Aware. *Computer 32*, 7 (1999), 38–45.

[16] Bezivin, J. On the Unification Power of Models. *Software and System Modeling (SoSyM) 4*, 2 (2005), 171–188. http://www.sciences.univ-nantes.fr/lina/atl/www/papers/OnTheUnificationPowerOfModels.pdf.

[17] Bezivin, J. Introduction to Model Engineering, 2006. http://www.modelware-ist.org/index.php?option=com_remository&Itemid=74&func=fileinfo&id=72.

[18] Blazy, S., Gervais, F., and Laleau, R. Reuse of Specification Patterns with the B Method. In *ZB 2003: Formal specification and development in Z and B* (2003), D. Bert, J. Bowen, S. King, and M. Waldén, Eds., vol. 2651 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 40–57.

[19] Box, D. *Essential COM*. Addison-Wesley, 1998.

[20] Bruel, J.-M., Henderson-Sellers, B., Barbier, F., Le Parc, A., and France, R. Improving the UML Metamodel to Rigorously Specify Aggregation and Composition, 2001. Technical Report. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.15.8132&rep=rep1&type=pdf.

[21] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. The Fractal Component Model and Its Support in Java. *Software Practice and Experience 36*, 11-12 (2006).

[22] Bures, T., Hnetynka, P., and Plasil, F. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications* (2006), IEEE CS, pp. 40–48.

[23] Cechich, A., and Moore, R. A formal specification of GoF design patters. Technical Report 151, UNU/IIST, P.O. Box 3058, Macau, 1999.

[24] Cheesman, J., and Daniels, J. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[25] Chiorean, D., Corutiu, D., Bortes, M., and Chiorean, I. Good Practices for Creating Correct, Clear and Efficient OCL Specifications. In *Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language (NWUML'2004)* (2004), K. Koskimies, L. Kuzniarz, J. Lilius, and I. Porres, Eds., no. 35 in TUCS General Publications, Turku Center for Computer Science (TUCS), Finland, pp. 127–142.

[26] Chiorean, D., and **Petraşcu, Vladiela**. Towards a Conceptual Framework Supporting Model Compilability. In *Proceedings of the Workshop on OCL and Textual Modelling (OCL 2010) at ACM/IEEE 13th International Conference on Model Driven*

Engineering Languages and Systems - MoDELS'10 (2010), vol. 36 of *Electronic Communications of the EASST*, European Association of Software Science and Technology (EASST). 14 pages, http://modeling-languages.com/events/OCLWorkshop2010/submissions/ocl10_submission_10.pdf [DBLP].

[27] Chiorean, D., **Petraşcu, Vladiela**, and Ober, I. Testing-Oriented Improvements of OCL Specification Patterns. In *Proceedings of the 2010 IEEE International Conference on Automation, Quality and Testing, Robotics - AQTR* (2010), vol. II, IEEE Computer Society, pp. 143–148. [ISI Proc., IEEE Xplore, IEEE CSDL].

[28] Chiorean, D., **Petraşcu, Vladiela**, and Ober, I. MDE-Driven OCL Specification Patterns. *Control Engineering and Applied Informatics (CEAI) n/a* (n/a), n/a. [ISI Journal].

[29] Chiorean, D., **Petraşcu, Vladiela**, and Petraşcu, D. How My Favorite Tool Supporting OCL Must Look Like. In *Proceedings of the 8th International Workshop on OCL Concepts and Tools (OCL 2008) at MoDELS* (2008), vol. 15 of *Electronic Communications of the EASST*, European Association of Software Science and Technology (EASST). 17 pages, http://journal.ub.tu-berlin.de/index.php/eceasst/article/viewFile/180/177 [DBLP].

[30] Chouali, S., Heiser, M., and Souquières, J. Proving Component Interoperability with B Refinement. *Electronic Notes in Theoretical Computer Science 160* (2006), 157–172.

[31] Clark, T., Sammut, P., and Willans, J. *Applied Metamodeling: A Foundation for Language Driven Development (Second Edition)*. Ceteva, 2008. http://itcentre.tvu.ac.uk/~clark/docs/Applied%20Metamodelling%20%28Second%20Edition%29.pdf.

[32] ClearSy System Engineering. Atelier B. http://www.atelierb.eu/index-en.php.

[33] Crnkovic, I., and Larsson, M., Eds. *Building Reliable Component-Based Software Systems*. Artech House, Inc., 2002.

[34] Damus, C. W. Implementing Model Integrity in EMF with MDT OCL. Eclipse Corner Articles, Eclipse Foundation. http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html.

[35] DeMichiel, L., Yalcinalp, L., and Krishnan, S. Enterprise JavaBeans Specification Version 2.0, 2001.

[36] Eclipse Foundation. Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf.

[37] Eclipse Foundation. Graphical Modeling Project (GMP). http://www.eclipse.org/modeling/gmp/.

[38] Eclipse Foundation. Model Development Tools (MDT) OCL. http://www.eclipse.org/modeling/mdt/?project=ocl.

[39] Eden, A. H. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, Department of Computer Science, Tel Aviv University, 2000.

[40] Eden, A. H., Hirshfeld, Y., and Yehudai, A. LePUS - A Declarative Pattern Specification Language. Tech. rep. 326/98, Department of Computer Science, Tel Aviv University, 1998.

[41] Flores, A., Moore, R., and Reynoso, L. A Formal Model of Object-Oriented Design and GoF Design Patterns. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity* (2001), FME '01, Springer-Verlag, pp. 223–241.

[42] Fuentes, J. M., Quintana, V., Llorens, J., Génova, G., and Prieto-Díaz, R. Errors in the UML metamodel? *ACM SIGSOFT Software Engineering Notes 28*, 6 (2003), 3–3.

[43] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[44] Garcia, M. Rules for Type-checking of Parametric Polymorphism in EMF Generics. In *Software Engineering (Workshops)* (2007), W.-G. Bleek, H. Schwentner, and H. Züllighoven, Eds., vol. 106 of *Lecture Notes in Informatics (LNI)*, GI, pp. 261–270.

[45] Gervais, F. Réutilisation de composants de spécification en B. Master's thesis, DEA IIE(CNAM) - University of Évry-INT, Évry, France, 2002. http://cedric.cnam.fr/PUBLIS/RC394.ps.gz.

[46] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification (Third Edition).* Addison-Wesley Longman, 2005.

[47] Groher, I., and Voelter, M. XWeave: Models and Aspects in Concert. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling* (2007), ACM Press, pp. 35–40.

[48] Krueger, C. W. Software Reuse. *ACM Computer Surveys 24*, 2 (1992), 131–183.

[49] Laboratorul de Cercetare în Informatică (LCI). Object Constraint Language Environment (OCLE). http://lci.cs.ubbcluj.ro/ocle/.

[50] Lau, K.-K., and Ornaghi, M. OOD frameworks in component-based software development in computational logic. In *Proceedings of LOPSTR98* (1999), vol. 1559 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 101–123.

[51] Marcano, R., Meyer, E., Levy, N., and Souquières, J. Utilisation de patterns dans la construction de spécifications en UML et B. In *Proceedings of AFADL 2000: Approches formelles dans l'assistance au développement de logiciels* (2000). Tech. rep. A00-R-009, LSR Laboratory, Grenoble, France, 15 pages.

[52] Marcano-Kamenoff, R., Lévy, N., and Losavio, F. Spécification et spécialisation de patterns en UML et B. In *Proceedings of LMO'2000: Langages et modèles à objets* (2000), C. Dony and H. A. Sahraoui, Eds., Hermès Science Publications, Mont Saint-Hilaire, Québec, Canada, pp. 245–260.

[53] Merks, E., and Paternostro, M. Modeling Generics with Ecore. In *EclipseCon 2007* (2007). http://www.eclipsecon.org/2007/index.php?page=sub/&id=3845.

[54] Meyer, B. Applying "Design by Contract". *Computer 25*, 10 (1992), 40–51.

[55] Newcomer, E. *Understanding Web Services: XML, WSDL, SOAP, and UDDI.* Addison-Wesley, 2002.

[56] Object Management Group (OMG). CORBA Component Model, V3.0, 2002. http://www.omg.org/technology/documents/formal/components.htm.

[57] Object Management Group (OMG). Model Driven Architecture (MDA) Guide, Version 1.0.1, 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf.

[58] Object Management Group (OMG). Unified Modeling Language (UML) Specification, Version 1.5, 2003. http://www.omg.org/spec/UML/1.5/PDF/.

[59] Object Management Group (OMG). Unified Modeling Language (UML) Specification, Version 1.4.2, 2005. http://www.omg.org/spec/UML/ISO/19501/PDF/.

[60] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification, Version 2.0, 2006. http://www.omg.org/spec/MOF/2.0/PDF.

[61] Object Management Group (OMG). Object Constraint Language (OCL), Version 2.2, 2010. http://www.omg.org/spec/OCL/2.2/PDF/.

[62] Object Management Group (OMG). Unified Modeling Language (UML), Infrastructure, Version 2.3, 2010. http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/.

[63] Object Management Group (OMG). Unified Modeling Language (UML), Superstructure, Version 2.3, 2010. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/.

[64] Parizek, P., and Plasil, F. Modeling Environment for Component Model Checking from Hierarchical Architecture. In *Proceedings of Formal Aspects of Component Software (FACS'06)* (2006), ENTCS.

[65] Pavel, S., Noyé, J., Poizat, P., and Royer, J.-C. A Java implementation of a component model with explicit symbolic protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)* (2005), Springer-Verlag, Ed., vol. 3628 of *Lecture Notes in Computer Science*, pp. 115–125.

[66] Plasil, F., and Visnovsky, S. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering 28*, 11 (2002), 1056–1076.

[67] Pospisil, R., and Plasil, F. Describing the Functionality of EJB using the Behavior Protocols. In *In Week of Doctoral Students (WDS 99)* (1999).

[68] Rausch, A., Reussner, R., Mirandola, R., and Plasil, F., Eds. *The Common Component Modeling Example: Comparing Software Component Models*, vol. 5153 of *LNCS*. Springer, 2008.

[69] Richters, M., and Gogolla, M. Validating UML models and OCL constraints. In *UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference Proceedings* (2000), A. Evans, S. Kent, and B. Selic, Eds., vol. 1939 of *Lecture Notes in Computer Science*, Springer, pp. 265–277.

[70] Schmidt, D. C. Model-Driven Engineering. *Computer 39*, 2 (2006), 25–31.

[71] Snook, C., and Butler, M. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM) 15*, 1 (2006), 92–122.

[72] Sun Microsystems. JavaBeans Specification, 1997. http://java.sun.com/products/javabeans/docs/spec.html.

[73] Szyperski, C., Gruntz, D., and Murer, S. *Component Software: Beyond Object-Oriented Programming (Second Edition)*. Addison-Wesley, 2002.

[74] **Ciobotariu-Boer, Vladiela**, and Petraşcu, D. X-Machines Modeling. A Case Study. In *Proceedings of the Symposium "Colocviul Academic Clujean de Informatică"* (2005), M. Frenţiu, Ed., Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania, pp. 75–80.

[75] **Petraşcu, Vladiela**, and Chiorean, D. Towards Improving the Static Semantics of XCore. *Studia Informatica LV*, 3 (2010), 61–70. http://www.cs.ubbcluj.ro/~studia-i/2010-3/06-PetrascuChiorean.pdf [MathSciNet, Zentralblatt].

[76] **Petraşcu, Vladiela**, and Chiorean, D. XCore Static Semantics - from Requirements to Implementation. In *Proceedings of the National Symposium Zilele Academice Clujene (ZAC)* (2010), M. Frenţiu, Ed., Presa Universitară Clujeană, pp. 73–78.

[77] **Petraşcu, Vladiela**, Chiorean, D., and Petraşcu, D. ContractCML - a Contract Aware Component Modeling Language. In *Proceedings of 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (2008), IEEE Computer Society, pp. 273–276. [ISI Proc., DBLP, IEEE CSDL, ACM DL].

[78] **Petraşcu, Vladiela**, Chiorean, D., and Petraşcu, D. Component Models' Simulation in ContractCML. In *Proceedings of KEPT 2009: Knowledge Engineering Principles and Techniques - Extended Abstracts* (2009), M. Frenţiu and H. F. Pop, Eds., vol. II of *Studia Informatica, Special Issue*, Babeş-Bolyai University of Cluj-Napoca, pp. 198–201. http://cs.ubbcluj.ro/~studia-i/2009-kept/Studia-2009-Kept-3-KSE.pdf [MathSciNet, Zentralblatt].

[79] **Petraşcu, Vladiela**, Chiorean, D., and Petraşcu, D. Component Models' Simulation in ContractCML. In *KEPT 2009: Knowledge Engineering Principles and Techniques - Selected Papers* (2009), M. Frenţiu and H. F. Pop, Eds., Presa Universitară Clujeană, pp. 231–238. (extended version of [78]) [ISI Proc.].

[80] **Petraşcu, Vladiela**, Chiorean, D., and Petraşcu, D. Proposal of a Set of OCL WFRs for the Ecore Meta-Metamodel. *Studia Informatica LIV*, 2 (2009), 89–108. http://www.cs.ubbcluj.ro/~studia-i/2009-2/09-PetrascuChiorean.pdf [MathSciNet, Zentralblatt].

[81] **Petraşcu, Vladiela**, and Petraşcu, D. Formalizing the State Pattern in B. *Pure Mathematics and Applications (PU.M.A) 17*, 3-4 (2006), 397–411. http://www.bke.hu/puma/17_3/PetrascuPetrascu.pdf [MathSciNet].

[82] **Petraşcu, Vladiela**, and Petraşcu, D. Proving the Soundness of an OO Model using the B Method. In *Proceedings of the Symposium "Zilele Academice Clujene"* (2006), Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania, pp. 107–112.

[83] **Petraşcu, Vladiela**, and Petraşcu, D. Architecting and Specifying a Software Component Using UML. In *Proceedings of KEPT 2007 - Knowledge Engineering: Principles and Technologies* (2007), M. Frenţiu and H. F. Pop, Eds., vol. I of *Studia Informatica, Special Issue*, Babeş-Bolyai University of Cluj-Napoca, pp. 332–340. `http://cs.ubbcluj.ro/~studia-i/2007-kept/415-Petrascu.pdf` [MathSciNet, Zentralblatt].

[84] Wahler, M. *Using Patterns to Develop Consistent Design Constraints.* PhD thesis, ETH Zurich, Switzerland, 2008. `http://e-collection.ethbib.ethz.ch/eserv/eth:30499/eth-30499-02.pdf`.

[85] Wahler, M., Basin, D., Brucker, A. D., and Koehler, J. Efficient Analysis of Pattern-Based Constraint Specifications. *Software and Systems Modeling 9*, 2 (2010), 225–255.

[86] Wahler, M., Koehler, J., and Brucker, A. D. Model-Driven Constraint Engineering. *Electronic Communications of the EASST 5* (2006).

[87] Warmer, J., and Kleppe, A. *Object Constraint Language: Precise Modeling with UML*, first ed. Addison-Wesley, 1999.

[88] Wigley, A., Sutton, M., MacLeod, R., Burbidge, R., and Wheelwright, S. *Microsoft .NET Compact Framework (Core Reference).* Microsoft Press, 2003.

Abbreviations

AMN	**A**bstract **M**achine **N**otation
AO(s)	**A**dditional **O**peration(s)
BCR	**B**usiness **C**onstraint **R**ule
CBSD	**C**omponent **B**ased **S**oftware **D**evelopment
CBSE	**C**omponent **B**ased **S**oftware **E**ngineering
CIM	**C**omputation **I**ndependent **M**odel
CMOF	**C**omplete **MOF**
DBC	**D**esign **b**y **C**ontract
DSML	**D**omain **S**pecific **M**odeling **L**anguage
EMF	**E**clipse **M**odeling **F**ramework
EMOF	**E**ssential **MOF**
GMF	**G**raphical **M**odeling **F**ramework
GoF	**G**ang **o**f **F**our
GSL	**G**eneralized **S**ubstitution **L**anguage
HOL	**H**igher **O**rder **L**ogic
HOML	**H**igher **O**rder **M**onadic **L**ogic
LDD	**L**anguage **D**riven **D**evelopment
MDA	**M**odel **D**riven **A**rchitecture
MDD	**M**odel **D**riven **D**evelopment
MDE	**M**odel **D**riven **E**ngineering
MOF	**M**eta **O**bject **F**acility
oAW	**o**pen**A**rchitecture**W**are
OCL	**O**bject **C**onstraint **L**anguage
OCLE	**OCL** **E**nvironment
OMG	**O**bject **M**anagement **G**roup
OMT	**O**bject **M**odeling **T**echnique
PIM	**P**latform **I**ndependent **M**odel
PO(s)	**P**roof **O**bligation(s)
PSM	**P**latform **S**pecific **M**odel
UML	**U**nified **M**odeling **L**anguage
WFR(s)	**W**ell **F**ormedness **R**ule(s)
XMF	**EX**ecutable **M**etamodeling **F**acility