TUCS

Ali Hanzala Khan

Consistency of UML Based Designs Using Ontology Reasoners

Turku Centre for Computer Science

TUCS Dissertations
No 168, December 2013

Consistency of UML Based Designs Using Ontology Reasoners

Ali Hanzala Khan

To be presented, with the due permission of the Department of Information Technologies at Åbo Akademi University, for public criticism in Auditorium Gamma, at ICT building, Turku, Finland, on December 19, 2013, at 12 noon.

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3–5 A, 20520 Turku, Finland

2013

Supervisor

Professor Iván Porres
Department of Information Technologies
Åbo Akademi University
Joukahasenkatu 3–5 A, 20520 Turku
Finland

Reviewers

Professor Oscar Pastor
Centro de Investigación en Métodos de Producción de Software
Universidad Politécnica Valencia (Technical University of Valencia)
Valencia 46022
Spain

Professor Fernando Silva Parreiras
Laboratory of Advanced Information Systems - LAIS
Faculty of Business Sciences
FUMEC University
Av. Afonso Pena 3880
30130-009 Belo Horizonte - MG
Brazil

Opponent

Professor Oscar Pastor
Centro de Investigación en Métodos de Producción de Software
Universidad Politécnica Valencia (Technical University of Valencia)
Valencia 46022
Spain

ISBN 978-952-12-2985-5
ISSN 1239-1883

Dedicated to my beloved father and mother Mr. and Mrs. Muhammad Ali Khan
who gave me guidance, support and love endlessly throughout my life

"Seek knowledge from the cradle to the grave."
Prophet Muhammad (Peace be upon him)

Abstract

Software plays an important role in our society and economy. Software development is an intricate process, and it comprises many different tasks: gathering requirements, designing new solutions that fulfill these requirements, as well as implementing these designs using a programming language into a working system. As a consequence, the development of high quality software is a core problem in software engineering. This thesis focuses on the validation of software designs.

The issue of the analysis of designs is of great importance, since errors originating from designs may appear in the final system. It is considered economical to rectify the problems as early in the software development process as possible. Practitioners often create and visualize designs using modeling languages, one of the more popular being the Unified Modeling Language (UML). The analysis of the designs can be done manually, but in case of large systems, the need of mechanisms that automatically analyze these designs arises.

In this thesis, we propose an automatic approach to analyze UML based designs using logic reasoners. This approach firstly proposes the translations of the UML based designs into a language understandable by reasoners in the form of logic facts, and secondly shows how to use the logic reasoners to infer the logical consequences of these logic facts. We have implemented the proposed translations in the form of a tool that can be used with any standard compliant UML modeling tool. Moreover, we authenticate the proposed approach by automatically validating hundreds of UML based designs that consist of thousands of model elements available in an online model repository. The proposed approach is limited in scope, but is fully automatic and does not require any expertise of logic languages from the user. We exemplify the proposed approach with two applications, which include the validation of domain specific languages and the validation of web service interfaces.

Sammanfattning

Programvara har en viktig roll i vårt samhälle och vår ekonomi. Programvaruproduktion är en invecklad process som består av flera olika delsteg: samling av krav, design av nya lösningar vilka uppfyller dessa krav, samt implementering av dessa designer med hjälp av programmeringsspråk till ett fungerande system. Som en följd av detta är utvecklingen av program av hög kvalitet ett centralt problem i programvaruproduktion. Denna avhandling fokuserar på validering av designer för programvara.

Analys av design är av stor betydelse, eftersom fel som härstammar från designer kan framträda i det slutliga systemet. Det anses vara ekonomiskt att åtgärda problem så tidigt som möjligt i utvecklingsprocessen. Vanligtvis skapar och visualiserar utvecklarna sina designer med hjälp av modelleringsspråk, t.ex. med ett av de mer populära språken Unified Modeling Language (UML). Analysen av designer kan göras manuellt, men för större system finns ett behov för automatiserade mekanismer för analysen.

I denna avhandling lägger vi fram en automatisk metod för analys av UML baserade designer med hjälp av logiska resonerare. För det första lägger detta tillvägagångssätt fram översättningar av UML baserade designer till ett språk som förstås av resonerare i form av logiska fakta. För det andra visar detta tillvägagångssätt hur logiska resonerare används för att härleda de logiska konsekvenserna vilka dessa logiska fakta medför. Vi har implementerat de framlagda översättningarna i form av ett verktyg som kan användas tillsammans med alla vanliga UML kompatibla modelleringsverktyg. Vidare har vi verifierat det framlagda tillvägagångssättet genom att automatiskt validera hundratals UML baserade designer, bestående av tusentals modellelement, vilka är tillgängliga i en on-line modelldatabas. Det framlagda tillvägagångssättet är begränsat i omfattning, men är fullt automatiserat och kräver inte expertkunskap om logiska språk av dess användare. Vi belyser tillvägagångssättet med två exempelapplikationer, vilka inkluderar validering av domänspecifika språk samt validering av gränssnitt för nätverkstjänster.

Acknowledgements

It is a pleasure for me to take this opportunity to express my deepest gratitude to those who made this thesis possible.

First of all, I would like to thank my supervisor Professor Iván Porres for having confidence in me and for supporting and guiding me in this work. It has been a privilege for me to have had the opportunity to work closely with Iván in many interesting projects.

I would also wish to thank Professor Oscar Pastor and Professor Fernando Silva Parreiras for reviewing this thesis and for providing me with useful comments. I would like to thank Prof. Oscar for kindly accepting the task of being my opponent at the public defence.

I would like to thank my teachers of all the courses that I have studied for my PhD degree. Especially Dragos Truscan, Barbro Back, Luigia Petre, Elena Troubitsyna and Patrick Sibelius.

I also thank my co-authors Sören Höglund, Ye Liu, Espen Suenson and Irum Rauf. I have had many interesting discussions with Espen and Irum that helped me evolving this work. I would like to thank Sören and Ye for helping me in the development of the translation tool.

I would like to thank Dragos, Espen and Irum for the proofreading of my articles. I would also like to thank Marta, Kristian, Irum, Adnan, Max, Benjamin and Jokhio for reviewing and proposing corrections in this work.

Furthermore, I would also like to extend my thanks to the administrative and technical staff of our department. Especially, Christel, Nina and Tove for keeping this department running and providing full support in all the administrative matters. Also, Joakim and Magnus for providing excellent technical support.

Thanks also goes to the colleagues at the Software Engineering Lab for providing an enjoyable and fun place to work. In particular I would like to thank Dragos, Adnan, Kristian, Max, Marta, Jeanette, Irum, Espen, Fredrik, Tanvir, Nouman, Benjamin, Mehdi and Torbjörn. Moreover, I would like to thank all my Pakistani friends, especially Moazzam, Qaisar, Mohsin, Adnan, Jokhio and Kashif for always being there for me.

I am honored and grateful for the generous funding I received for four years from VAMOLA project sponsored by Academy of Finland. I am also thankful for the generous scholarships I received from Åbo Akademi and Ulla Tuominen Foundation during the last year of my studies.

I would also like to thank all my family members Hyder, Asif, Umair, Adnan and Aoug for their endless support, for missing me and praying for me. I would like to thank my father (Baa) and mother (Ami), Mr. and Mrs. Muhammad Ali Khan for guiding me throughout my life and loving me endlessly. What I am today is only because of your prayers. I would also like to thank my wife, Faiza Urooj Khan, and my daughter, Shahzeen Khan for their understanding, unyielding devotion, support, patience and unwavering love to lighten up my spirit to complete this thesis.

Ali Hanzala Khan
Turku, Finland.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Software is a key element of our contemporary world. The presence of a variety of software is increasing in all domains of life, therefore, the failure of a software system may not only cause the loss of money and time, but also may become life-threatening.

Software development is a complex process. It starts from gathering requirements, to designing and analysis of specifications using models or by other means, and then implementing the designs using a programming language.

The quality of software is of great concern. There are a number of techniques available in order to improve the quality of a software system, being the most popular software testing.

In software testing, the quality of software is assured by first giving inputs to the software system and then by checking whether the outputs comply to the specification. A specification of software is a description of a system to be developed. It plays an important role in the development of software throughout the development process. Software testing is in many cases a manual job and it involves an iterative loop between developers and testers. Once a software module is developed, it is sent for testing, the tester then checks the module by giving a set of inputs and then analyzes the output. If the module is found erroneous, then the module is sent back to the developer to fix the errors. This iterative loop continues until all errors that are found using this technique are corrected.

Software testing as a quality assurance technique depends on the quality of the system specification. Therefore, we need mechanisms to analyze the specifications of a software system as early in the software development process as possible. The sooner we discover an error, the cheaper it is to fix it.

The specifications are usually written in the form of a natural language or using mathematical notations. In order to analyze the specifications during a software development process, practitioners often create models of these specifications using a modeling language.

A software model usually comprises a number of diagrams. Each diagram in a software model represents different viewpoint of a system. These diagrams allow us to decompose the design of a large system into smaller and more manageable views, and allow us to examine the software designs in the early stages of a software development process. This helps in the detection of errors or shortcomings beforehand, which allows us to make corrections in software designs in a pre development phase. We assume that if a model is a true depiction of a specification, then any error in the model is considered a reflection of an error in the specification.

The diagrams in a software model are described using a particular modeling language. The modeling language can be a general modeling language, or can be limited to some specific domain. A well-known general modeling language used by practitioners during software development process is the Unified Modeling Language (UML) [77, 32]. A language that is limited to some specific domain is called a Domain Specific Language (DSL) [37]. The definition of a modeling language is given in terms of a metamodel by using a metamodeling language, such as Meta Object Facility (MOF) [71] or Kernel Meta Meta Model (KM3) [52]. This thesis focuses on the analysis of modeling artifacts, such as models and metamodels specified using UML superstructure specification [77] and MOF. An example of the modeling artifacts that we tackle in this thesis is shown in Figure 1.1.

In formal methods, the specification of a system is written in the form of mathematical notations. These notations represent an abstraction of a system. Once the specification is produced, the specification is used as a guide for the development of a system. One way to write the specification is in the form of preconditions and postconditions. A precondition is a condition that must be true before a function in software is invoked, and a postcondition is a condition that must be true after the invoked function is completed.

UML models can also be represented in the form of mathematical logics [69], such as, description logics [13] or predicate logics [35]. A *consistent* logical theory is the one which does not contain a contradiction [91, 12]. Similarly, a model which does not contain a contradiction is considered as consistent.

A model in many cases comprises of *classes* and their *associations*, whereas in mathematical/description logic the classes are considered as *concepts* and associations as *roles*. A mathematical theory in many cases comprises a number of formulas. In mathematical theory these formulas are also known as $well-formed\ formulas$ [69]. A mathematical theory is

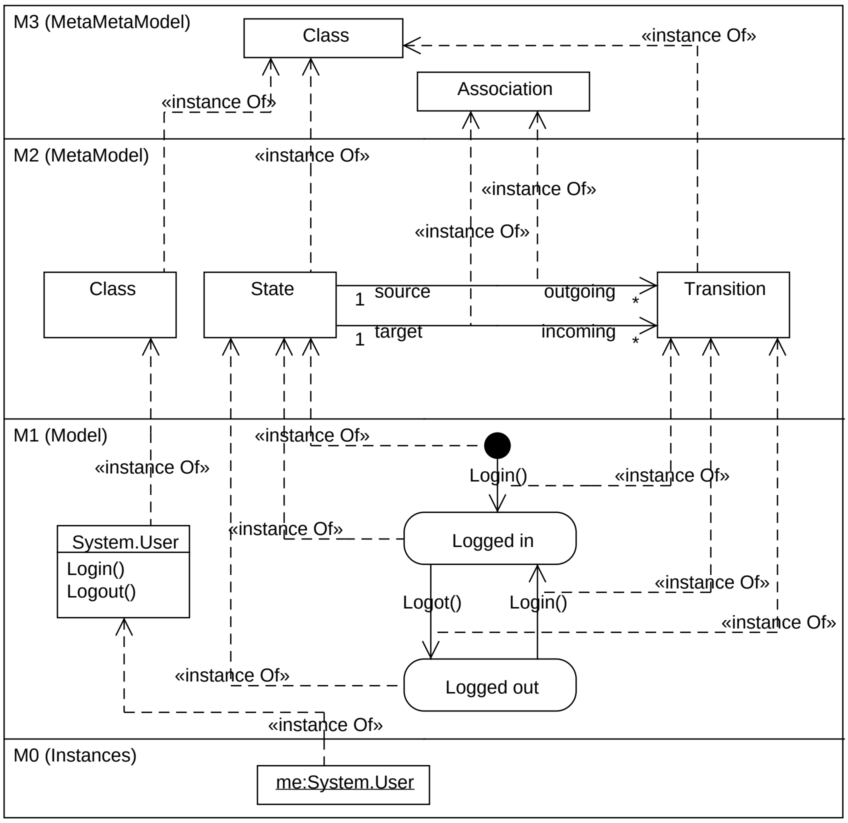

Figure 1.1: Example of modeling artifacts that we treat in this thesis.

considered as consistent if and only if all formulas in the theory are true. In mathematical logic the true formulas of a mathematical theory are called $satisfiable$ formulas [33]. Consequently, the formulas which are not true are called $unsatisfiable$ formulas.

Similar to the formulas of mathematical logic, the modeling artifacts may also have unsatisfiable concepts, and the presence of any unsatisfiable concept in a modeling artifact makes the whole artifact inconsistent. Consequently, such artifacts cannot be instantiated, meaning that if an artifact is inconsistent, then a system based on that artifact cannot exist. More correctly, if a class diagram has unsatisfiable classes, then the objects of these classes cannot exist. Furthermore, in case of an inconsistent behavior diagram (such as inconsistent statechart diagrams), an object cannot enter into an unsatisfiable state.

The unsatisfiable concepts in modeling artifacts should be identified as early in the modeling process as possible. In this thesis we propose an approach that automatically checks the consistency of modeling artifacts. If an artifact is found to be inconsistent, then the proposed approach indicates the unsatisfiable concepts that make the whole artifact inconsistent.

We call the task of finding out the inconsistencies in modeling artifacts the *model validation*. The problem of the validation of modeling artifacts has been discussed in many research papers [104, 85, 96, 15, 41, 3], however, it still remains open. A software model usually comprises a number of structural and behavioral diagrams that represent static and dynamic abstractions of a system, respectively. In each type of a diagram in a model, there are a number of validation problems that have been discussed by other researchers in the past.

Among the validation problems in behavioral diagrams that has already been discussed by other researchers are for instance, analysis of the control looping to find deadlocks [104], analysis of method invocations against the class description for finding deadlocks [85]. Also, checking the consistency of statechart diagrams and class diagrams by using the π-calculus [96].

The validation problems of structural diagrams are very vast. A number of problems that have been discussed by other researchers include, the consistency of UML class diagrams with hierarchy constraints [15], the reasoning of UML class diagrams [19], the Full satisfiability of UML class diagrams [11], and the inconsistency management in model driven engineering [94]. Moreover, there is a number of theorem proving tools available that are based on a high order logic, such as HOL [41]. These tools are very powerful but they require interaction with an expert human user. Also, there are model finders, such as Alloy [10] or Microsoft formula [3], which are automatic, but require that we artificially limit the search space. Furthermore, there are reasoners for logics with the efficient decision procedures that are automatic [90, 83]. In order to use these reasoners we need a mechanism, which translates the designs from the languages used by designers to the input language used by reasoning tools.

In this thesis we propose an approach that first automatically translates the modeling artifacts into logical facts, and then uses automatic logical reasoners to infer the logical consequences in the translated logical facts. Also, the approach we present in this thesis is implemented in the form of a tool that can be used with any standard compliant UML modeling tool. The implemented tool takes a UML model as an input then processes it. This allow us to integrate our approach with different modeling tools.

Although, a lot of research work has already been done in the area of the validation of structural and behavioral diagrams, we still believe there is a room for improvements in the existing work. In this thesis we address the issues that have been inadequately or not addressed in the previous research. The novelty of our work is that we offer the validation of many modeling concepts under one approach. The modeling concepts that can be validated using our approach include the following: classes, objects, associations, links, labeled links, domain and range, multiplicity, composition (herein unshearedness and acyclicity), unique and non-unique associations, ordering, class generalization, and association generalization. Furthermore, the proposed approach also allows us to analyze the conformance of object diagrams against class diagrams, consistency of class diagrams and statchart diagrams, consistency of state invariants written using a subset of object constraint language, and the consistency of multiple models of the same metamodels when merged together.

This thesis also contains several applications of the proposed approach. These include the validation of DSLs and the validation of REST web service interfaces. The DSLs are usually defined in the form of class diagrams. We exemplify the proposed approach by validating more than 300 DSLs comprising of thousands of model elements available in an online repository (the Atlantic metamodel zoo [1]). Another application is the validation of REST web service interfaces. REST [36] is an architectural style to design scalable web services which play well with the existing infrastructure of web. They usually offer simple interfaces that can create, retrieve, update and delete information from a database. In this application, we address the problem of determining the consistency of REST interface design models and explain our reasoning approach. The REST interface design models usually consist of class diagrams and statechart diagrams with state invariants. Here, the class diagrams and the statchart diagrams depict the structural and behavioral abstractions of the REST web services respectively, whereas the state invariant characterizes a state in the statechart diagram. The state invariant is written using a subset of object constraint language and it must be true in case of a specific state is active.

The detection of errors in the above mentioned models, and the results of the performance tests of the proposed approach is the evidence that the proposed approach is viable and practical, and can be applied in the industry.

In the next section we describe the main objectives of the research work that we present in this thesis.

1.2 Research Objectives

The main objective of this thesis is to study and develop an automatic approach addressing the following research questions:

I How can one ensure that a class model is consistent?

II How can one detect inconsistencies between different models, expressed in different languages such as class diagrams and statechart diagrams? Assuming the each model is valid with respect to its language, we still need to ensure that the model elements of one model do not contradict with the model elements of the other model.

III How can one validate the constraints applied to models such as: State invariants in the form of Object Constraint Language (OCL)?

IV How can one validate a given model against its metamodel?

V How can one detect the possible inconsistencies that arise due to the merging of multiple models representing different views of the same system using the same metamodel?

VI How can we integrate our approach in existing UML modeling tools?

1.3 List of Original Publications

This thesis is based on the articles listed below. The details about the contribution of the author is stated in the description of each article. In addition, the research work presented in this thesis extends and in some cases improves the content of some of the listed publications.

I Höglund, Sören and Khan, Ali Hanzala and Liu, Ye and Porres, Ivan. Representing and Validating Metamodels using OWL 2 and SWRL. In Proceedings of the 9th Joint Conference on Knowledge-Based Software Engineering JCKBSE, Aug 2010 [46].
Author's contribution: The author jointly developed the main idea of this paper with other authors. The author contributed to the translations of UML class diagrams to OWL 2.

II Liu, Ye and Höglund, Sören and Khan, Ali Hanzala and Porres, Ivan. A Feasibility Study on the Validation of Domain Specific Languages Using OWL 2 Reasoners. In Third Workshop on Transforming and Weaving OWL Ontologies and MDE at TOOLS, Jun 2010 [63].
Author's contribution: This article is the implementation and the validation of the work presented in Article I. The author jointly developed the main idea of this paper with other authors. The author

contributed to the implementation of UML class diagram to OWL 2 translations in the form of a MOFScript-based translation tool. The translation tool is used to translate UML metamodels of the Atlantic Zoo into OWL 2 in order to conduct the study shown in this article.

III Khan, Ali Hanzala and Suenson, Espen and Porres, Ivan. Representation and Conformance of UML Models Containing Ordered Properties Using OWL 2. In OrdRing First International Workshop on Ordering and Reasoning, at The 10th International Semantic Web Conference (ISWC) Oct 2011 [57].
Author's contribution: The main idea of this article was jointly developed by all authors. The author and Suenson proposed the translations of UML object diagrams to OWL 2 and developed the translation tool using MOFScript.

IV Khan, Ali Hanzala and Suenson, Espen and Porres, Ivan. Class and object diagram Conformance using OWL 2 Reasoners. In 12th Symposium on Programming Languages and Software Tools SPLST, Oct 2011 [58].
Author's contribution: This article is the extension of the work presented in Article III. The main idea of this article was jointly developed by all authors. The author and Suenson proposed the translations of UML object diagrams with ordered properties to OWL 2 and developed the translation tool using MOFScript.

V Khan, Ali Hanzala and Rauf, Irum and Porres, Ivan. Consistency of UML Class and Statechart Diagrams. In First International Conference on Model-Driven Engineering and Software Development, MODELSWARD Feb 2013 [56].
Author's contribution: The main idea of this article was jointly developed by all authors. The author and Rauf proposed the translations of UML statechart diagrams and OCL constraints to OWL 2 and developed the prototype of the translation tool using Python.

VI Rauf, Irum and Khan, Ali Hanzala and Porres, Ivan. Analyzing Consistency of Behavioral REST Web Service Interfaces . The 8th International Workshop on Automated Specification and Verification of Web Systems, WWV, Jun 2012 [86].
Author's contribution: This article is the implementation and evaluation of the work presented in Article V. The main idea of this article was also jointly developed by all authors. The author and Rauf proposed the translations of UML diagrams and OCL constraints depicting an interface of a REST Web service to OWL 2 and developed the prototype of the translation tool using Python.

1.4 Research Contribution Overview

The summary of the research objectives that we address in different articles is given in Table 1.1. In this table, the rows refer to the research objectives that we address in this thesis, and the columns point to the articles on which these research objectives are based.

Table 1.1: Research contribution overview.

Research	**Publications**					
Objectives	I	II	III	IV	V	VI
I	✓	✓				
II					✓	✓
III					✓	✓
IV			✓	✓		
V	✓		✓	✓	✓	
VI	✓	✓	✓	✓	✓	✓

1.5 Thesis Organization Overview

This thesis is organized as follows: Chapter 2 discusses the terminologies used in this thesis, such as: MDE, UML, ontology foundations, description logic and OWL 2. Chapter 3 describes the related work. The related work is categorized according to the problems discussed in the rest of the chapters. Chapter 4 discusses the translations of UML class diagram concepts, DPF and OCL constraints into OWL 2. This chapter is based on the work presented in Articles I, III and V. Chapter 5 presents the application of the UML class diagrams translations proposed in Chapter 1, and discusses how to validate the UML class diagrams available in the Atlantic metamodel zoo (an online repository containing more than 300 metamodels) by using OWL 2 reasoners. This chapter is based on the work presented in Article II. Chapter 6 addresses the issue of the consistency of class diagrams and statechart diagrams with state invariants. This chapter is based on the work presented in Articles I and V. Chapter 7 presents the application of the work discussed in Chapter 6 by using a scenario of a REST web service interface. It is based on the work described in Articles V and VI. Chapter 8 discusses the translation of UML object diagraming concepts into OWL 2, and also describes how to validate these concepts using OWL 2 reasoners. This chapter is based on the work presented in Articles I, III and IV. Chapter 9 proposes the method of the validation of multiple UML diagrams of the same metamodel using OWL 2 reasoners. This chapter is based on the lessons learned from the Articles I, III, IV and V. Finally, Chapter 10 concludes the thesis.

Chapter 2

Background

In this chapter, we give an overview of the MDE foundations, ontology foundations and the validation problems that we address in this thesis. We also discuss the proposed approach that addresses these problems.

2.1 MDE Foundations

Model Driven Engineering (MDE) [55] advocates the use of models to represent the most relevant design decisions of a software development project. Each software development project involves the creation of many models. A *model* is a description or an analogy used for describing, visualizing and observing different viewpoints of a system at different levels of abstractions [77]. Each model in a software development project is described by using a particular modeling language, such as UML or DSL. A modeling language is an abstract syntax for models that specifies the allowed elements and their relationships in the form of a *metamodel* [77]. Similarly, each modeling language is designed by using a meta-language, such as Meta Object Facility (MOF) [71] or Kernel Meta Meta Model [52]. A meta-language is defined in term of *metametamodel* that describes the meta-language architecture for the designing of metamodels or DSLs [77].

2.1.1 Four Layer OMG Modeling Hierarchy

The Object Management Group (OMG) [71] specifies the object oriented modeling concepts in the form of a four layer modeling hierarchy. Each layer represents different types of instances involved in the object oriented modeling such as, objects are the instances of classes, and the classes are the instances of meta-classes, where classes are the part of a model, and meta-classes are the part of a metamodel ([43], p.5). These layers are named as M3, M2, M1 and M0. A graphical overview of these layers is given in Figure 2.1. The details about these layers are as follows:

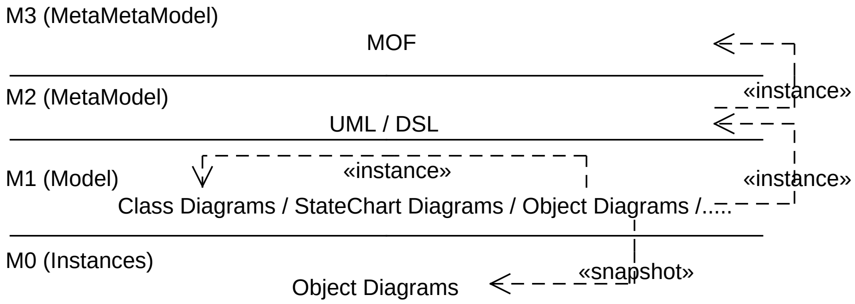

Figure 2.1: The four layer OMG Modeling Hierarchy.

M3 The M3 layer is the top most layer of the OMG modeling hierarchy and known as a metametamodel layer. This layer holds a specification of a language in terms of a metametamodel that is used to describe modeling languages. The OMG uses MOF in this layer. The MOF is a metametamodel or an abstract syntax of modeling languages specified by the OMG. It comprises of model elements such as, Classes, Attributes and Associations [94]. There are two different variants of MOF, i.e., Essential MOF (EMOF) and Complete MOF (CMOF). In this thesis we focus on the analysis of MOF based models.

M2 The metamodel layer M2 is an instance of a metametamodel layer M3, and it holds the specification of DSLs in the form of metamodels. A metamodel also consists of model elements, and the model elements of a metamodel are the instances of the model elements of a metametamodel.

M1 The M1 layer is responsible to represent the knowledge of a certain domain by using the instances of a modeling language specified at the M2 layer. For example, if M2 layer defines an abstract syntax of UML class diagrams then layer M1 defines the class diagrams.

M0 The last layer M0 of the MDE hierarchy is responsible for holding the runtime instances of model elements defined at the M1 layer. For instance, if the M1 layer specifies a class diagram of a certain part of a software, then the M0 layer defines the instances of a model element defined in the class diagram of the layer M1 in the form of object diagrams.

The OMG proposes the use of UML for the designing of modeling artifacts at all levels of modeling hierarchy [72].

2.1.2 UML

The Unified Modeling Language (UML) is a widely used graphical notation for the designing of modeling artifacts at all layers of the OMG modeling hierarchy, i.e., Metametamodel, Metamodel and Model layer [77, 43]. The UML was formally proposed by OMG in 1997 ([60], p.176) and it is based on MOF. The main purpose of UML is to help IT professionals in the designing of computer applications. With time UML evolved considerably and still the scope of UML is expanding rapidly. Nowadays, UML is not only used in the designing of computer applications, but also in the automatic model based software development, which includes automatic code generation from models.

A software project involves the creation of many designs that depict both static and behavior abstractions of a software project. The UML has a capacity to portray both kinds of abstractions. The static structure of a software project is captured by using a class diagram, whereas, a behavior of the software is designed using statechart diagrams or sequence diagrams. A class diagram is a collection of classes and their associations. A class in a class diagram can depict the attributes and operations, but the actual behavior of these operations is captured by using behavioral diagrams.

The UML behavioral diagrams, such as statechart diagrams and sequence diagrams, are used to describe behaviour of an object of a class during its lifetime. These diagrams are composed of states and transitions, where each transition is annotated with an operation. The operation on the transition describes what happens to the object during its whole lifetime. Each state in a statechart diagram is made using state invariant. A state invariant is a condition that characterises the state and must be true in case of a specific state is active. These invariants are written using object constraint language or some other constraint language.

In this thesis, we are focused on the analysis of modeling artifacts based on UML/MOF.

2.1.3 Meta Object Facility

The Meta Object Facility (MOF) is a standard for MDE [55], and it is based on the UML superstructure specifications [77] specified by OMG. The MOF is used for defining UML based metamodels. It consists of model elements such as, Classes, Attributes and Associations [94]. The relevant part of MOF architecture used in this thesis is shown in Figure 2.2. In this thesis we mainly focus on the validation of metamodels and models. The model validation includes the validation of class diagrams, object diagrams and statechart diagrams including constraints written using a subset of OCL. Therefore, the MOF architecture that we present here is restricted to the fragment, which describes the metamodels that are capable to define both static and behavioral structure of a software project.

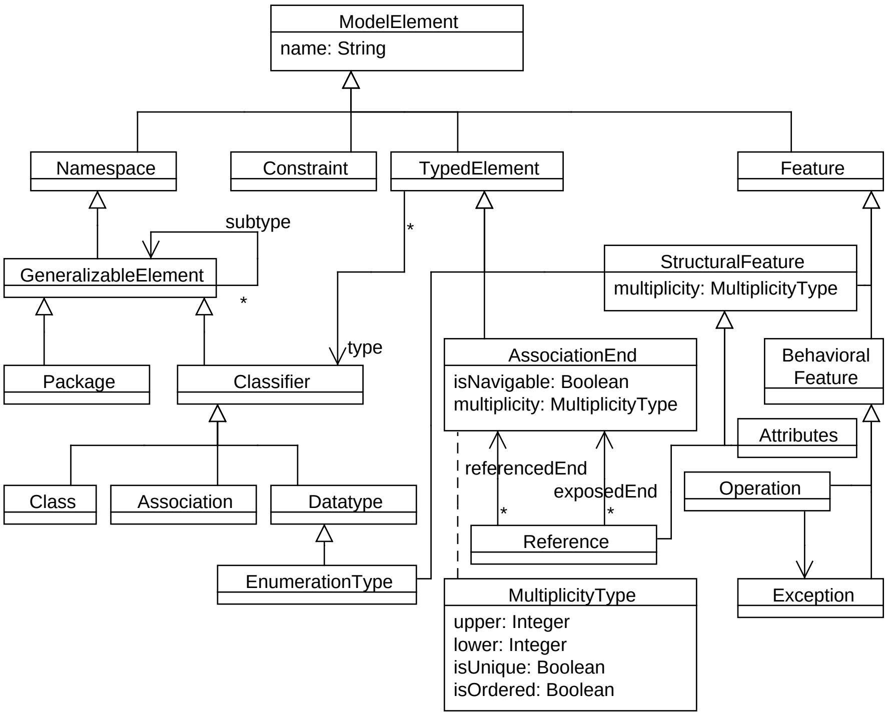

Figure 2.2: A fragment of MOF Metametamodel.

The model of MOF is based on the foundation of object relationship modeling. Here, the model elements *Class*, *Association* and *Datatype* are used to define the objects, links and data values, respectively. All these model elements are derived by $Classifier$, and every $Classifier$ is derived by a *GeneralizableElement* and has a $Namespece$. Each element of the $Namespace$ is derived by a $ModelElement$. Every object can have attributes and operations which are described by the model element *Attributes* and *Operations* respectively, where the model element *Attributes* is derived by $StructuralFeature$ and the model element *Operat−ion* is derived by $BehavioralFeature$. Similarly the attributes of associations, such as multiplicity, ordering, uniqueness and navigability are described by the attributes of the model element *AssociationEnd*. The model element *AssociationEnd* is derived by the model element *TypeElement*. The referenced and exposed end of the association is defined by the model element $Reference$, which is derived from the model element $StructuralFe-$*ature*. The additional constraints on metamodels in the form of OCL or in the other textual format are described by the model element *Constraint*. The model element *Constraint* in the MOF metametamodel is derived by *ModelElement*.

2.1.4 Constraints

The UML models can be annotated with constraints. The constraints are used to add additional restrictions on the models. These constraints may also have errors and can make whole model inconsistent. In this thesis, we also discuss the semantic meaning of a subset of constraint languages and show how to validate the UML diagrams with constraints using logic reasoners. The constraint languages that are supported by the proposed approach are as follows:

Object Constraint Language (OCL): OMG specifies the UML well-formedness rules and a part of UML semantics by using the Object Constraint Language (OCL) [74]. The OCL is a textual constraint language and is always specified by using certain OCL constructs, for example, context, inv, value and size. The OCL constraints are also used to apply additional restrictions over UML diagrams that cannot be applied using existing UML notations. The constraints are also known as invariants.

Diagram Predicate Framework (DPF): In order to apply restrictions on UML diagrams, there are also other textual constraint languages available, one of them is proposed by the Diagram Predicate Framework DPF [61]. DPF proposes to apply constraints by adding a textual symbol over the existing UML notations, such as [*irr*], [*inj*] and [*bij*].

The detail about the semantics of different types of constructs supported in our approach, and how to analyze UML diagrams with these constraints will be discussed later in different chapters of this thesis.

2.1.5 XML Metadata Interchange

The XML Metadata Interchange (XMI) [73] is a Extendable Markup Language (XML) [99] based standard for a model interchange specified by OMG. The models are composed of model elements and each model element has some attributes. In XMI each model element of a model is represented as a XML element with unique ID, and an attribute of a model element is represented as a XML element attribute. In models, the model elements are usually interconnected with each other. In XMI the interconnection between XML elements is achieved by using the ID of one XML element as a reference in another. The modeling tools usually generate the XMI of UML models in some specific format. The XMI format that is understandable by our approach is the XMI 2.1 schema for UML 2.3.

2.2 Ontology Foundations

An ontology is a specification of a conceptualization [62]. In this thesis our understanding of the term *"specification of a conceptualization"* is the specification of concepts and relationships that can represent an abstraction of a program. The abstraction of a program is typically expressed in the form of models by using modeling languages such as UML or DSLs. For example the fact that a class C1 is a subclass of a class C2 is drawn by using UML, as given in Figure 2.3.

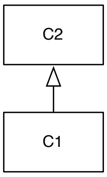

Figure 2.3: A UML model depicting a class C1 being a subclass of a class C2.

In logics the abstraction of a program is expressed in the form of logical facts using logical languages such as description logic [48] or predicate logic [35]. For example the fact depicted in Figure 2.3 that the class C1 is a subclass of the class C2 is written in description logics as:

$$C1 \subseteq C2$$

In ontologies the concepts are represented in the form of axioms that depict the specification of a conceptualization [62]. These axioms represent concepts as classes, and the relationship among concepts as properties. Since the ontology deals with concepts and their relationships, the language used for writing an ontology is semantically very close to the language used to express the logic [62]. This allows us to write the logical facts as axioms in the ontology. For example, the specialization relation $C1 \subseteq C2$ (shown in Figure 2.3) is written in ontology as:

```
SubClassOf( C1 C2 )
```

The ontologies are typically used at the semantic level, due to this, they play an essential role in the interoperability of heterogeneous systems. Also, ontologies are part of the W3C standards [23]. The main role of ontologies in the semantic web is to act as a knowledge base that represents data models at all levels of abstractions. The prominent use of ontologies includes interoperability of systems containing multiple database searches using different web services [62]. However, in this thesis we are using ontologies to represent the semantics of UML models as logical facts, so that we may able to infer the logical consequences from these logical facts automatically using a reasoner.

2.2.1 Reasoners

A reasoner is a utility that automatically infers the logical consequences from a set of logical facts. The reasoning performed by a reasoner is based on the inference rules [14]. These rules are written in a form of a logic, often by using description logic or first-order predicate logic [14]. The reasoning is typically carried out by forward chaining and backward chaining [28]. A reasoner takes a set of logical facts as an input and then checks if each rule is valid or not. A fact is valid if it follows the inference rules implemented in a reasoner. If the fact is true under an interpretation, then so is the conclusion.

2.2.2 Description Logic

The Description Logic (DL) used in our approach is classified as $\mathcal{SROIQ}$ [48]. Description Logic is made up of concepts, denoted here by C, D, and roles, denoted here by R, Q. A concept or role can be named, also called atomic, or it can be composed from other concepts and roles.

An interpretation $\mathcal{I}$ consists of a non-empty set $\Delta^{\mathcal{I}}$ and an interpretation function which assigns a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ to every named concept C and a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to every named role R.

The constructors of Description Logic are as follows:

Everything	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
Nothing	$\bot^{\mathcal{I}} = \emptyset$
Complement	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
Inverse	$(R^{-})^{\mathcal{I}} = \{(y, x) \mid (x, y) \in R^{\mathcal{I}}\}$
Intersection	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Union	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Restriction	
Universal	$(\forall R.C)^{\mathcal{I}} = \{x \mid \forall y.(x, y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
Existential	$(\exists R.C)^{\mathcal{I}} = \{x \mid \exists y.(x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
Cardinality	$(\geq n\, R)^{\mathcal{I}} = \{x \mid \#\{y \mid (x, y) \in R^{\mathcal{I}}\} \geq n\}$
	$(\leq n\, R)^{\mathcal{I}} = \{x \mid \#\{y \mid (x, y) \in R^{\mathcal{I}}\} \leq n\}$

where $\#X$ is the cardinality of X. The axioms in DL can be either inclusions $C \sqsubseteq D$, $C \sqsubseteq D$ or equalities $C \equiv D$, $R \equiv Q$.

An interpretation satisfies an inclusion $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ and an inclusion $R \sqsubseteq Q$ if $R^{\mathcal{I}} \subseteq Q^{\mathcal{I}}$. An interpretation satisfies an equality $C \equiv D$ if $C^{\mathcal{I}} = D^{\mathcal{I}}$ and an equality $R \equiv Q$ if $R^{\mathcal{I}} = Q^{\mathcal{I}}$. $\mathcal{I}$ satisfies a set of axioms if it satisfies each axiom individually – $\mathcal{I}$ is then said to be a model of the set of axioms. Given a set of axioms $\mathcal{K}$, a named concept C is said to be satisfiable if there exists at least one model $\mathcal{I}$ of $\mathcal{K}$ in which $C^{\mathcal{I}} \neq \emptyset$. A set of axioms is said to be satisfiable if all of the named concepts that appear in the set are satisfiable. If a set of axioms $\mathcal{K}$ is satisfiable, we say that an axiom ϕ is satisfiable with respect to $\mathcal{K}$ if $\mathcal{K} \cup \{\phi\}$ is satisfiable. Similarly, we say that ϕ is unsatisfiable (w.r.t. $\mathcal{K}$) if $\mathcal{K} \cup \{\phi\}$ is unsatisfiable.

The decidability of $\mathcal{SROIQ}$ is demonstrated by Horrocks et al. [48], and there exist several reasoners that can process answer satisfiability problems automatically [90, 83].

2.2.3 Web Ontology Language OWL 2

The Web Ontology language OWL 2 [23] is a language for defining ontologies. The OWL 2 provides axioms to express model-theoretic semantics which are compatible with the DL $\mathcal{SROIQ}$ such as classes, properties, individuals and data values [23]. The OWL 2 is also supported by logic reasoners such as Pellet [90] and HermiT [83]. In this thesis, we use the OWL 2 functional syntax (OWL2fs) [20] to explain the translations of UML concepts to OWL 2 axioms. The UML to OWL 2 translation that we propose in this thesis is also implemented in the form of a translation tool. The translation tool generates OWL 2 files that contain the translations of UML models in two different syntaxes such as: OWL2fs and OWL 2 Manchester syntax. The reason of generating outputs in two different OWL 2 syntaxes is because we want to analyze the performance of different reasoners. The detail about the performance of different reasoners will be discussed in Chapter 5. The interpretation of the main OWL 2 expressions used in this thesis is shown in Table 2.1. A complete description of the OWL 2 semantics, including support for data types can be found in [23].

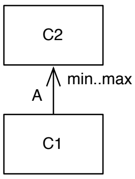

Figure 2.4: The UML Model representing two classes C1 and C2 connected with each other using the association A.

Table 2.1: DL interpretation of the main OWL 2 expressions used in this thesis

OWL 2	**DL**
`SubClassOf(C D)`	$C \sqsubseteq D$
`EquivalentClasses(C D)`	$C \equiv D$
`DisjointClasses(C D)`	$C \sqcap D = \emptyset$
`ObjectIntersectionOf(C D)`	$C \sqcap D$
`ObjectUnionOf(C D)`	$C \sqcup D$
`SubObjectPropertyOf(R1 R2)`	$R1 \sqsubseteq R2$
`InverseObjectProperties(R1 R2)`	$R1 \equiv R2^{-1}$
`InverseFunctionalObjectProperty(R)`	$\top \sqsubseteq (\leq 1\, R^{-})$
`ObjectPropertyDomain(R C)`	$\exists R.\top \sqsubseteq C$
`ObjectPropertyRange(R C)`	$\top \sqsubseteq \forall R.C$
`ObjectMinCardinality(n R)`	$\geq n\, R$
`ObjectMaxCardinality(n R)`	$\leq n\, R$
`ObjectExactCardinality(n R)`	$(\geq n\, R) \sqcap (\leq n\, R)$
`ClassAssertion(C x)`	$C(x)$
`ObjectPropertyAssertion(R x y)`	$R(x, y)$
`NegativeObjectPropertyAssertion(R x y)`	$-R(x, y)$

Example: The OWL 2 translation of the UML model shown in Figure 2.4 is as follows:

```
Declaration(Class(C1))
Declaration(Class(C2))
Declaration(ObjectProperty(A))
ObjectPropertyDomain( A C1 )
ObjectPropertyRange( A C2 )
SubClassOf( C1 ObjectMinCardinality( min A ) )
SubClassOf( C1 ObjectMaxCardinality( max A ) )
```

2.2.4 Ontology Definition Metamodel

The OMG specifies the Ontology Definition Metamodel (ODM) [75] that makes the concepts of OMG modeling hierarchy applicable to the ontology engineering [75]. The ODM follows the similar hierarchy as the one mentioned in a four layer OMG modeling hierarchy (see Figure 2.1). The overview of the ODM hierarchy is shown in Figure 2.5.

The ODM is a specification of ontology structure, and it is derived from MOF [75, 60]. It comprises of classes, associations and constraints ([60], ch.7).

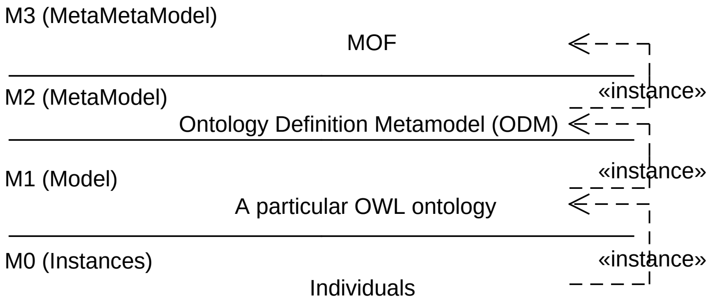

Figure 2.5: A graphical overview of the ODM in a four layer OMG Modeling Hierarchy.

The core of ODM represents formal logic languages, such as Description Logic (DL), Common Logic (CL) and first-order predicate logic. The scope of these languages covers the representations of higher order complex representations of simple taxonomic expressions [75].

The other metamodel derived from MOF that is essential for ODM is the UML [75, 60]. The UML is the widely used modeling language for the designing of conceptual and logical models. The UML is also derived from MOF, and there exist commonalities between UML and ODM specification. Therefore UML notations are also used for ontology modeling([60], ch.7).

The ODM includes a number of metamodels that provide the modeling specification of languages such as: OWL and Resource Definition Framework [50]. In our approach, we use ODM metamodel OWL in order to represent the MOF/UML based models. In our approach we use a decidable fragment of OWL 2 that is based on DL. This fragment is known as OWL 2 DL.

2.2.5 ODM vs UML

The UML and ODM are derived from MOF [75, 60], therefore there exist commonalties, as well as differences between them. The summary of the common features of UML and ODM is shown in Table 2.2. Since in this thesis we use OWL 2 DL as a specification language for ontologies, the comparison between UML and ODM mentioned here is given in terms of UML and OWL 2 DL.

There are some features in UML which do not have equivalent OWL 2 elements (see Table 2.3). In such cases we translate the UML elements by using a combination of existing OWL elements which have an equivalent DL meaning as UML. These translations will be discussed later in this thesis in different chapters.

Table 2.2: UML elements which have the DL equivalent OWL 2 DL elements

UML Elements	OWL Elements
class	class
instance	individual
binary association	property
binary association link	property assertion
class generalization	subclass
property generalization	subproperty
enumeration	oneOf
multiplicity	min/max/exact Cardinality
navigation	domain/range
datatype	datatype

2.2.6 Open and Closed world assumptions

Although there are commonalities between UML and OWL, still there exists a foundational difference between UML and OWL, i.e. the open world and closed world assumptions. The UML follows closed world assumptions where complete knowledge is assumed to be provided, whereas in OWL 2 the knowledge, which is not provided, remains unknown. This knowledge includes the well-formedness rules which are defined in the UML superstructure specifications [77].

While reasoning about the UML models using an OWL reasoner we realize that the representation of UML elements using only the obvious OWL 2 axioms is not enough, because it lacks UML well-formedness. For example in UML an object cannot belong to two classes, except when these two classes are in a specialization relationship, whereas in OWL 2 an individual (class assertion or object) can belong to many classes except to those classes, which are marked as disjoint with each other. In this thesis, we make sure that the translations that we present in this thesis follow the UML well-formedness rules and also bridge the gap of open-close assumptions between UML and ontologies.

2.3 Analyzing UML Models

Analysis of UML models and metamodels is very important at all layers of OMG modeling hierarchy [104, 85, 96, 15, 41, 3]. The analysis approach of UML models presented in this thesis addresses a number of validation

Table 2.3: UML elements which do not have the DL equivalent OWL 2 DL elements

UML Elements	OWL Elements
ordering	X
composition	X
composition unshearedness	X
composition acyclicity	X
non-unique properties	X
label on a link	X
state	X
transition	X
state invariant	X
OCL constructs	X
DPF textual symbols/constructs	X

problems throughout the MOF modeling hierarchy. The validation problems that we address in this thesis are as follows:

Validation of Metamodels against MOF: Two problems are addressed: (1) Are all model elements of metamodels at layer M2 instances of model elements of a metametamodel that exists at layer M3?, (2) Do all model elements of a metamodel at layer M2 follow the well-formedness rules specified at the layer M3?

Validation of Models against Metamodels: Two problems are addressed: (1) Are all model elements of the models at layer M1 instances of model elements of a metamodel that exists at layer M2? (2) Do all model elements at layer M1 follow the constraint expressing the well-formedness and other constraints in the form of OCL or textual symbols specified at layer M2?

Validation of Models against Models with Constraints: Two problems are addressed: (1) Do the valid models of different metamodels at layer M1 contradict with one another when viewed together? We address this problem by checking consistency of a class diagram and a statechart diagram with state invariants altogether. (2) Do the valid models of the same metamodel at layer M1 contradict with one another when viewed together. We address this problem by validating multiple valid models of the same metamodel together, so that the possible contradictions that arise when we view multiple models together can be detected.

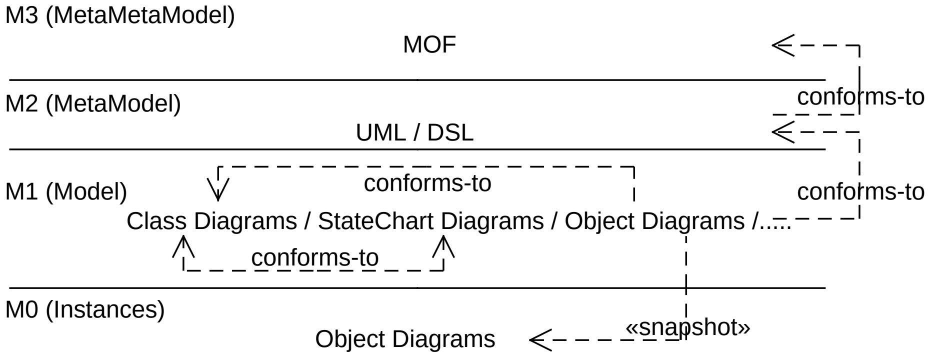

Figure 2.6: An overview of a Metamodel/Model conformance hierarchy.

Validation of Object Diagrams against Class Diagrams: Two problems are addressed: (1) Are model elements of object diagrams at layer M0 instances of the model elements of class diagrams that exist at layer M1, (2) Do model elements on layer M0 follow the well-formedness rules and other constraints expressed by using OCL or textual symbols specified at layer M1.

In order to address all above mentioned validation problems, we propose to automatically translate UML metamodels and/or models into OWL 2 [48], and analyze these translations using automated OWL 2 reasoners [90, 83].

2.3.1 OWL 2 Reasoners

We have several selection criteria for the reasoners, which include, first, a complete support for OWL 2 and Semantic Web Rule Language (SWRL, discussed later Chapter 4), and second, the reasoner is freely available as open source. The first requirement is motivated by the ontologies used in our research. The second requirement ensures that the research is easily repeatable by others.

Based on these criteria, we have chosen the following two reasoners:

1. Pellet [90]: An open source Java-based ontology reasoner developed by Clark& Parsia LLC, which is an R&D firm, specializing in Semantic Web and advanced systems.

2. HermiT [83]: An open source reasoner that is implemented in Java, and developed by the Information Systems Group of Oxford University.

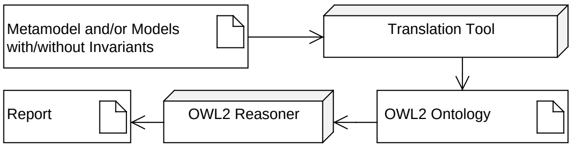

Figure 2.7: Workflow of the proposed consistency checking approach

Both reasoners are implemented in Java, offer complete support for OWL 2 and SWRL and are freely available as open source, which satisfy our requirements.

2.3.2 Reasoning Tool Chain for UML Models

The workflow of our approach is shown in Figure 2.7. A number of UML metamodels/class diagrams and object diagrams or statechart diagrams are taken as an input. All the inputs are translated to a decidable fragment of OWL 2, i.e., OWL 2 DL [20]. We have chosen OWL 2 DL to represent our UML models, because there exist several OWL 2 reasoners [90, 83] for checking concept satisfiability. In the next step, the OWL 2 translations of UML diagrams are passed to a reasoner in the form of an ontology. The reasoner processes the ontology and produces a validation report. The validation report reveals the inconsistencies in the ontology representing the UML models. The detailed discussion about the contents of the validation report are discussed later in different chapters of this thesis.

Chapter 3

Related Work

In this chapter, we discuss the most important related works done by other researchers in the area of model validation. The discussion included in this chapter is categorized according to the problems that we address in the rest of the thesis.

3.1 Consistency of UML Class Diagrams

Description Logics: The pioneering work in the area of DL formalization of UML class diagrams was presented by Berardi et al. [19]. Their approach is purely theoretical, and gives very detailed formalization of UML class diagrams using DL. We consider their work as a starting point of our research, specially in the area of metamodel validation. In our work of metamodel validation, we expand the scope of their work, by addressing the formalization of some of the important class diagram concepts, which were not addressed in their work, such as composition, property specialization and the most specific class assumption. Also, we propose and implement the DL equivalent OWL 2 translations of UML class diagram concepts in the form of a translation tool. We have tested the translation tool on published metamodels. The results indicate that most of the errors discovered involve composition. This clearly shows the importance of the validation of composition and the feasibility of our approach in practice.

Artale et al. [11] proposed an approach that is similar to our approach. However, the problem that they solved, i.e., full satisfiability, is a special case of satisfiability. It is not clear why they limit themselves in this way, since the general case of satisfiability is much more useful for validation.

Several authors have proposed formalizations of UML, MOF and Ecore metamodeling languages, including Akehurst et al. [31], Alanen and Porres [9], Clark et al. [29], Varró [95] and Van Der Straeten [94]. Balaban et al. [15] discussed the consistency of UML class diagrams with hierarchy constraints. These formalizations can be seen as alternatives to the one presented here.

Although a lot of work has already been done in the area of class diagram satisfiability, there is a still room for improvements in the existing works. The existing works are mostly theoretical or leave out the validation of modeling concepts that we discuss in this thesis. Therefore, whether they are actually suited to that purpose, only an actual implementation can show. The importance of a usable, automatic implementation of a validation approach that has been tested on published examples, as the present, cannot be underestimated. The detail, about the testing of our approach on published examples are discussed in Chapter 5.

Ontology: Parreiras et al. proposed the OntoDSL an ontology based framework for defining DSLs [101]. Their framework uses KM3, Ontology and OCL combination at the metametamodel layer M3, whereas our approach uses MOF at layer M3. Their approach also gives the facility of reasoning of DSLs drawn by using their framework, whereas our approach gives the facility of reasoning of any MOF based DSL which is drawn by using any tool that follows UML 2 3.0.0. The main evidence of this claim is the validation of 303 DSLs that exist in the Atlantic Metamodel Zoo. The detail about the validation of these DSLs and the results are discussed in Chapter 5.

Gašević et al. discussed the use of UML diagrams to construct ontologies [39]. This is a totally different domain than ours, since their work is basically in the domain of Ontology Development Modeling.

Rahmani et al. proposed a mapping from OWL to Ecore [84]. The main idea of their work is to preserve the web knowledge available in the form of ontologies into Ecore. Their work is basically opposite to our work. We translate UML to OWL and their work translates OWL to Ecore.

Alloy: Alloy Analyzer [10] is a tool for the formalization, simulation and verification of UML models. In order to use Alloy, we can either make a metamodel using Alloy script or draw the diagram of a metamodel in a UML modeling tool such as: Magic draw or Topcased and then translate the metamodel into an Alloy script by using UMLtoAlloy translator [89]. However, the UMLtoAlloy translator does not provide the translations of all UML concepts that we discuss in this thesis. These include composition, ordered properties, non-unique associations and the translation of OCL and textual constraints. However, any missing translation of UML concepts or constraints during the translation of UML to Alloy script may lead the whole validation result to become false positive.

OCL: USE [40] is a tool for the validation of UML models using OCL expressions. In this tool, a UML model is taken as an input along with all its possible well-formedness rules in the form of OCL expressions. Then, the parser parses the UML model and OCL expressions, and checks whether the UML model is according to the specifications mentioned in the OCL expression or not. It means that the overall validation process is based on the OCL expressions, and if any well-formedness rule of UML is missing, the overall validation process may produce false positive results.

DPF: The Diagram Predicate Framework (DPF) [87] provides a category theory and graph transformation based formal approach for designing metamodels and models. The DPF workbench [61] allows us to draw metamodels and models based on UML notations. Also, it gives the facility to draw textual symbols on UML models to express constraints such as: [*irr*] for irreflexive, [*comp*] for association composition morphism, [*surj*] for surjective and [*inj*] for injunctive associations. However, the DPF workbench does not provide any facility to validate metamodels. Therefore, we also address this issue in Chapter 4 by translating the textual constraint into an OWL 2 ontology and then validate the ontology by using an OWL 2 reasoner.

TWOUSE Approach: The TWOUSE approach [6] is focused on two areas: The first is the ontology development modeling [78], and the second is the validation of DSLs by using OWL 2 reasoners [100]. The TWOUSE approach proposes the same methodology for the validation of DSLs as presented in this thesis for the validation of metamodels. However, it leaves out some of the important UML concepts such as: composition including unshared and acyclicity constraints, open-world assumptions in the translation of class specializations and class memberships, non-unique associations, ordered properties, and the validation of basic textual constraints like OCL and the constraints proposed by the DPF. Their work is mainly conducted in parallel with our work on the metamodel validation i.e., during the year 2010. Moreover, their work on the validation is limited to the validation of DSLs and does not offer the validation of object diagrams against the class diagrams, nor analyzes the consistency of statechart diagrams with or without invariants.

3.2 Consistency of Class and Statechart Diagrams

Description Logics: The use of ontology languages and description logic in the context of model validation has been proposed in the past by different authors [94, 102, 19, 16]. However, to the best of our knowledge, none of them has addressed the reasoning of the satisfiability of state invariants using OWL 2 DL. These works focus on the problem of class diagrams satisfiability, i.e. whether a class diagram can generate consistent object diagrams or not. Furthermore, the TWOUSE approach [100] is focused on two areas: The first is the ontology development modeling, and the second is the translation and validation of Domain Specific Languages (DSL) by using OWL 2. The TWOUSE approach proposed the same methodology for the validation of DSLs as presented in this thesis. However, their work on validation is limited to the validation of DSLs, and has not yet offered the validation of statechart diagrams with state invariants.

B-Method and CSP: Yeung et al. [104] analyzed control looping to find deadlocks, by translating class diagrams into the B-Method and statechart diagrams into CSP. Their approach does not focus on the consistency of state invariants, the translation is done manually and there is no discussion about the verification method, whether it is manual or automatic.

Object-Z, CSP and FDR: Rasch et al. [85], used Object-Z for the formalization of class diagrams and CSP for statechart diagrams. Their approach analyzes method invocations against the class description and finds deadlocks by running the class and statechart diagram formalization in FDR. Their approach is not focused on analyzing the consistency of state invariants.

π-Calculus: Lam et al. [96] analyzed the consistency of statechart diagrams and class diagrams by using the π-calculus. The translation of UML diagrams to π-calculus is done manually and the consistency is analyzed by running the π-calculus script on the Workbench. Their work is not analyzing the consistency of state invariants.

Program Statements in Transitions: Emil Sekerinski [88] verified the UML statecharts, in which the events are manually translated into generalized program statements, and these statements appear as the body of a transition. The execution of the program statements is based on the assumption that the body of the transition can read or write the values of the class variables.

Alloy: Moaz et al. [68], analyzed the consistency of class and object diagrams by using Alloy. It is a fully automatic approach, in which the class and object diagrams are first translated into a parameterized Alloy module, and then the consistency analysis is done by analyzing the translated Alloy module by using the Alloy Analyzer. Their approach does not yet support statecharts and OCL constraints.

OCL: The state invariants are usually written by using the Object Constraint Language (OCL). It is the widely accepted language for writing constraints over UML diagrams. The reason why the existing research is not focused on analyzing the satisfiability of state invariants is because, in general, OCL is undecidable. However, the undecidability can be avoided, if we limit our approach to known constructs of OCL. The use of a limited subset of OCL to avoid undecidability is not new. For example $OCL - Lite$ [82] uses a limited subset of OCL to express constraints on UML class diagrams. Similarly, in our approach, we use a limited subset of OCL to express state invariants in statechart diagrams.

Bogumila et al. [45] analyzed the consistency of the statechart diagram of a class by writing OCL rules manually, and then execute the OCL rules by using an OCL compiler. This approach is limited to analyzing consistency of statechart diagrams against the class description and not using state invariants. In OCL, the model validation rules must be defined explicitly, based on the syntax of the UML models. In our approach, model validation is defined on the semantic interpretation of the OCL and UML models. The difference is that while OCL must define a large number of well-formed rules for different variations and combinations of model elements, a logic approach requires a smaller number of axioms that are often simpler.

To the best of our knowledge, none of the above mentioned works proposed an automatic translation and consistency checking approach for UML models with state invariants.

In the next section, we discuss the most important related works for the validation of a REST web service interface. The REST web service interface comprises class diagrams and state machine diagrams with state invariants. These diagrams depict the structural and behavioral abstractions of a REST web service. In this thesis, we also validate the REST web service interface in order to show the application of our proposed approach.

3.3 Consistency of REST Web Service Interfaces

Consistency analysis and checking of design models has been studied by a number of researchers in the past, but in the area of web services it has not been researched very extensively, especially in the area of REST web services, we were unable to find any consistency checking approaches. However, in the area of consistency checking for web services, the following works are noteworthy.

MTT: Yin et al. [105] used type theory to verify consistency of web services behavior. Their work addresses web services choreography. It analyzes the structure of service behavior and uses extended MTT, which is a constructive type theory, to formally describe service behavior. The procedures of deductions are given that verify the suitability between services along with discussion on type rules for subtype, duality and consistency of web services behavior.

C & C: In [92] and [93], Tsai et al. presented a specification based robust testing framework for web services. The approach first uses an event driven modeling and specification language to specify web services and then uses a completeness and consistency (C & C) approach to analyze them. Based on these, positive and negative test cases are generated for robustness testing. The approach assumes that web services are specified in OWL-S. The approach aims towards testing of web services. However, C&C analysis is performed on OWL-S specification, which was of interest for us. The approach identifies missing conditions and events, whereas, our approach checks the structure of web services and validates implementation of the service requests.

Model Checking: Xiaoxia [27] verified the service oriented requirements using model checking. The service-oriented computer independent model is used to structure the requirements and then automated model checking is done to do completeness and consistency checking of requirements. It provides a formal definition of completeness checking as a check that all the required services are included in a model. Moreover, it does not gives any specific consistency checking constraint except the requirement relation applied to an example.

Xlinkit Framework: Nentwich et al. [70] presented a static consistency checking approach for distributed specifications. It describes xlinkit framework that provides a distribution-transparent language for expressing constraints between web service specifications. It provides semantics for constraints that allow the generation of hyperlinks between inconsistent elements. The implementation of xlinkit is done on light-weight web service using XML.

CSP: Heckel et al. [42] presented a model-based consistency management approach for web service architectures. They advocate the use of UML class and activity diagrams for modeling web services. The consistency problems are then identified in the UML based development process. For each consistency problem, a partial formalization into a suitable semantic domain is done and a consistency check is defined. The consistency problems identified include syntactic correctness and deadlock freedom. Based on these, those activity diagrams are identified that are relevant to consistency checks. These are partially translated to CSP which are then assembled in a single file and handed over directly to model checker. The paper outlined a good consistency management approach for web services architecture that needs a concrete development.

3.4 Consistency of Class and Object Diagrams

Description Logics and OWL 2: The use of ontology languages and description logic in the context of model validation has been proposed in the past by different authors [94, 79, 65, 84, 102, 19, 11]. However, to the best of our knowledge, none of them has addressed the reasoning of composition, ordered properties and non-unique properties in detail, neither the enforcement of the closed-world restrictions in OWL 2 DL. These works focus on the problem of ontology modeling or on class diagram satisfiability, i.e. if a class diagram can be instantiated or not.

OCL: The validation of UML models using OCL has been discussed by several authors, including [54, 66]. In OCL, the model conformance rules must be defined explicitly based on the syntax of the UML models. In our approach, model conformance is defined on the semantic interpretation of the models. The difference is that while OCL must define a large number of well-formed rules for different variations and combinations of model elements whereas a logic approach requires a smaller number of axioms that are often simpler.

MOVA: MOVA tool [30] provides the facility to draw and validate models against a subset of the UML metamodels. There are a number of limitations in this tool, firstly, this tool produces MOVA specific XMI of UML models, and due to this the models generated by using this tool are not readable by any other modeling tools and vice versa. Secondly, this tool only supports a limited subset of a UML metamodel and does not support the object and class diagram concepts such as: ordered properties and ordered links, composition and non-unique associations and links. Lastly, this tool admittedly does not support the full OCL syntax.

Alloy: Alloy [10] is a tool for the validation of a model against the metamodel. In order to use this tool we need to give a model and its metamodel as an input in the form of an Alloy script. There are some plugins available for example UMLtoAlloy [89], which can transform a UML model into the Alloy script, but the details about whether they can translate UML concepts such as: composition, ordered properties and non-unique associations is missing.

UML Analyzer: UML Analyzer [34], used a text based rulebase for the analysis of UML models. Any missing translation rule of UML constraints during translation may lead the whole validation result to become false positive.

3.5 Consistency of Multiple UML Diagrams

The problem of model merging has been discussed by several authors in the past. For instance, Lutz et al. [64] discussed the merging of models by humans. In their approach, the common model elements in models are calculated manually, and then models are merged using these common model elements. Moreover, the approaches presented at [18, 7, 67] discuss the difference and union of UML models. These approaches propose to take the union of all models in order to perform model merge. We have also used a similar approach while merging UML models by translating the union of all model elements of all models into a single ontology. To the best of our knowledge, none of the above mentioned works does this using OWL 2. Moreover, they do not discuss the automatic discovery of inconsistencies, which occur due to the merging of different versions of a UML model.

Chapter 4

Representation of UML Class Diagrams in OWL 2

In this chapter, we firstly explain our understanding of UML class diagram concepts using logic and show how to translate these concepts into OWL 2. Secondly, we discuss different types of textual constraint languages that we treat in our approach. We also show the equivalent OWL 2 translations of a subset of these languages. This chapter is based on the works presented in Articles I, III and V.

4.1 Introduction

The UML class diagrams are defined as a set of classes and their relationships in the form of generalizations and associations ([77],p.144). The diagrams are used to express the static content or a structure of the system under development. Unfortunately, the semantics of UML are mostly specified semi-formally by means of a textual description [77]. The problem of the automatic validation of these diagrams necessitates the need of a formalization that can be understood by a reasoner. Therefore, in this chapter, we show the formalization and the corresponding OWL 2 translations of UML class diagram concepts that we address in our approach. The basis for the UML metamodel approach is at the same time the core set of constructs of the UML class diagrams [76]. Hence, the formalizations presented here can equally well be viewed as a core construction of MOF/UML-based metamodels. In this chapter, we additionally show the formalization and translation of constraints that can be applied on the UML class diagrams, and also motivate our choice of features that are included in the formalizations.

4.2 Basic Class Diagrams

4.2.1 Class

A class in a class diagram represents a collection of objects which share the same features, constraints and definition. Each class in a class diagram is treated as a *class* in OWL 2. A UML class C is translated in OWL 2 as:

```
Declaration(Class(C))
```

4.2.2 Class Specialization

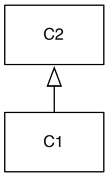

Figure 4.1: Class Specialization

Class specialization is reduced to the set inclusion. We represent the fact that a UML class C1 is a specialization of UML class C2 with the condition:

$$C1 \subseteq C2.$$

In this case, we say that C2 is a superclass of C1. If two classes C1 and C2 have a common superclass, or C2 is the superclass of C1 we say that they are in a specialization relation. The specialization relation $C1 \subseteq C2$ is translated in OWL 2 as:

```
SubClassOf( C1 C2 )
```

4.2.3 Disjoint Classes

We assume that an object cannot belong to two classes, except when these two classes are in a specialization relation. In our semantic interpretation of a UML class diagram, it is equally important to denote the facts that two classes are not in a specialization relation. This is due to the fact that in object-oriented models an object cannot belong to two classes, except when these two classes are in a specialization relation. We represent the fact that UML class C1 and UML class C2 are not in a specialization relation with the condition:

$$C1 \cap C2 = \emptyset$$

With this condition, an object cannot belong to these two classes simultaneously. Due to the open-world assumption used in Description Logic, we need to explicitly state this fact in OWL 2 as:

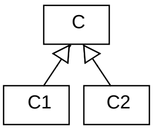

Figure 4.2: Disjoint Classes

```
DisjointClasses( C1 C2 )
```

It is necessary for all classes to either explicitly or implicitly declare that they are disjoint to classes that they do not share model elements with. However, given that these axioms need to take the entire class hierarchy into account, how to efficiently generate this information is not immediately apparent. We have declared every class equivalent to the union of its subclasses and direct instances — or if it has no subclasses, equivalent to its direct instances. We have therefore provided enough information for a reasoner to be able to deduce which direct instances any given class is made up of. Correctly declaring the direct instances disjoint is consequently enough information for a reasoner to infer if any given pair of classes are disjoint.

A pair of classes are disjoint if neither is a superclass to or a subclass of the other, and they do not share any subclasses. The direct instance class never has any subclasses, so deciding whether it is disjoint to another class merely requires that we verify that the class in question is not a superclass of the direct instance. This make it necessary to only traverse part of the class hierarchy. Furthermore, this approach limits the amount of generated axioms to one per class. As classes inherit the properties of superclasses, it is only necessary to include top level classes and its superclasses' unshared direct subclasses in the axiom.

4.2.4 Associations

The association is another fundamental concept of UML class diagrams and it represents a basic relationship between the instances of two or more classes. We represent a UML directed binary association A from class C1 to C2 as a relation:

$$A : C1xC2$$

Each association in a class diagram contains two properties, namely *Domain* and *Range*, that represent each end of the association. In our example, $C1$ is the domain and $C2$ is the range of the association A.

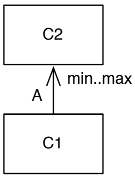

Figure 4.3: Association

A UML association A from UML class $C1$ to $C2$ is represented in OWL 2 as:

```
Declaration(ObjectProperty(A))
ObjectPropertyDomain( A C1 )
ObjectPropertyRange( A C2 )
```

4.2.5 Multiplicity

A UML association in a class diagram is annotated with a positive number; this number indicates the multiplicity of an association. Association multiplicity describes the number of allowable objects of a range class to link with the object of a domain class. The multiplicity of an association defines additional conditions over this relation:

$$\#\{y|(x,y) \in A\} \geq min$$
$$\#\{y|(x,y) \in A\} \leq max$$

We map the multiplicity of a UML association into OWL 2, by defining the domain class of an association as a subclass of a set of classes, which relates with the same property and the given cardinality.

The UML association A from class $C1$ and $C2$ having a multiplicity constraint of (min, max) is represented in OWL 2 as:

```
SubClassOf( C1 ObjectMinCardinality( min A ) )
SubClassOf( C1 ObjectMaxCardinality( max A ) )
```

4.2.6 Bidirectionality

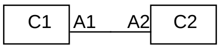

Figure 4.4: Association Bidirectionality

In UML, the associations that share opposite domain and range form a bidirectional association. For example if $A1$ and $A2$ are UML associations and both associations share opposite domain and range, then both associations are considered as bidirectional of each other, such that:

$$A1 = (x, y)|(y, x) \in A2$$

The UML bidirectionality between the associations $A1$ and $A2$ is expressed in OWL 2 as:

```
InverseObjectProperty( A1 A2 )
```

4.2.7 Association Generalization

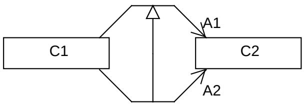

Figure 4.5: Association Generalization $A1 \subseteq A2$

An association can be generalized by another association. The association generalization is also known as association subsetting. Association subsetting allows the specialization of an existing association, with new characteristics while retaining its existing features, such as: domain and range. However, we can reassign a domain and a range of a subassociation, provided that the new domain and the range of a subassociation are the subclasses of the domain and the range of a parent association. Each instance of the specialized association is also an instance of the original property. Therefore, elements that are a part of its slot should be a part of the original association slot. The association subsetting between association $A1$ and $A2$ is defined as:

$$A1 \subseteq A2$$

Where $A1$ is a subassociation of $A2$, and translated in OWL 2 as:

```
SubObjectPropertyOf( A1 A2 )
```

4.2.8 Class Attributes

The class attributes depicting variables of various datatypes such as string, integer or boolean are also considered as relations. In this case, the range of the relation A belongs to the set D representing the datatype as:

$$\forall x, y : (x, y) \in A \implies y \in D$$

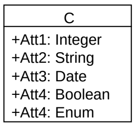

Figure 4.6: Class Attributes

Attributes usually have a multiplicity restriction to one value. The attributes of a UML class in a class diagram are translated in OWL 2 as a *DataProperty*. In OWL 2, the data property uses datatype in its range. The datatype can be xsd:boolean, xsd:string, xsd:int and other datatypes (shown in [20],Table 3). We map attributes that use basic types by declaring a data property with the attribute's name. An attribute is a required component of its class. Consequently, the data properties describing attributes have an exact cardinality of one. The attribute *Att* of the UML class *C* having any of the above mentioned *DataType* is translated in OWL 2 as:

```
Declaration(DataProperty( Att ))
SubClassOf(C DataExactCardinality(1 Att ))
DataPropertyDomain( Att C )
DataPropertyRange( Att DataType )
```

4.2.9 Data Enumeration

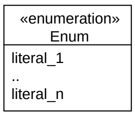

Figure 4.7: Enumeration Datatype

Enumeration is a kind of datatype, whose instances are user-defined enumeration literals ([77],p.67). The enumeration *Enum* is declared by using a `DatatypeDefinition` axiom in OWL 2 DL. The class attribute *Att* having a datatype *Enum*, means:

$$\forall x, y : (x, y) \in Att \implies y \in Enum$$

Where *Enum* being a set of enumeration literals $\{(literal_1), ..., (literal_n)\}$ is represented in OWL 2 as:

```
DataPropertyRange(Att DataOneOf("literal1"^^datatype ..))
```

4.2.10 Composition

In composition, an object of a class is made up of parts that are the objects of another class. To give a formal definition of composition, we use a single predicate *owns* to keep track of the composition relationships. If C1 owns C2 via a composition association P, and P is the property from C1 to C2, then:

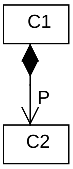

Figure 4.8: Composition C1 owns C2

$$\forall x, y : (x, y) \in P \implies x \in C1$$
$$\forall x, y : (x, y) \in P \implies y \in C2$$

$$P \subseteq owns$$

Composition relationships are defined in UML by two constraints, exclusive ownership and acyclicity. Exclusive ownership means that an object can have only one owner:

$$\forall x, y, z : (x, z) \in owns \ and \ (y, z) \in owns \implies x = y$$

Acyclicity means that an object cannot transitively become an owner of itself. A situation where an object x owns y, y owns z and z owns x is disallowed. A necessary and sufficient condition for acyclicity of *owns* is that the transitive closure of the relation is irreflexive. We can define the transitive closure of *owns*, in the following way:

$$\forall x, y, z : (x, y) \in owns \ and \ (y, z) \in owns \implies (x, z) \in owns$$

Irreflexivity of the transitive closure is then simply expressed as:

$$\forall x : x \in \Delta_i \implies (x, x) \notin owns$$

In order to translate the composition association mentioned above into OWL 2, we first define an object property named "P" from class $C1$ to class $C2$.

```
Declaration( ObjectProperty( P ) )
ObjectPropertyDomain( P C1 )
ObjectPropertyRange( P C2 )
```

Next, we consider the exclusive ownership constraint of the composition on the owning end of a composite relationship. To implement the single owner requirement of a composition relationship in OWL 2, we have firstly, defined the global property *owns* as:

```
InverseFunctionalObjectProperty( owns )
```

The inverse functional property will restrict the individuals of containing class to link with more than one individuals of owning class. Secondly, we make the composite relationship ”P”, a subproperty of the global property *owns*:

```
SubObjectPropertyOf( P owns )
```

Moreover, in order to capture the acyclic requirement of the composition, we make *owns* transitive and irreflexive at the same time. Transitivity will capture the self ownership and irreflexiveness will disallow the individual becoming an owner of itself. This is equivalent to saying that the transitive closure of the ownership property is irreflexive. However, it is not possible to combine a cardinality restriction with transitive properties [51]. In OWL 2 DL, if we could do so, the logic system would no longer be decidable, and we would not be able to use a fully automatic reasoner to carry out validation. To solve this problem, we translate transitivity in Semantic Web Rule Language (SWRL) and irreflexivity in OWL 2. The transitivity of the property *owns* is written in SWRL as:

$$owns(?x, ?y) \wedge owns(?y, ?z) \implies owns(?x, ?z)$$

and irreflexivity of *owns* is translated in OWL 2 as:

```
IrreflexiveObjectProperty( owns )
```

4.3 Class Diagrams with DPF Constraints

The Diagram Predicate Framework (DPF) [87] provides the category theory and graph transformation based formal approach for designing metamodels and models by using the DPF workbench [61]. The DPF workbench allows us to draw metamodels and models based on UML notations. Also, it gives the facility to draw textual symbols on UML models to express constraints such as: [*irr*] for irreflexive, [*comp*] for association composition morphism, [*surj*] for surjective and [*inj*] for injunctive associations. An example of a

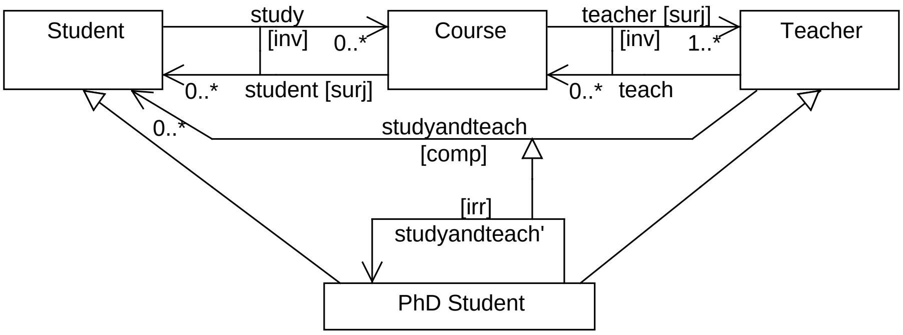

Figure 4.9: The UML class diagram using DPF constraints depicting a scenario that a PhD student cannot enroll in a course that he is teaching by himself.

UML class diagram with DPF constraints is shown in Figure 4.9. In this section, we show the formalization and OWL 2 translations of a subset of textual constraints proposed by DPF. All OWL 2 translations we propose in this section are based on the semantics of textual constraints provided in [87]. The OWL 2 translations shown in this section are later passed to an OWL 2 reasoner for the validation of UML models including textual constraints.

4.3.1 Irreflexive

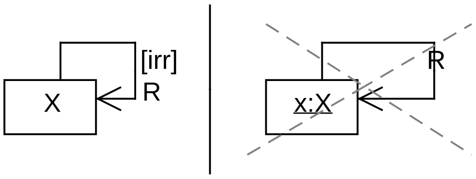

Figure 4.10: Irreflexive Constraint

An irreflexive [irr] constraint on an association disallows an object of a class to link with itself, i.e., if there exists a concept X which has a self association R and it is irreflexive, then:

$$\forall x \in X : x \notin R(x)$$

The association R having an irreflexive constraint $[irr]$ is translated in OWL 2 as:

```
IrreflexiveObjectProperty( R )
```

4.3.2 Injective

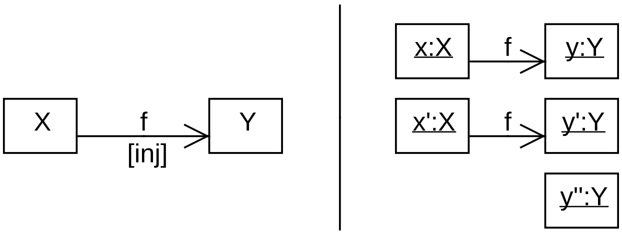

Figure 4.11: Injective Constraint

The injective $[inj]$ constraint of an association restricts an object of a domain class to link to at most one object of a range class. The injective constraint is also known as "One-to-One". If we have the association f with the injective constraint from class X to class Y, then:

$$\forall x, x' \in X : f(x) = f(x') \implies x = x'$$

The injective property is translated in OWL 2 by using axioms $InverseFun-$ *ctionalObjectProperty* and *FunctionalObjectProperty*. The inverse functional object property disallows an object of a range class to link with more than one object of a domain class, whereas, the functional object property disallows an object of a domain class to link with more than one object of a range class. The injective association f is translated in OWL 2 as:

```
InverseFunctionalObjectProperty(f)
FunctionalObjectProperty(f)
```

4.3.3 Jointly Injective

The jointly injective $[ji]$ constraint is a collection of injective associations, i.e., if we have three classes X, Y and Z and there exists two injective associations f and g, where f is from X to Y, and g is from X to Z, and both associations are jointly injective then:

$$\forall x, x' \in X : f(x) = f(x') \ and \ g(x) = g(x') \implies x = x'$$

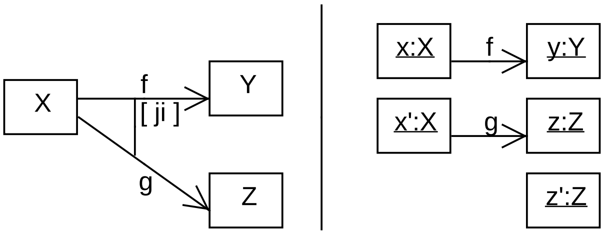

Figure 4.12: Jointly Injective Constraint

In order to translate jointly injective associations in OWL 2, we first declare an object property *JInjective* in OWL 2, and then, make both injective associations a *SubObjectPropertyOf* of *JInjective*. Since, both injective associations share the same domain, the domain of *JInjective* will be the same as the domain of injective associations. However, to join both injective associations, the range of *JInjective* is the union of the range classes of both injective associations. The translation of jointly injective associations mentioned above into OWL 2 is as follows:

```
Declaration(ObjectProperty(JInjective))
SubObjectPropertyOf(f JInjective)
SubObjectPropertyOf(g JInjective)
ObjectPropertyDomain(JInjective X)
ObjectPropertyRange(JInjective ObjectUnionOf(Y Z))
```

4.3.4 Surjective

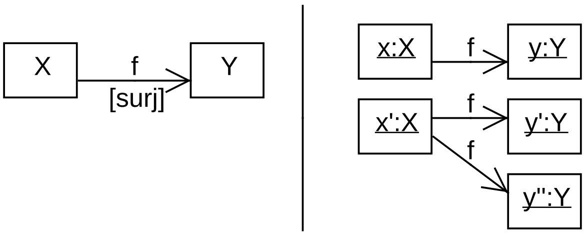

Figure 4.13: Surjective Constraint

The surjective constraint [*surj*] of an association allows an object of a domain class to link to many objects of a range class. The surjective constraint is also known as ”One-to-Many”. If an association f from class X to class Y is surjective, then for every individual in Y, there is at least one individual in X such that:

$$f(X) = Y$$

One-to-many is the by default feature of an object property in OWL 2. The surjective constraint on the association f mentioned above is translated into OWL 2 as:

```
Declaration(ObjectProperty(f))
ObjectPropertyDomain(f X)
ObjectPropertyRange(f Y)
```

4.3.5 Jointly Surjective

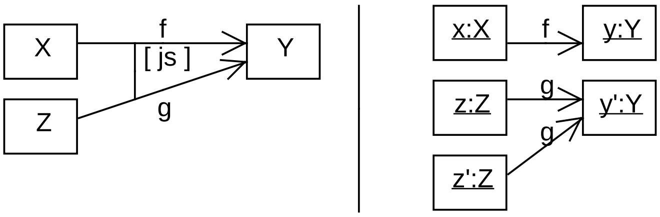

Figure 4.14: Jointly Surjective Constraint

Jointly surjective $[sj]$ is a collection of surjective associations, i.e., if we have three classes X, Y and Z and there exists two surjective associations f and g, where f is from X to Y, and g is from Z to Y, and both associations are jointly surjective then:

$$f(X) \cup g(Z) = Y$$

To translate the jointly surjective associations in OWL 2, we first declare an object property *JSurjective* in OWL 2, and then, make both surjective associations i.e., f and g, a *SubObjectPropertyOf* of *JSurjective*. To join both surjective associations, the domain of *JSurjective* is the same as the domain of the surjective associations, and the range is the union of the range of both surjective associations. The OWL 2 translation of jointly surjective associations is as follows:

```
Declaration(ObjectProperty(JSurjective))
SubObjectPropertyOf(f JSurjective)
SubObjectPropertyOf(g JSurjective)
ObjectPropertyDomain(JSurjective ObjectUnionOf(X Z))
ObjectPropertyRange(JSurjective Y)
```

4.3.6 Bijective

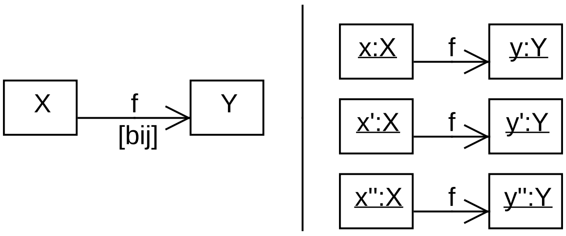

Figure 4.15: Bijective Constraint

The bijective constraint [*bij*] of an association is a combination of both surjective and injunctive constraint, i.e, if an association f from class X to Y is bijective, then for every object in Y, there is exactly one object in X. Since, the bijective constraint is a combination of both injective and surjective constraint, the OWL 2 translation of a bijective association contains the axioms discussed in the translation of both surjective and injective associations. The bijective association f from class X to class Y is translated in OWL 2 as:

```
InverseFunctionalObjectProperty(f)
FunctionalObjectProperty(f)
ObjectPropertyDomain(f X)
ObjectPropertyRange(f Y)
SubClassOf(X ObjectExactCardinality(1 f ))
```

4.3.7 Composition Morphism

The composition morphism [*comp*] allows two or more consecutive associations to connect with each other and form a chain. If we have three classes X, Y and Z and similarly an association f between classes X and Y, and there exists an association g between classes Y and Z, then:

$$\forall x \in X : f; g(x) = \cup\{g(y)|y \in f(x)\}$$

The composition morphism between associations f and g is translated in OWL 2 as:

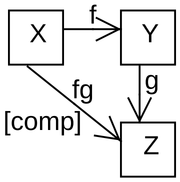

Figure 4.16: Composition Morphism

```
Declaration(ObjectProperty(f))
Declaration(ObjectProperty(g))
Declaration(ObjectProperty(fg))
```

However, according to the restrictions on the axiom closure in OWL 2 DL ([22], Sec. 11.2), to prevent cyclic definitions involving object subproperty axioms with property chains, it is not possible in OWL 2 DL to use *ObjectPropertyChain* axiom with *SubObjectPropertyOf*. If we could do so, the logic system would no longer be decidable, and we would not be able to use a fully automatic reasoner to carry out validation. To solve this problem, we have expressed the OWL 2 object chaining i.e. *SubObjectPropertyOf(ObjectPropertyChai-n(f g) fg)* in Semantic Web Rule Language (SWRL) [49]. The [*comp*] constraint mentioned above is translated into SWRL as:

$$f(?x, ?y) \wedge g(?y, ?z) \implies fg(?x, ?z)$$

4.3.8 Example

The class diagram shown in Figure 4.9 is translated to OWL 2 as follows:

```
Declaration(Class(Student))
Declaration(Class(Course))
Declaration(ObjectProperty( study ))
ObjectPropertyDomain(study Student)
ObjectPropertyRange(study Course)
Declaration(ObjectProperty( hasStudent ))
ObjectPropertyDomain(hasStudent Course)
ObjectPropertyRange(hasStudent Student)
InverseObjectProperties( study student )
SubClassOf(Student ObjectMinCardinality(2 study))
SubClassOf(Student ObjectMaxCardinality(1 study))
```

Since the constraints proposed by DPF apply restrictions on objects, the conformance of the objects against these constraints is discussed in Chapter 8.

4.4 Class Diagrams Including OCL Constraints

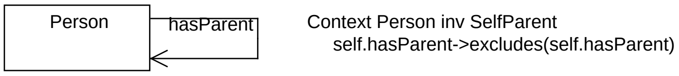
Person
hasParent
Context Person inv SelfParent
self.hasParent->excludes(self.hasParent)

Figure 4.17: A UML class diagram with OCL constraints.

The UML specification proposes the use of Object Constraint Language (OCL) [74] to define constraints on UML models, such as the restrictions on the values of object attributes and the restrictions on the existence of objects by using a multiplicity constraint of an association. These constraints are combined using boolean operators.

The OCL constraints may also have inconsistencies. According to Wilke and Demuth [103], 48.5% of the OCL constraints used for expressing the well-formedness of UML in OMG documents are erroneous. The erroneous OCL constraints may cause a context class in a class diagram to become unsatisfiable. In order to identify the inconsistencies in OCL constraints, we need to do the reasoning of these constraints. But unfortunately, in general, OCL is not decidable. However, we can avoid undecidability by restricting our approach to a reduced fragment of the full OCL. Therefore, in order to use the reasoners for checking the inconsistencies in OCL constraints, we use a reduced subset of OCL which is limited to the constructs of multiplicity, attributes value and boolean operators. In this section, we discuss and translate the different types of OCL constructs supported in our approach. The grammar of OCL supported in our approach is shown in Figure 4.18.

4.4.1 Linking OCL Constraints with Classes in OWL 2

Each OCL constraint of a class diagram comprises a number of elements such as context, name and the invariant. These elements hold the name of a constrained class, the name of an invariant and the constraint in OCL, respectively. The OCL constraints are used to apply restrictions on object memberships, i.e. an object can only belong to a particular class if it fulfills the conditions applied by the OCL constraint of that class, such that:

⟨OCL-expression⟩	::=	⟨cond-expr⟩ (⟨logic-op⟩⟨cond-expr⟩)*
⟨logic-op⟩	::=	*and* | *or*
⟨cond-expr⟩	::=	⟨ref⟩ →size()⟨relational-operator⟩⟨integer-literal⟩ | ⟨ref⟩⟨relational-operator⟩⟨primitive-literal⟩ | ⟨ref⟩ →isEmpty() | ⟨ref⟩ → notEmpty() | ⟨ref⟩ →excludes(⟨ref⟩)
⟨ref⟩	::=	self.⟨identifier⟩
⟨identifier⟩	::=	′{⟨characters⟩} | 0..9 {0..9}′
⟨relational-operator⟩	::=	< | <= | > | >= | <> | =
⟨primitive-literal⟩	::=	⟨boolean-literal⟩ | ⟨integer-literal⟩ | ⟨string-literal⟩ | null
⟨boolean-literal⟩	::=	true | false
⟨integer-literal⟩	::=	0..9 {0..9}
⟨string-literal⟩	::=	′{⟨characters⟩}′

Figure 4.18: The grammar of the supported OCL fragment.

$$Context \equiv Invariant$$

4.4.2 Attribute Constraints

The value of the attribute is accessed in OCL by using a keyword $self$ or by using a class reference ([74],p.15), the value constraint of the attribute *Att* is written in OCL as `self.Att=Value`, meaning:

$$\{x|(x, Value) \in \mathit{Att}\}$$

where $Value$ represents the attribute value. The restriction on the value of the attribute is translated in OWL 2 by using the axiom `DataHasValue`. The OCL attribute value constraint `self.Att=Value` is translated to OWL 2 as:

```
DataHasValue(Att "Value"^^datatype )
```

In this translation, *Att* is the name of the attribute, $Value$ is the value of the attribute, and *datatype* is the datatype of the attribute $Value$.

4.4.3 Multiplicity Constraints

The multiplicity of an association is accessed by using the $size()$ operation in OCL ([74],p.144). The multiplicity constraint on the association A in OCL is written as `self.A->size()=Value`, where $Value$ is a positive integer and represents the number of allowable instances of the range class of the association A. We can use a number of value restriction infix operators with the $size()$ operation, such as =, >=, <=, < and >. The multiplicity constraint on an association A is defined as:

$$\{x|\#\{y|(x, y) \in A\}OP\ Value\}$$

Where OP is the infix operator and $Value$ is a positive integer. The translation of the $size()$ operation in OWL 2 is based on the infix operator used with the $size()$ operation, such as:

- ”$size()$ >=” or ”$size()$ >” translated using the `ObjectMinCardinality` axiom.
- ”$size()$ <=” or ”$size()$ <” translated using the `ObjectMaxCardinality` axiom.
- ”$size()$ =” translated in OWL 2 using the `ObjectExactCardinality` axiom.

For example, the OCL constraint `self.A->size()=Value`, in which A is the name of an association and $Value$ is a positive integer, is translated to OWL 2 as:

```
ObjectExactCardinality(Value A)
```

Furthermore, the constructs *isEmpty* and *notEmpty* represent $size() = 0$ and $size() > 0$ respectively. The invariant `self.A->isEmpty()` is translated in OWL 2 as:

```
ObjectExactCardinality(0 A)
```

and similarly the invariant `self.A->notEmpty()` is translated in OWL 2 as:

```
ObjectMinCardinality(1 A)
```

The OCL constraints over the multiplicity of an association is further extended by using construct *excludes*. This construct is used to apply restriction on the objects of a domain class of an association to not to link with some specific objects of a range class. For example, if we consider an example that an object of a class cannot link with itself by using a link of an association A, such that `self.A->excludes(self.A)`, the construct *excludes* in this specific case in translated in OWL 2 as:

```
IrreflexiveObjectProperty( A )
```

In OWL 2, an association that is declared $Irreflexive$ disallows an object of its domain or range class to link with itself.

4.4.4 Boolean Operators

The constraints in a state invariant are written in the form of a boolean expression, and joined by using the boolean operators, such as "*and*" and "*or*"([74],p.144).

The binary "*and*" operator evaluates to true when both boolean expressions Ex_1 *and* Ex_2 are true. In our translation, this is represented by the intersection of the sets that represent both expressions, i.e.:

$$Ex_1 \cap Ex_2$$

This is represented in OWL 2 as:

```
ObjectIntersectionOf(Ex1 Ex2)
```

The binary "*or*" operator evaluates to true when at least one of the boolean expression Ex_1 *or* Ex_2 is true. In our translation, this is represented by the union of the sets that represent both expressions, such as:

$$Ex_1 \cup Ex_2$$

This is represented in OWL 2 as:

```
ObjectUnionOf(Ex1 Ex2)
```

4.4.5 Example

The OCL invariant $self.hasParent- > excludes(self.hasParent)$ has the context class $Person$ shown in Figure 4.17 representing a condition that a person cannot become a parent of itself. The class diagram with the OCL constraint is translated in OWL 2 as:

```
Declaration(Class(Person))
Declaration(ObjectProperty( hasParent ))
ObjectPropertyDomain(hasParent Person)
ObjectPropertyRange(hasParent Person)

IrreflexiveObjectProperty(hasParent)
```

The details about how the inconsistent OCL constraint makes a UML model inconsistent and how to detect the inconsistencies in OCL constraints using OWL 2 reasoners is discussed in Chapter 6. Furthermore, the conformance of the objects against the OCL constraints specified with a class diagram is discussed in Chapter 8.

4.5 Conclusion

In this chapter, we presented the translations of UML class diagrams with or without constraints into OWL 2. These translations allow us to validate UML class diagrams with or without constraints using reasoners. In the next chapter, we will show how to validate the metamodels or class diagrams using the translations proposed in this chapter. Also, we discuss the implementation of translations in the form of a translation tool. The translation tool will allow us to automatically translate UML diagrams to OWL 2. The output of the translation tool is later on passed to a reasoner for the reasoning of translated diagrams.

Chapter 5

Application: Metamodel Validation

In this chapter, we present an application of the UML class diagram translations proposed in Chapter 4 and discuss how to validate the UML class diagrama or metamodels using OWL 2 reasoners. Also, we exemplify the proposed approach by validating more than 300 metamodels comprising of up to thousands of model elements available in an online repository called the *Atlantic Metamodel Zoo* [1]. This chapter is based on the work presented in Article II.

5.1 Introduction

Each software model created during a software development process is described using a particular modeling language such as the UML or a domain-specific language. The definition of a modeling language is given in the form of a metamodel by using a metamodeling language or a language to define modeling language.

A metamodel represents constraints on how concepts in a model can be related to each other, such as multiplicity, domain and range, composition and subset constraints. A metamodel, like any other software artifact, may contain errors. As an example, let us assume that we are creating a metamodel for a statechart language that includes concepts such as simple states and composite states that can contain other states and transitions between states. A fragment of such a metamodel is shown in the first half of Figure 5.1. This metamodel contains a rather obvious contradiction: there is a property called outgoing with its minimum multiplicity larger than its maximum multiplicity. Such contradictions mean that there are no valid models that can make use of this property.

Another example of a metamodel containing a contradiction is shown

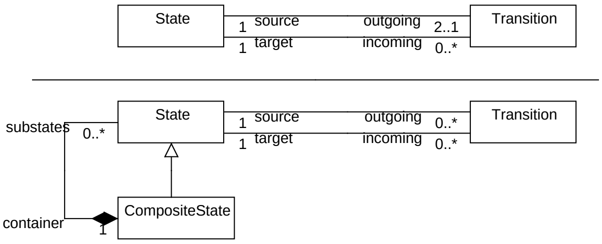

Figure 5.1: Examples of invalid metamodels. Top: Invalid due to the multiplicity error. Bottom: Invalid due to the composition error.

in the second half of Figure 5.1. Here, the multiplicity restriction on the container property leads to each instance of CompositeState having to be in a composite relationship with another instance of CompositeState. This violates one of the composition constraints — elements having only one owner and there being no cycles in the composition, given a finite set of instances of CompositeState. Since models are finite, any conforming model will contain a violation of the composition constraints, as exemplified by the model in Figure 5.2 where the element cs violates the constraint on cycles by owning itself.

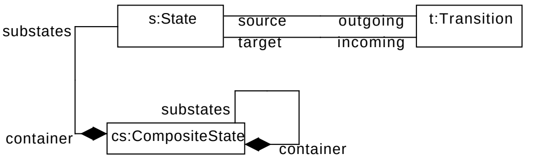

Figure 5.2: An erroneous model conforming to the bad metamodel in the bottom of Figure 5.1.

As shown in the examples, metamodels can contain errors or contradictions. We consider it necessary to validate a metamodel to ensure that there are no such problems before they are used. This requires a formal definition of a metamodeling language and an approach to reason about the formalized metamodels.

5.1.1 Validation Approach

To show the application of the translations proposed in the previous chapter, in this chapter we present an approach and a tool to validate metamodels by formalizing the semantics of metamodels in terms of Description Logic (DL) [13] and represent the DL in terms of an ontology language, in our case the OWL 2 Web Ontology Language [21].

By creating a mapping between a metamodeling language (UML) and a DL, we obtain important benefits:

- We provide a formal and unambiguous definition of the metamodeling concepts that is independent of a specific model repository. This may help ensure interoperability between metamodeling tools.
- Use the same language to represent the constraints that apply to metamodels and the constraints that metamodels impose on models.
- We can use existing reasoning tools to analyze and validate metamodels and detect problems.

The translation from a UML metamodel to an OWL 2 ontology is done automatically using a model transformation. Once the ontology of a metamodel has been generated by using our transformation, we use a reasoner to check the ontology consistency and satisfiability.

5.1.2 Consistency Analysis of Metamodels

In our work, the consistency of metamodels is defined with the assumption that there is a nonempty set $\Delta^{\mathcal{I}}$ called the object domain containing all the possible objects in our domain. We propose that a UML metamodel depicting a class diagram is interpreted as a number of subsets of $\Delta^{\mathcal{I}}$ representing each class in the metamodel and as a number of conditions that need to be satisfied by these sets. A metamodel is consistent if each class in a metamodel can be instantiated i.e., if C is a class in the metamodel and $C \subseteq \Delta^{\mathcal{I}}$, then $C \not\equiv \perp^{\mathcal{I}}$ must hold.

5.2 Implementation

We have implemented an automatic transformation from UML metamodels to OWL 2. The transformation is implemented using the model-to-text transformation tool in the Eclipse Modeling Framework (EMF) by using MOFScript [5, 8]. EMF is an open source software and provides an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle.

MOFScript is an Eclipse plugin for the model to text transformation, for example, to generate code or documentation from models. It provides a framework that allows generating text or code from models based on any kind of metamodel and its instances. Also, the MOFScript language is currently a candidate in the OMG RFP process on MOF Model to Test Transformation. MOFScript transformations define a set of rules as to how metamodel elements should be translated through print statements. The metamodel input format in our implementation is UML XMI 2.1 parsed according to UML 2 3.0.0.

We used the MOFScript subproject for code generation of the UML class and object diagrams. This subproject is built for the generation of MOF-based models, and has the ability to generate text from any MOF-based model, for example, UML models or any kind of domain model, such as, metamodels expressing the specification of domain specific languages. This subproject also has the ability to specify basic control mechanism, such as, loops and conditional statements. It also provides the facility for string manipulation and generates the output of expressions referencing a model to a specified output file. In our case, the output files are in both OWL 2 functional syntax and in Manchester Syntax that carries the OWL 2 translation of UML models, whereas, the SWRL rules are written in a sperate $.OWL$ file wherever it is required, for example, SWRL is required for the translation of composition transitivity.

5.3 Validation of Metamodels at Atlantic Zoo

To evaluate our approach, we conducted a number of experiments where we translated the metamodels from the Atlantic Metamodel Zoo to OWL 2 ontologies and then validated them by using OWL 2 reasoners.

5.3.1 Selection of Evaluation Data

The Atlantic Metamodel Zoo is a library consisting of 303 metamodels at the time of writing, maintained by the AtlanMod team. The AtlanMod team does not state the criteria for the inclusion of a metamodel in the zoo, but most of the metamodels seem to have been entered by modeling and computer science researchers. The metamodels are available in different languages, including UML 2, OWL, KM3, among others.

The choice of the Atlantic Metamodel Zoo as evaluation suite has both advantages and disadvantages. One disadvantage is that the repository is not completely recent; many of the published metamodels date back to 2005 or 2006. Another disadvantage is that many of the metamodels presumably have not been used in practice. The high rate of metamodels containing errors, nearly half of the repository, is an indication of this, since the erroneous metamodels would most likely have been weeded out if the metamodels had been used in practical work.

A related disadvantage is that a part of the errors found seem not to be conceptual errors introduced by humans, but rather artifacts of conversion processes from other formats. This is for example the case with the matemodels listed in table A.5, which are presumably originally written in OWL and subsequently converted to UML.

However, the use of the Atlantic Metamodel Zoo also has advantages that we consider outweigh the disadvantages. The main advantage of Atlantic Zoo is that it is publicly available on the Internet. While it would be optimal to test our approach on live, production use metamodels, it would be quite difficult to obtain a sufficient amount of such metamodels. This is due to the fact that businesses understandably often consider their metamodels part of business sensitive intellectual property, and as such they are reluctant to release them to researchers.

While it would have been better to use production metamodels for evaluation, the high rate of errors in the Atlantic Metamodel Zoo can from a testing perspective be seen as an advantage rather than a disadvantage. If we instead had reported 300 flawless production metamodels, we would not have been as convinced of the ability of our approach to actually catch errors. The high error rate of the Atlantic Metamodel Zoo is unfortunate from a metamodeling perspective, but from the perspective of developing a diagnostic tool it is helpful. The same can be said for the errors caused by conversion artifacts in the Atlantic Metamodel Zoo. Instead of being a nuisance, they rather show that our approach can catch both automatically introduced errors and those introduced by humans.

5.3.2 Method

We ran our OWL 2 translator through all the examples in the Atlantic Metamodel Zoo and used the reasoners Pellet and HermiT to validate the generated ontologies. Given a source metamodel, the process consists of three steps:

1. Run the source metamodel through the translator.
2. Check the satisfiability of the generated ontology. If the ontology is inconsistent, it will also be reported in this step.
3. If the reasoner reports a problem, an explanation is generated by the reasoner in the form of a list of violated axioms. It is necessary to manually inspect the generated explanation and usually also a diagram of the metamodel in question since the cause of the problem in many cases is not obvious from the explanation.

In our experiment, we ran HermiT 1.2.1 from the command line and used the options `-k` and `-U` to check the consistency and concept satisfiability, respectively. In the case of Pellet, we used version 2.0.1 and later versions from the command line and used the options `consistency` and `unsat` to check the consistency and concept satisfiability, respectively.

5.3.3 Validation Results

Out of the 303 metamodels, 149 were found to be invalid. The errors are reported in Appendices A.2–A.4. For each metamodel, one or more classes that are involved in the constraint violation is listed. The class list is not exhaustive. The metamodels may be listed in more than one category.

As can be seen from the Appendixes, the most common error by far is a violation of the one ownership constraint of composition (Appendix A.4). An example of such violation is shown in Figure 5.3.

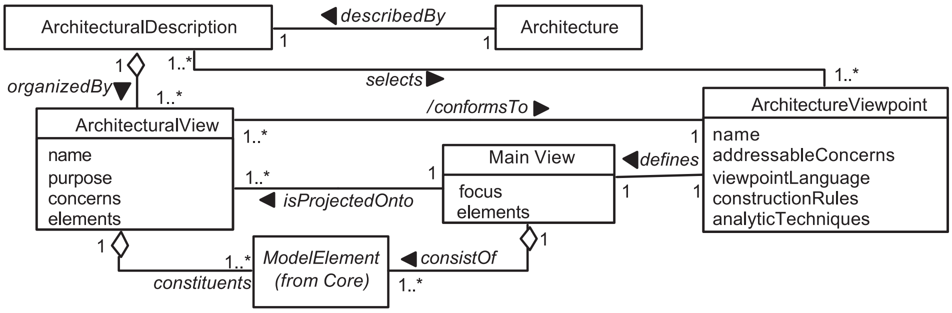

Figure 5.3: (As in [53, 1]) The metamodel fragment describes an Architectural Description, containing an unsatisfiable class ModelElement. This metamodel is unsatisfiable due to the violation of the single owner restriction of composition.

There is also a fair number of metamodels violating the acyclic composition constraints. An example of such a violation is shown in Figure 5.4 and the list of all violations of acyclic composition constraints found by using our approach is shown in Appendices A.1 and A.3.

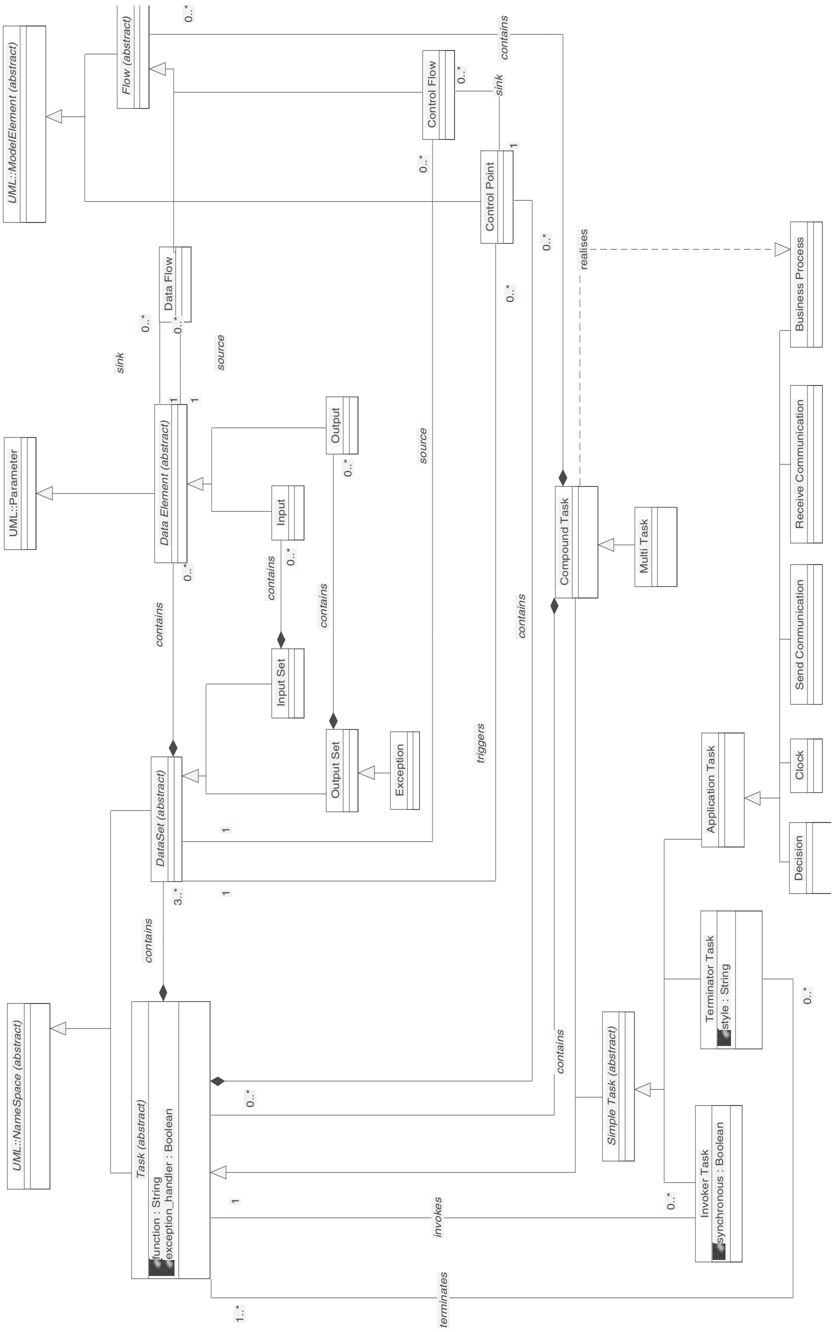

Figure 5.4: (As in [17, 1]) The metamodel fragment describes a Business Process Model with unsatisfiable classes Task and CompoundTask due to the composition cycle error.

A number of metamodels erroneously declare the same class name several times, which will generate conflicting restrictions on the class (Appendix A.5). These metamodels all seem to originally have been written as OWL ontologies, where this would not have been an error.

The metamodels that were not found to have problems are not guaranteed to be free of errors, they are just free from inconsistencies that our approach can detect.

A very useful future improvement would be to enhance the way problems in ontologies are reported. As it is now, the reasoner can give a list of violated OWL 2 axioms. However, the relationship between UML concepts and OWL 2 axioms is not so obvious that it is possible to immediately point out the cause of the problem based on these violations without manual inspection of the problematic metamodel. It would greatly add to the usefulness of the method to have some sort of automated discovery of the cause of violations.

5.4 Performance Test for Reasoners

We ran our OWL 2 translator through all the examples in the Atlantic Metamodel Zoo [1] and used the reasoners Pellet and HermiT to validate the generated ontologies. In our experiments, we ran HermiT version 1.2.5.929 and Pellet version 2.2.0. We evaluated the reasoners in terms of maturity, performance, expressiveness, and their problem reporting mechanism.

5.4.1 Expressiveness

The expressiveness is evaluated in two aspects: the concepts contained in the metamodels and the Description Logic which the reasoners are based on.

On one hand, the metamodels in the Atlantic Zoo are simple in the sense that they do not cover all the concepts that are within the ability of the UML metamodeling language. For example, there are no metamodels that contain any ordered properties, ordered composition or ordered subset properties. The constraints of these concepts are expressed in SWRL rules.

On the other hand, we noticed problems with UML composition: translating composition to OWL 2 requires the transitivity axiom and irreflexive axiom, whereas the OWL 2 specification forbids the two axioms used on the same object property. This can be solved by expressing them in SWRL, although SWRL rules are only applied on individuals rather than classes.

5.4.2 Maturity of Reasoners

Both HermiT and Pellet can process all the ontologies generated from the Atlantic Zoo without generating any runtime error or getting into an infinite loop. Furthermore, both reasoners always produce the same report for all the metamodels. This is a significant observation because they have been implemented independently by two different development groups. Based on this, we claim that the current implementation of both HermiT and Pellet are mature enough for the task of validating metamodels.

During our experiment, we found that Pellet 2.0.1 supports OWL 1.1 functional syntax [24] which is different from OWL 2 in terms of prefix declaration and datatype maps. HermiT supports strict OWL 2 functional syntax. The ontologies to be processed by HermiT and Pellet are slightly different syntactically, accordingly, the MOFScript transformation for each syntax should be different too.

5.4.3 Performance

We ran our performance experiments using a desktop computer with an Intel Core 2 6400 processor running at 2.13GHz, 2GB of RAM, Linux Fedora Core 10 and Java 1.6.0_10_rc2. We processed each metamodel three times and reported the average execution time.

The time taken to translate the metamodels grows exponentially with its size. 83% of the metamodels took under 10 seconds, but large metamodels can take a few minutes, with the largest taking 4.5 minutes to translate. Generally, the increase in time follows an exponential curve, however, the graph shown in Figure 5.5 is a linear graph, because we validate class diagrams/metamodels without instances. The time taken for the reasoners is generally below 10 seconds; 286 of the 303 ontologies were processed by HermiT in under 10 seconds compared to 282 for Pellet. The results are shown in Figure 5.6. However, it should be noted that there are several outliers. HermiT, while generally slightly faster than Pellet has five outliers that take more than 300 seconds to process, while Pellet only has one. The longest processing time for HermiT was 131 minutes, while the longest for pellet was 56 minutes.

Figure 5.6 also shows the comparison of the times taken to process each metamodel by both reasoners. The x-axis represents the number of axioms in a generated ontology. The y-axis represents the total time in seconds necessary to load an ontology representing a metamodel, and check it for consistency and satisfiability. As for the counting of number of axioms, though, certain expressions are a combination of OWL 2 axioms. For instance, the minimum cardinality in Figure 5.1 is expressed as follows,

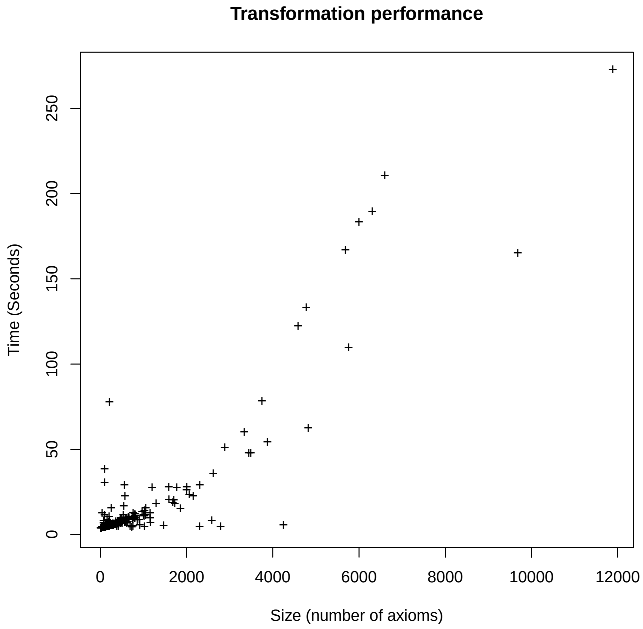

Figure 5.5: MOFScript transformation performance.

which is a combination of axiom `SubClassOf` and `ObjectMinCardinality`, we count this as one axiom.

```
SubClassOf(State ObjectMinCardinality(2 State_outgoing ))
```

As we can observe from Figure 5.6, HermiT is consistently faster than Pellet, but both tools can process each metamodel in less than 10 seconds.

Based on this, we consider that the efficiency of the two reasoners is satisfactory for the given problem and an average desktop computer.

5.4.4 Complexity

We used OWL 2 DL for the representation of UML concepts wherever it is possible, and used a decidable fragment of SWRL only for the representation of a model composition constraint. OWL 2 DL is the standardized formalism of DL which is equivalent to $\mathcal{SHOIN}(\mathcal{D}^+)$. The complexity of OWL 2 DL with regard to the reasoning problems of ontology consistency and instance checking is NEXPTIME complete [50], whereas the complexity of SWRL is undecidable [49, 106]. The combination of OWL 2 DL and SWRL becomes undecidable [49]. However, if all atoms that exist in the SWRL rule use OWL 2 class and property names are restricted to known

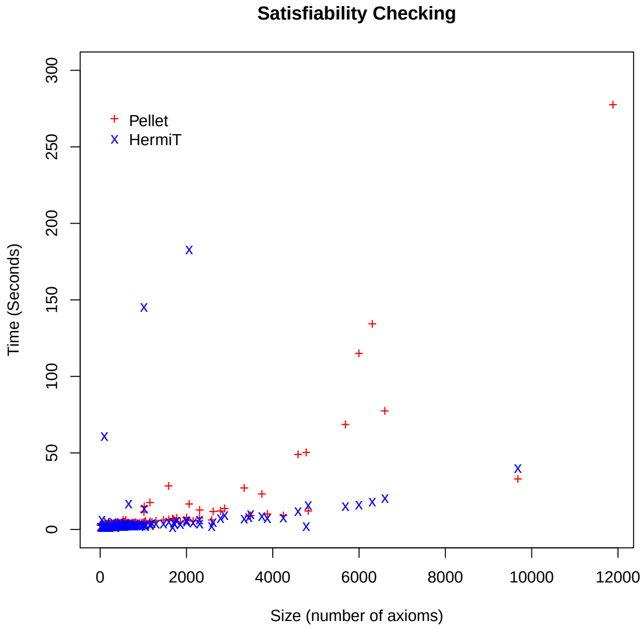

Figure 5.6: Reasoner satisfiability checking performance.

individuals, then the SWRL rule is considered DL-Safe rule and becomes decidable [106, 44]. The data complexity of query answering in the decidable fragment of SWRL is deterministic exponential time [106].

The ontology generated by the translation tool is written by using both OWL 2 DL and the decidable fragment of SWRL. In the light of above discussion, we conclude that the ontology generated by the translation tool is decidable, and the complexity with regard to the reasoning problems such as ontology consistency and instance checking is deterministic NEXPTIME complete.

5.4.5 Problem Reporting

Pellet produces an error message if a given input contains a syntax error or the wrong file header. Such input cannot be further checked for consistency and unsatisfiability. When checking consistency, Pellet simply shows if the input ontology is consistent or not. When checking unsatisfiability, given a consistent ontology, Pellet reports the number of elements checked, time used and number of unsatisfiable elements. If there are unsatisfiable elements, the specific class names are shown.

Pellet provides an option to print verbose information while reasoning. The printed information indicates input size, specific number of classes, properties and individuals, expressivity, used time summary, among others. Once an element that leads to an inconsistency is found, Pellet stops further processing and points out a possible reason, but the actual reason still needs be verified by users.

HermiT also provides verbose information printing, such as the file under processing, and timing for parsing, among others. In contrast to Pellet, when an input is inconsistent, HermiT shows no possible reasons. As far as processing procedure is concerned, HermiT differs from Pellet in that it allows checking unsatisfiability of an inconsistent metamodel.

Regardless of how the two reasoners report the results, the information they give can be difficult to interpret when a metamodel is inconsistent or has unsatisfiable concepts. In the best case, they show the name of the unsatisfiable concepts but no further explanations. More information on what makes a concept unsatisfiable would be extremely helpful.

5.5 Conclusion

In this chapter, we proposed an approach to validate UML and MOF-like metamodels by providing a mapping from UML-like metamodel concepts to OWL 2, to allow automatic validation of metamodels using reasoners. We have used our approach to validate 303 published metamodels. Out of the metamodels in this public repository, 49% are not well-formed. This shows that the problem of metamodel validation is indeed an issue of practical concern. It also demonstrates the value of our approach.

Chapter 6

Consistency of Class Diagrams and Statechart Diagrams

In this chapter we discuss how to check the consistency of class diagrams and statechart diagrams with state invariants. This chapter is based on the work presented in Articles I and V.

6.1 Introduction

UML is a widely used modeling notation for documenting the design of software intensive systems [77]. A UML model usually comprises a number of diagrams providing different views of a system. These diagrams allow us to decompose the design of a large system into smaller and more manageable views. However, representing a system as a collection of diagrams raises the issue of possible design inconsistencies. In this chapter we address the problem of the consistency of UML class and statechart diagrams with state invariants.

A class diagram describes the structure of a system in the form of classes, their associations with each other, the attributes of each class and operations that can be invoked on them. On the other hand, a statechart diagram provides the behavioral interface of a class. It defines all possible sequences of method invocations, the conditions under which methods can be invoked and their expected results.

Each state in a statechart diagram represents a certain condition that is true when the state is active. The condition can be implicit in the design, or defined explicitly in the form of a state invariant. A state invariant is a boolean expression that is true when a given state is active and false otherwise. State invariants are defined using the attributes and associations described in the class diagram and expressed using the Object Constraint Language (OCL) [74].

Given a number of UML class and statechart diagrams, it is possible to specify unsatisfiable state invariants which describe states that can never be active or operations that cannot be implemented according to the well-formedness rules specified in the UML superstructure specification [77]. An unsatisfiable state invariant is considered inconsistent with respect to a class diagram since there are no object instances that can make an unsatisfiable invariant evaluate to true.

The inconsistent state invariants are design errors and, in order to reduce development cost and time, they must be detected and corrected as early in the software development process as possible. The approach we propose to detect such inconsistencies is based on the use of the automatic reasoning tools developed initially in the context of the semantic web. We first translate the class and statechart diagrams with state invariants in a UML model to the Web Ontology Language version 2 for Description Logic (OWL 2 DL) [20], and then use an OWL 2 DL reasoning tool [90, 83] to determine the consistency of the UML design.

The approach presented here is limited to a fragment of the OCL language, but on the other hand is decidable and fully automatic. The designer does not need to know the details of the translation or the reasoning performed by the underlying tools. In addition, the current reasoning tools and desktop computers can process relatively large UML models in few seconds. Therefore we consider that this approach has the potential to be integrated with existing and future UML tools and provides consistency analysis service that goes beyond what is being offered in the current tools which only use basic syntactic analysis and well-formed rules.

6.2 Consistency of Class Diagrams and Statechart Diagrams

In this section we present an overview of our approach that we demonstrate with a running example. Our example system is a Content Management System (CMS). In this system, authors post new articles to be published after being reviewed by a reviewer. A reviewer can accept, reject or advise a revision of the article. Only an accepted article can be published. An article can be withdrawn if it is under review. However, a published article cannot be withdrawn. The structure of this system is described as a UML class diagram (Figure 6.1), while its behavior is described using a UML statechart diagram (Figure 6.2).

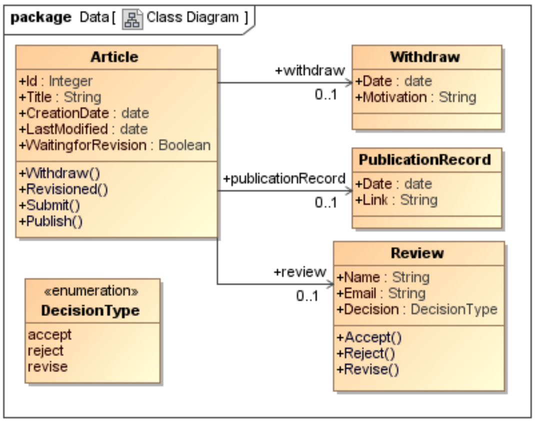

Figure 6.1: The static view of Content Management System.

6.2.1 Class Diagram Representing Structure of CMS

The class diagram provides the main classes involved in the system under development and their associations with each other. It exposes the attributes of each class and operations that can be invoked on them.

The class diagram shown in Figure 6.1 shows the syntactic view of CMS. It consists of 5 classes, namely, *Article, Review, Withdraw, PublicationRecord* and an enumeration class *DecisionType.* An article is associated to *Review, Withdraw* and *PublicationRecord* classes via associations *review, withdraw* and *publicationRecord*, respectively. *Review* class is further associated to an enumeration class *DecisionType* with literals accept, reject and revise. An instance of the article class can be submitted, withdrawn, published or revisioned via *Submit, Withdraw, Publish* and *Revisioned* operations, respectively. An *accept, reject* and *revise* operation can be called on an instance of *Review* class.

6.2.2 Statechart Diagram Representing Behavior of CMS

A statechart diagram defines behavior of a class in terms of states that an instance of a class takes during its lifecycle and the transitions between them. Each transition from a source to a target state is triggered by a function call.

The statechart diagram shown in Figure 6.2 defines the behavioral view of the class *Article* of the class diagram shown in Figure 6.1 in terms of states. It consists of one composite state *ArticleReview* and two simple states *Publish* and *ArticleWithdraw.* The *ArticleReview* composite state consists of four simple states namely, *WaitingforReview*, *Revisioning*, *ArticleRejected* and *Accept.* When the *submit*() method is called on an object

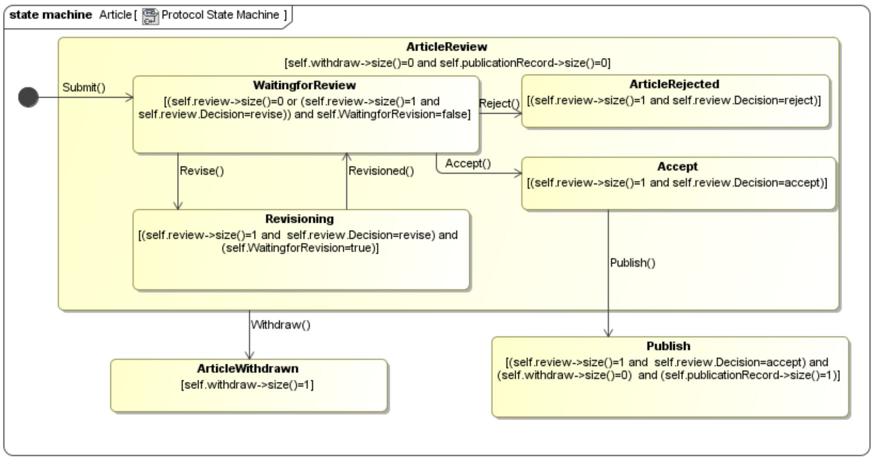
state machine Article [Protocol State Machine]
ArticleReview
[self.withdraw->size()=0 and self.publicationRecord->size()=0]
Submit()
WaitingforReview
[(self.review->size()=0 or (self.review->size()=1 and self.review.Decision=revise)) and self.WaitingforRevision=false]
Reject()
ArticleRejected
[(self.review->size()=1 and self.review.Decision=reject)]
Revise()
Revisioned()
Accept()
Accept
[(self.review->size()=1 and self.review.Decision=accept)]
Revisioning
[(self.review->size()=1 and self.review.Decision=revise) and (self.WaitingforRevision=true)]
Publish()
Withdraw()
ArticleWithdrawn
[self.withdraw->size()=1]
Publish
[(self.review->size()=1 and self.review.Decision=accept) and (self.withdraw->size()=0) and (self.publicationRecord->size()=1)]

Figure 6.2: The behavioral view of the class *Article* of the class diagram shown in Figure 6.1.

of the class *Article*, the statechart diagram is initiated and the object enters into the state *WaitingforReview*, a substate of *ArticleReview*. The method calls *accept*(), *reject*() and *revise*() take the object to the *Accept*, *ArticleRejected* and *Revisioning* states respectively. When the author of the article is revisioning the article, the object of the class *Article* is in the *Revisioning* state. When the author revises the article, he invokes the *Revisioned*() method of the *Article* class and the object again comes into the *WaitingforReview* state. The *publish*() method can be invoked from the *Accept* state and the object switches to the *Publish* state. An article can be withdrawn by invoking the method *withdraw*() whenever the state *ArticleReview* is active, but the *withdraw*() method cannot be invoked if the object of the class *Article* is in the *Publish* state.

6.2.3 State Invariants

Each state in a statechart is annotated with a state invariant. The state invariant is a boolean expression that links classes of a class diagram to the states of a statechart diagram. We say that an object of a class is in a certain state if the state invariant of that state is true. We express the state invariant of each state by using OCL and annotate the behavioral diagram of our example with state invariants in Figure 6.2. The details about the OCL constructs used in our approach will be discussed in Section 6.6.

6.2.4 Invalid State Invariant

We consider the state invariants which let the statechart diagram behave against the UML superstructure specifications for statechart diagrams [77] as inconsistent state invariants, and they may cause the whole system to become unsatisfiable or inconsistent. The examples of inconsistent state invariants are as follows:

Inconsistent State Invariant Example 1: According to the UML superstructure specification, invariants of non-orthogonal states must be mutually exclusive ([77], p.564), for example in the statechart diagram shown in Figure 6.2, the article cannot be in the state *ArticleRejected* if at the same time this article is in the state *Accept*. If we introduce an error by changing the invariant value of the state *ArticleRejected* to `self.review->size()=1 and self.review.Decision=accept`, this means that an article can be rejected and accepted at the same time. The introduced error allows an object of the class *Article* to belong to two non-orthogonal states i.e. *Accept* and *ArticleRejected*, which is the violation of the UML superstructure specification of the statechart diagram, and as a consequence the invariants of states *ActicleRejected* and *Accept* become inconsistent.

Inconsistent State Invariant Example 2: According to the UML superstructure specification, whenever a state is active, all its superstates are active ([77], p.565). This means that all the invariants of an active state and its superstates directly or transitively are true. For example, in a statechart diagram, see Figure 6.2, if the state *Accept* is active then its superstate *ArticleReview* should be also active. If we introduce an error by adding the condition `self.withdraw->size()=1` in the invariant of the state *Accept*, this means that a withdrawn article can also be accepted. The introduced error causes the contradiction between the invariants of the state *Accept* and its superstate *ArticleReview*, and violates the UML superstructure specification of the statechart diagram. Consequently it makes the invariant of the states *Accept*, *ArticleReview* and *ArticleWithdrawn* inconsistent.

In the next section we discuss how we can carry out the analysis of these kind of models using OWL 2 reasoning tools.

6.3 Consistency Analysis

In this section we define the problem of determining the consistency of UML models containing class and statechart diagrams as follows. Our view of model consistency is inspired by the work of Broy et al. [25]. This work considers the semantics of a UML diagram as their denotation in terms of a so-called system model and defines a set of diagrams as consistent when the intersection of their semantic interpretation is nonempty.

In our work, we assume that there is a nonempty set Δ called the object domain containing all the possible objects in our domain. We propose that a UML model depicting a number of class and statechart diagrams is interpreted as a number of subsets of Δ representing each class and each state in the model and as a number of conditions that need to be satisfied by these sets.

A UML class is represented by a set C, such that $C \subseteq \Delta$. An object o belongs to a UML class C iff $o \in C$. We also represent each state S in a statechart as a subset of our domain $S \subseteq \Delta$. In this interpretation, the state set S represents all the objects in the domain that have such state active, i.e., object o is in UML state S iff $o \in S$.

Other elements that can appear in a UML model such as generalization of classes, association of classes, state hierarchy and state invariants are interpreted as additional conditions over the sets representing classes and states. For example class specialization is interpreted as a condition stating that the set representing a subclass is a subset of the set representing its superclass. These conditions are described in detail in the next section.

In this interpretation, the problem of a UML model consistency is then reduced to the problem of satisfiability of the conjunction of all the conditions derived from the model. If such conditions cannot be satisfied, then a UML model will describe one or more UML classes that cannot be instantiated into objects or objects that cannot ever enter a UML state in a statechart. This can be considered a design error, except in the rare occasion that a designer is purposely describing a system that cannot be realized. To analyze the UML models and discover possible inconsistencies we will use the services of an OWL 2 reasoning tool, as described in the rest of this section.

6.3.1 Reasoning

In order to determine the satisfiability of the concepts represented in a UML model, we propose to represent the UML model using a Description Logic, and analyze the satisfiability of the concepts using automated reasoning tools. We have chosen OWL 2 DL to represent our UML models since we consider it well supported and adapted, and there exist several OWL 2 reasoners [90, 83] for analyzing concept satisfiability. A number of UML class diagrams, statechart diagrams and state invariants are taken as an input. All the inputs are translated to OWL 2 DL, and then analyzed by a reasoner. The reasoner provides a report of unsatisfiable and satisfiable concepts. Unsatisfiable concepts will reveal UML classes that cannot be instantiated or UML states that cannot be entered.

In the next section, we discuss and translate the structure of UML models with state invariants, and the UML superstructure specification conditions over the sets representing classes and states into OWL 2 DL.

6.4 From Statechart Diagrams to OWL 2 DL

A statechart diagram provides the behavioral interface of a class and defines the sequence of method invocations, the conditions under which they can be invoked and their expected results. In order to analyze the satisfiability of state invariants in a statechart diagram, we need to translate the states and their invariants into OWL 2 DL. The translation of the state and the state invariant includes the reference of the class and its attributes. Therefore, we translate a statechart diagram in the same ontology that contains the OWL 2 translation of a class diagram.

6.4.1 State and State Hierarchy

We represent a UML state as a concept representing the objects that have such state active. A concept representing a state will be included in the concept representing all object instances of the class associated to the statechart diagram, since all objects that can have the state active belong to the given class. That is, if the state S belongs to a statechart diagram describing the behavior of the class C, then:

$$S \sqsubseteq C$$

We represent this in OWL 2 as follows:

```
Declaration(Class(S))
SubClassOf( S C )
```

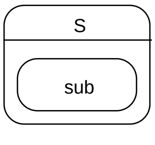

Figure 6.3: State and State Hierarchy

State hierarchy is also represented using the concept inclusion. Whenever a substate is active, its containing state is also active. This implies that the concept representing a substate will be included in the concept representing its parent state, such as:

$$sub \sqsubseteq S$$

This is represented in OWL 2 as:

```
SubClassOf( sub S )
```

6.4.2 Non-Orthogonal States are Exclusive

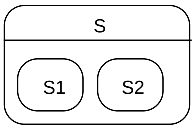

Figure 6.4: Non-Orthogonal States are Exclusive

The UML Superstructure specification requires that if a composite state is active and not orthogonal, at most one of its substates is active ([77], p.564). This means that an object cannot be at the same time in the two concepts representing two exclusive states, i.e., if S_1 and S_2 represent substates of an active and not orthogonal composite state then:

$$S_1 \sqcap S_2 = \perp$$

When representing a statechart diagram in OWL 2, the non-orthogonal exclusive states are declared as disjoint, so that they may not able to share any object.

```
DisjointClasses( S1..Sn )
```

6.4.3 Orthogonal States are Non-Exclusive

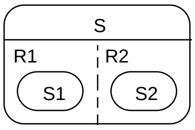

Figure 6.5: Orthogonal States are Non-Exclusive

The UML Superstructure specification requires that if a composite state is active and orthogonal, all of its regions are active ([77], p.564). That is if R_1 and R_2 are concepts representing the two regions of an orthogonal composite state represented by the concept S, then:

$$R_1 \sqcup R_2 = S$$

We should note that if S_1 and S_2 represent two substates where,

$$S_1 \sqsubseteq R_1$$
$$S_2 \sqsubseteq R_2$$

then they are not exclusive and

$$S_1 \sqcap S_2 \neq \perp$$

Due to the open-world assumption of DL, concepts may represent common individuals unless they are explicitly declared as disjoint.

6.5 State invariant into OWL 2 DL

The UML specification defines a state in a UML diagram as the representation of a specific condition "A state models a situation during which some (usually implicit) invariant condition holds" ([77], p.559-560). We understand from this definition that the invariant condition characterizes the state: if the invariant condition holds the state is active, otherwise if the invariant condition does not hold the state is not active. In our approach we represent an invariant as an OWL 2 concept representing objects that make that invariant evaluate to true. Since the invariant holds iff the associated state is active, the concept representing a state will be the same as the concept representing an invariant. This is represented in OWL 2 as an equivalent class relation between the state and its invariant:

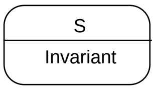

Figure 6.6: OCL State invariant

```
EquivalentClasses (S Invariant)
```

Due to the equivalent relationship between the state and its invariant, all objects that fulfill the condition of its state invariant will also be in that specific state.

6.5.1 State Constraints

The UML also allows us to define additional constraints to a state, and names these constraints also as state invariants. However, the semantics of a state constraint are more relaxed since it "specifies conditions that are always true when this state is the current state" ([77], p.562). In this sense, the state constraints define necessary conditions for a state to be active, but not sufficient. This means that, the actual state invariant may remain implicit. However, we consider a state invariant as a predicate characterizing a state. That is, a state will be active if and only if its state invariant holds.

6.5.2 A State Invariant Characterizes a State

The UML superstructure specification requires that whenever a state is active its state invariant evaluates to true ([77], p.562). A consequence of this is that state invariants should be satisfiable. That is, every state invariant in a statechart diagram must hold in at least one object configuration. Otherwise there cannot be objects that have such state active. Since invariants should be satisfiable, the concept S representing a state should be satisfiable, i.e.

$$S \neq \perp$$

6.6 OCL to OWL 2 DL

A state invariant is a runtime constraint on a state in a statechart ([77], p.514). The UML specification proposes the use of OCL to define constraints in UML models, including state invariants. OCL is well supported by many modeling tools [2, 38]. Unfortunately, in general OCL is not decidable. However, we can avoid undecidability by restricting our approach to a reduced fragment of the full OCL [81]. The use of a limited fragment of OCL to avoid undecidability has been proposed in the past also by other authors [81, 82].

In this thesis, we consider OCL constructs using mainly multiplicity, attributes value and boolean operators. The detail translation of OCL to OWL 2 has been already discussed in Section 4.4. The only difference is the context of OCL constraint. In case of class diagrams the context of an OCL constraint is a *Class*, whereas in case of statecharts the context of OCL constraint is a *State*, for example:

In case of *AttributeConstraints*: The restriction on the value of the attribute is translated in OWL 2 by using the axiom `DataHasValue`. The

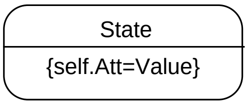

Figure 6.7: OCL State invariant - Attribute Constraint

OCL attribute value constraint `self.Att=Value` is translated in OWL 2 as:

```
EquivalentClasses (State DataHasValue(Att "Value"^^datatype ))
```

In the above translation, *State* is the context of the OCL constraint, *Att* is the name of the attribute, *Value* is the value of the attribute and it is always written in OWL 2 in double quotes, and *datatype* is the datatype of the attribute *Value*.

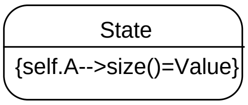

Figure 6.8: OCL State invariant - Multiplicity Constraints

In case of *MultiplicityConstraints*: The OCL constraint `self.A->size()=Value`, in which *A* is the name of an association, ”>” is an infix operator and *Value* is a positive integer, is translated in OWL 2 as:

```
EquivalentClasses (State ObjectExactCardinality(Value A))
```

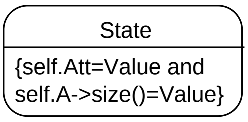

Figure 6.9: OCL State invariant - Boolean Operators

In case of *BooleanOperators*: The OCL constraint `self.Att=Value and self.A->size()=Value` is translated in OWL 2 as:

```
EquivalentClasses (State
ObjectIntersectionOf(
DataHasValue(Att "Value"^^datatype )
ObjectExactCardinality(Value A) ))
```

6.7 Consistency Analysis using an OWL 2 Reasoning Tool

We have defined earlier the satisfiability of UML models in Section 7.3. The consistency analysis of UML models is reduced to the satisfiability of the conjunction of all conditions derived from a model. In order to determine the satisfiability of the conditions represented in UML models, we first translate the UML models into an OWL 2 ontology, then use an OWL 2 reasoner to analyze the satisfiability of translated concepts.

```
// Class Diagram into OWL 2 DL
Declaration(Class(Article))
Declaration(Class(Review))
Declaration(Class(Withdraw))
Declaration(Class(PublicationRecord))
...
DisjointClasses( Article Review ...)
Declaration(ObjectProperty(review))
ObjectPropertyDomain( review Article)
ObjectPropertyRange( review Review )
...
SubClassOf( Article
ObjectMaxCardinality( 1 review ))
....
Declaration(
DataProperty( WaitingforRevision))
SubClassOf(Article
DataExactCardinality(1
        WaitingforRevision))
...
DataPropertyDomain(
WaitingforRevision Article )
..
DataPropertyRange(
WaitingforRevision xsd:boolean )
```

```
//Statechart diagram into OWL 2 DL

Declaration(Class(ArticleReview))
Declaration(Class(ArticleWithdraw))
Declaration(Class(Publish))
SubClassOf( ArticleReview Article )
SubClassOf( ArticleWithdraw Article )
SubClassOf( Publish Article ))
...
DisjointClasses( ArticleReview
ArticleWithdraw Publish )
Declaration(Class(WaitingforReview))
SubClassOf( WaitingforReview ArticleReview )
...
//Invariant of state Publish Start
EquivalentClasses (Publish
ObjectIntersectionOf(
ObjectIntersectionOf(
ObjectExactCardinality(1 review)
DataHasValue(Decision "accept"^^xsd:string ))
ObjectIntersectionOf ( ObjectExactCardinality
(0 withdraw) ObjectExactCardinality
(1 publicationRecord)) ) )
//Invariant of state Publish End
```

Figure 6.10: Excerpt of the output ontology generated by the translation tool.

To translate UML models into OWL 2 ontology, we have implemented the translations of class diagrams, statechart diagrams and state invariants discussed in previous sections, in an automatic model to text translation tool. The implemented translation tool allows us to automatically translate class diagrams, statechart diagrams and state invariants into OWL 2 DL.

```
Found 4 unsatisfiable concept(s):
a:Accept
a:ArticleRejected
a:ArticleReview
a:ArticleWithdrawn
```

Figure 6.11: The satisfiability report of the ontology shown in Figure 8.8 generated by the OWL 2 reasoner Pellet.

The translator reads class diagrams, statechart diagrams and OCL state invariants from an input model serialized using the XMI format. The XMI is generated by using a modeling tool. We used Magicdraw to create the example designs used in this chapter. The output of the translation tool is an ontology file ready to be processed by an OWL 2 reasoner.

As an example, we have translated the class diagram, statechart diagram and OCL state invariants shown in Figure 6.1 and Figure 6.2, into OWL 2 DL ontology using the implemented translation tool. An excerpt of the output ontology generated by the translation tool is shown in Figure 8.8.

6.7.1 Reasoning

After translating the class diagram, statechart diagram and state invariants into an OWL 2 ontology by using the implemented translation tool, we process the ontology by using an OWL 2 reasoner. The OWL 2 reasoner combines all the facts presented as axioms in the ontology and infers logical consequences from them. When we give the generated ontology to the reasoner, it generates a satisfiability report indicating which concepts are satisfiable and which not. If the ontology has one or more unsatisfiable concept, this means that the instance of any unsatisfiable concept will make the whole ontology inconsistent. Consequently, an instance of the class describing an unsatisfiable concept in a class diagram will not exist, or objects will not enter into a state describing an unsatisfiable condition, otherwise vice versa.

In order to analyze the satisfiability of the inconsistent invariants listed in Section 6.2.4, the ontology of an example model with inconsistent invariants is validated by using an OWL 2 reasoner name Pellet [90]. The satisfiability report of the ontology of UML models with inconsistent state invariants is shown in Figure 6.11. As explained in Section 6.5, a state invariant characterizes the state ([77], p.559-560). Therefore, the presence of unsatisfiable states in the satisfiability report indicates the existence of inconsistent state invariants in identified states.

6.7.2 Performance Analysis

In order to determine the performance of the translation and reasoning tools, we conducted an experiment using UML class and statechart diagrams consisting of 10 to 2000 model elements. We use a desktop computer with an Intel Core 2 Duo E8500 processor running at 3.16GHz with 2GB of RAM. The performance tests are conducted for both consistent and mutated models containing inconsistencies introduced by us. For each test, we measure the time required to translate a model from UML to OWL 2 and the time required by the OWL 2 reasoners Pellet [90] and HermiT [83] to analyze the models. The results are shown in Table 6.1, and in Figure 6.12.

Table 6.1: Time taken by the translation tool and reasoning engines to process UML models.

Model Elements	10	100	500	1000	1500	2000
Translation Time	0.08s	0.11s	0.19s	0.30s	0.44s	0.53s
Pellet						
Valid	2.2s	2.3s	2.6s	3.2s	3.6s	3.8s
Mutated	2.2s	2.4s	2.7s	3.2s	3.6s	3.9s
HermiT						
Valid	0.6s	0.7s	1.2s	1.7s	2.2s	2.6s
Mutated	0.7s	0.7s	1.3s	1.8s	2.3s	2.6s

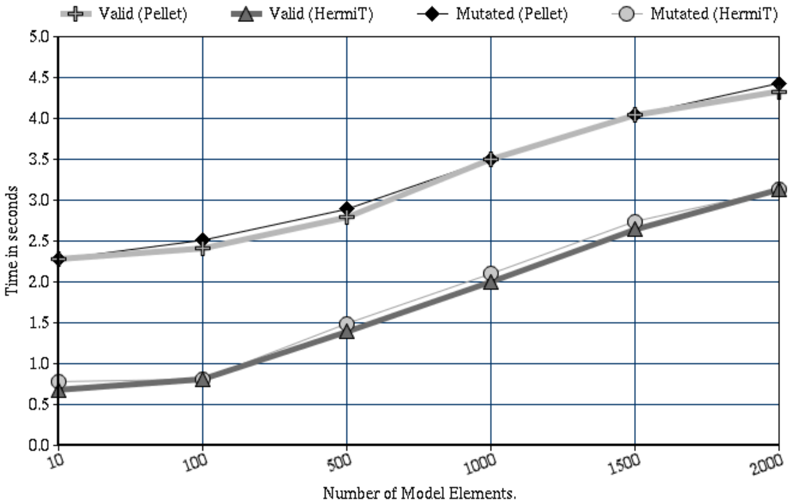

Figure 6.12: The graph of the total time (Translation time + Reasoning time) to process valid and mutated models.

The time complexity of OWL 2 DL with respect to the reasoning problems of the ontology consistency and instance analyzing is NEXPTIME complete [50]. However, the graph (Figure 6.12) of the performance test shows that the time required to reason about models only grows linearly. This is due to the fact that in our approach we analyze the consistency of class and statechart diagrams without individuals.

6.8 Conclusion

In this chapter we have presented an approach to analyze the consistency of UML class diagrams and UML statechart diagrams with state invariants. The approach is fully automated thanks to the translation tool and the existing OWL 2 reasoners. Since the translation tool accepts standard UML models serialized using the XMI standard, the approach can be easily integrated with existing UML modeling tools.

Our approach is decidable because we restrict ourselves to an admittedly small fragment of OCL. This strategy has been already used for expressing constraints over class diagrams [26, 82]. We believe that the use of limited subsets of OCL does not reduce the merits of this and similar approaches even if they cannot be used to process all possible OCL constraints. An analysis tool could in fact integrate different analysis approaches and use the right one depending on the fragment of OCL used in the models.

The performance experiments show that the proposed approach can process relatively large UML models in few seconds by using current reasoning tools on desktop computers. Therefore, we consider that this approach has the potential to be incorporated with existing and future UML modeling tools and offer consistency analysis services that go ahead of what is being offered in current modeling tools.

Chapter 7

Application: Design of Behavioral REST Web Service Interfaces

In this chapter we present an application of the work discussed in Chapter 6 by using a scenario of a REST web service interface, and it is based on the work presented in Articles V and VI.

7.1 Introduction

The design phase of a software development lifecycle is crucial in the development of a reliable software, since the design models developed in this phase are carried forward to all the other phases. It is therefore important that these design models are constructed correctly. The design models are created from different viewpoints to capture different features of the system under development, since all the features are difficult to capture in a single model and can make a single model complex. Capturing the system under development in different models from different viewpoints gives a better and simpler understanding of the system, but raises the issue of models inconsistency. Models can become inconsistent, if they define the same system but have contradicting specifications in different models or have specifications that cannot be satisfied and resulting in undesirable results in its implementation. This raises the need for consistency analysis of design models. The need for consistency analysis of models can rise even if the models themselves have no errors. Designers may specify certain requirements in different models that contradict each other and thus, cannot exist together leading to inconsistent diagrams. These mistakes can lead to implementations that do not provide correct functionality as expected from them.

As the software shifts from software as a product to software as a service, the need for consistency analysis of design models rises even further since the service users have no control over the software and completely rely on advertised service specifications. Also, since web services are offered from remote locations that consumers use via Internet using standard internet protocols, they can be expensive in terms of bandwidth and other costs. Thus, it is important to deliver services that are reliable and do not contain unintended design mistakes to avoid undesired results.

In this chapter, we discuss a consistency checking approach for the design models of behavioral REST web service interfaces. REST [36] is an architectural style to design scalable web services which play well with the existing infrastructure of web. They usually offer simple interfaces that can create, retrieve, update and delete information from a database. However, it is possible to create REST web service interfaces that do more than simple CRUD operations. Such beyond CRUD REST interfaces offer different service states that need to be preserved as the consumer goes through trails of resources.

In [80], we present a design approach that creates behavioral REST web service interfaces by construction. Designing and publishing a REST web service interface with stateful behavior may involve many resources and different service states that are dependent on these resources. This can result in inconsistencies leading to service implementations with unintended behavior. In this chapter we present a consistency checking approach that analyzes the REST design models to detect inconsistent behavior and as such advises the developer to correct the detected design mistakes and create consistent behavioral REST web service interfaces. A behavioral interface is said to be consistent if it does not contain any contradicting specifications and there exists a service that can satisfy it.

7.2 REST Designs and their Inconsistencies

REST is a resource-centric architecture and a REST interface exposes resources that can be manipulated using standard HTTP methods. It offers features of *connectivity*, *addressability*, *statelessness* and *uniform interface.* Connectivity requires that there is no isolated resource and every resource is reachable. Addressability feature requires that every resource can be reached independently with a URI (path). The statelessness feature requires that no hidden session or state information is passed in the method calls and the uniform interface feature requires that the same set of methods (standard HTTP methods) is used to manipulate all resources. A behavioral REST interface should exhibit all these features of a REST interface and also provide information on how to use a service, e.g. the sequence of method invocations and the effects of service requests on that service.

7.2.1 Modeling REST Behavioral Interfaces

We represent a behavioral REST interface with resource and behavioral models using UML class and protocol state machine diagrams from UML [77], respectively, with some additional constraints to make them RESTful. For example, Figure 7.1 and Figure 7.2 shown in Section 4 depicts a resource model and a behavioral model for a hotel booking REST web service interface that takes payment from the customer and books a room in the hotel. The web service reserves a room for the customer and uses a third party payment service for confirmation. The service can be canceled when it is not processing payment and can be deleted only if it is canceled.The example is simple to understand and helps in demonstrating complex service states.

A resource is a piece of information that is exposed via a URI and that can be manipulated with standard HTTP methods. A resource can be either a collection resource or a normal resource. Collection resource does not have any attributes of its own and contains a list of other resources, whereas, a normal resource has its own attributes and represents a piece of information. In our resource model, we represent *resource definitions* as classes, such that instance of these *resource definitions* are termed as resources, analogous to the relationship between *class* and its *objects* in object oriented paradigm. A collection resource definition is represented by a class with no attributes and a normal resource definition has one or more attributes. Each association has a name and minimum and maximum cardinalities. These cardinalities define the minimum and maximum number of resources that can be a part of the association. We also define a root resource definition in the resource model that represents the service. Root resource definition is connected to every other resource definition in the behavioral model. In Figure 7.1, *Booking* is the root resource definition.

The behavioral specifications of a REST interface are represented as a behavioral model that represents different states of a service during its lifecycle, the methods that can be invoked on these resources and a sequence of these methods invocations. A state in a behavioral model represents the resource configuration of the service at a particular instance of time. A transition (from source state to target state) with a trigger method indicates the change of service state when an HTTP method with a side effect is invoked. The only allowed methods in our behavioral model are HTTP GET, PUT, POST and DELETE methods leading to a uniform interface. Of these methods, GET is idempotent and does not change the state of a service whereas PUT, POST and DELETE have side-effects and can change the state of a service.

7.2.2 Inconsistency Problems

Each service state has a state invariant. We define invariants of states as predicates over resources defined in the resource model having either a true or a false value. For a state to be active, its state invariant should be true, otherwise it should be false. When the client makes a service request, it is mapped to a transition in the behavioral model that has that method as a trigger. The transition is fired from a source state to a target state. If the state invariant of the source state is inconsistent, a service can never exist in this state and it would be impossible for the implementation of the interface to decide which transition to take as a result of a service request. For example in Figure 7.2, the state invariant of state *processingPayment* is $self.payment- > size() = 1$ *and* $payment.waiting = True$. If we change its invariant to have $payment.waiting = False$, then it could conflict with the state invariant of *unpaidBooking*, i.e. both the states can be true at the same time. In this case, if PUT is invoked on *payment* resource, the implementation would not know which transition to take. Such inconsistency problems can lead to service implementations with undesirable behavior.

7.3 Consistency Analysis

In this section we define the problem of determining the consistency of our REST web service design models. In our work, we assume that there is a nonempty set $\Delta^{\mathcal{I}}$ called the domain containing all the possible resources and resource configurations in our domain. We propose that a design model depicting a number of resource and behavioral diagrams is interpreted as a number of subsets of $\Delta^{\mathcal{I}}$ representing each resource definition and each state in the model and as a number of conditions that need to be satisfied by these sets.

A resource definition is represented by a set R, such that $R \subseteq \Delta^{\mathcal{I}}$. A resource r belongs to a resource definition R iff $r \in R$. We also represent each state S in a statechart as a subset of our domain $S \subseteq \Delta^{\mathcal{I}}$. In this interpretation, the state set S represents all the resources in the domain that have such state active, i.e., resource r is in state S iff $r \in S$.

Since resource and behavioral models are represented with class and state machine diagrams respectively, other elements that can appear in a UML model such as generalization of classes, association of classes, state hierarchy and state invariants are interpreted as additional conditions over the sets representing resources and states. For example specialization is interpreted as a condition stating that the set representing a subresource is a subset of the set representing its superresource. These conditions are described in detail in the next section.

In this interpretation, the problem of design model consistency is then reduced to the problem of satisfiability of the conjunction of all the conditions derived from the model. If such conditions cannot be satisfied, then a design model will describe one or more resource definitions that cannot be instantiated into resources or resources that cannot ever enter a behavioral state in a statechart. This can be considered a design error, except in the rare occasion that a designer is purposely describing a system that cannot be realized.

7.3.1 Reasoning Tool Chain

In order to determine the satisfiability of the concepts represented in our design model, we propose to represent the resource and behavioral models using a Description Logic, and analyze the satisfiability of the concepts using an automated reasoning tool. We have chosen OWL 2 DL to represent our UML models since we consider that it is well supported and adapted, and there exist several OWL 2 reasoners for checking concept satisfiability.

A number of resource models, behavioral models and state invariants are taken as an input. All the inputs are translated to OWL 2 DL, a web ontology language [20]. The OWL 2 translation of design models are passed to a reasoner. The reasoner provides report of unsatisfiable and satisfiable concepts. Unsatisfiable concepts will reveal resource definitions that cannot be instantiated or behavioral states that cannot be entered.

We have also implemented tool that generates a) skeleton of REST web services from design models, b) OWL 2 DL from design models. The generation of REST web service skeleton is implemented using python in Django web framework [47]. We translate resource and behavioral models into models.py, urls.py and views.py that are basic files of Django. The skeleton for each resource contains information on its representation and its relative URI, the allowed methods, and the preconditions and postconditions for methods that have side-effects. The developer can inspect the generated code and fill in the desired logic for methods in the generated REST web service skeletons. The tool that generates OWL 2 DL from design models is discussed later in Section 7.6.

In order to translate the design models into OWL 2 ontology, we need to first formally define the structure of our design models.

7.4 Structure of Behavioral RESTful Interfaces

In this section, we formally define the resource and behavioral models of behavioral REST interfaces and define the constraints that make them RESTful.

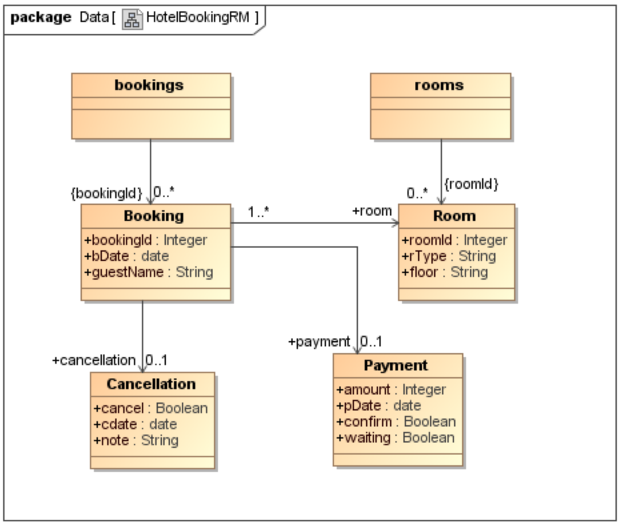

Figure 7.1: Resource Model for RESTful Web Service with Invariants

7.4.1 Structure of Resource Model

The structure of a REST interface is represented with a class diagram from UML [77] with some additional constraints. Figure 7.1 shows the resource model of Hotel Booking REST web service interface. It has six resources with two collection resources: *bookings* and *rooms*, and four normal resources: *Booking*, *Room, Cancellation* and *Payment*. There is no isolated resource definition and every resource definition is connected via association. Each association has a name and multiplicity constraint.

Definition 1 A resource model RM is defined as a tuple:

$$RM = \langle R\_def, root, Att, A, l, issubresource, min, max \rangle$$

where R_def is a set of resource definitions, *root* is the root resource definition and *Att* is a set of attributes. Also:

- A is a set of associations where each association connects two resource definitions. We define association as a relation between two resource definitions, i.e. $a : R\_def \times R\_def$.
- $l(a)$ gives name of the association. l is defined as an injective function from set of associations A to L, the set of labels, i.e. $l : A \to L$. The injective function implies that every association has a unique name.
- $min(a)$ and $max(a)$ give the minimum and maximum cardinalities on association a. They are both defined as a function from a set of

associations to natural numbers, i.e.,$min : A \rightarrow N$ and $max : A \rightarrow N$, such that $min(a) \leq max(a)$.

- $issubresource(r_1, r)$ evaluates to true if r_1 is a subresource of r. Resource model can have resource hierarchy in which subresources of a resource inherit the properties and attributes of its parent resource.

R_def represents *resource definition* that defines a resource in resource model. The instances of these resource definitions are resources. The relation between *resource definition* and its *resources* is the same as the relationship between *class* and its *objects* in object oriented paradigm where class represents all the entities that share the same set of properties and its *object* represents its instance.

The set of resource definitions R_def in RM is the union of a collection resource R_c and a normal resource R_n definitions, i.e. $R\_def = R_c\_def \cup R_n\_def$ and $R_c\_def \cap R_n\_def = \emptyset$

- R_c_def is a set of collection resource definitions. A collection resource does not contain any attribute of its own, i.e. $R_c\_def = \{r_c \in R_c\_def : \forall r_c \in R_C \wedge \forall att \in Att : att \notin r_c\}$, where r_c is an instance of collection resource R_c.
- R_n_def is a set of normal resource definitions. We call a resource normal (not collection) if a resource has at least one attribute, i.e., $R_n\_def = \{r_n \in R_n\_def : \exists att \in Att : att \in r_n\}$

We take a *root* as a resource definition that represents the service. All other resource definitions are linked to *root* and are navigated through it. In Figure 7.1, we take *Booking* resource definition as *Root*. This root resource definition is accessed with its specific booking Id, i.e., the starting navigation path for all the resource definitions in resource model is $/\{bookingId\}/$.

In order to exhibit features of connectivity and addressability, we constrain our resource model with the following design decisions.

Connectivity

All resource definitions should be connected via associations and every resource definition is reachable from the root resource definition such that there is no isolated resource definition or sub-graph. For example, the resource model in Figure 7.1 is connected because there is a path from root resource definition *Booking* to every other resource definition.

Addressability

The addressability feature of REST interface requires that every resource should have a URI address. We can retrieve the relative navigation path to a resource definition from a resource model by concatenating the association names of associations that make a path to the resource definition.

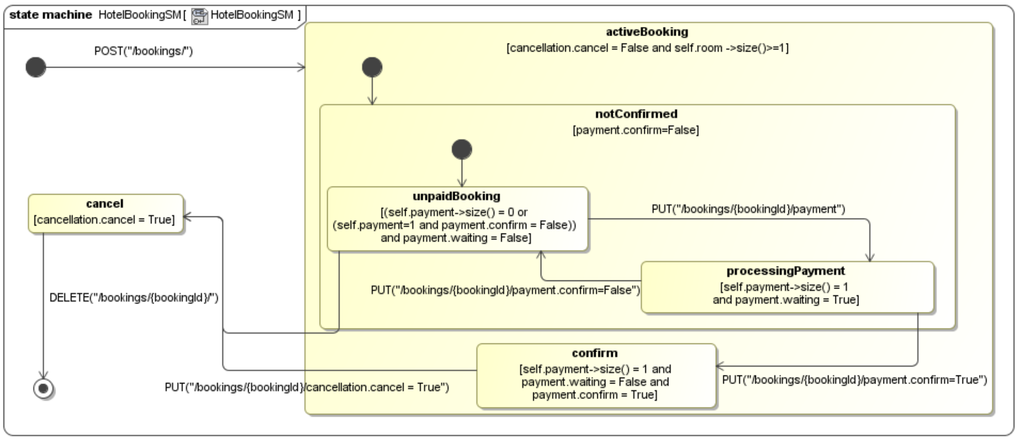
state machine HotelBookingSM [HotelBookingSM]
POST("/bookings/")
activeBooking
[cancellation.cancel = False and self.room ->size()>=1]
notConfirmed
[payment.confirm=False]
unpaidBooking
[(self.payment->size() = 0 or
(self.payment=1 and payment.confirm = False))
and payment.waiting = False]
PUT("/bookings/{bookingId}/payment")
processingPayment
[self.payment->size() = 1
and payment.waiting = True]
PUT("/bookings/{bookingId}/payment.confirm=False")
cancel
[cancellation.cancel = True]
DELETE("/bookings/{bookingId}/")
confirm
[self.payment->size() = 1 and
payment.waiting = False and
payment.confirm = True]
PUT("/bookings/{bookingId}/payment.confirm=True")
PUT("/bookings/{bookingId}/cancellation.cancel = True")

Figure 7.2: Behavioral Model of REST Web Service Interface

For example, the payment resource definition in Figure 7.1 can be reached via $/\{bookingId\}/payment$. We ensure the addressability feature by constraining each association to have a unique association name and direction. This is implied by the injective l function defined above. These association names and directions give the addressable path to the resource definitions and navigation directions.

7.4.2 Structure of Behavioral Model

The behavior of a REST interface is represented as a protocol state machine from UML [77] with some additional constraints. Figure 7.2 shows the behavioral model of Hotel Booking REST web service interface. A *Booking* resource is created when POST is called on *bookings* collection resource. A *Booking* is active and unpaid when it is created. When user of the service invokes a PUT on *payment* resource, the web service goes to *processingPayment* state. If the payment is confirmed to be true, it goes to *confirm* state, otherwise it goes back to the *unpaidBooking* state. The booking cannot be cancelled if it is processing the payment. When DELETE is invoked on *Booking* resource from the state *cancel*, it is deleted from the system.

Definition 2 A behavioral model of REST web service is given as a tuple:

$BM = \langle S, \iota, F, T, \sigma, g, pc, inv, issubstate, trigger, region\rangle$

where S is a set of states, ι is the initial state such that $\iota \in S$, F is the set of final states such that F $\subseteq$ S, T is a set of transitions and σ gives the resource configuration of the web service in a particular state such that $\sigma \in \sum$ where $\sum$ is the set of all possible resource configurations. Also,

- $g(\sigma, t)$ evaluates the guard and $pc(\sigma, t)$ evaluates the postcondition of transition t in state σ . They evaluate to true in case they are not mentioned.
- $inv(s, \sigma)$ evaluates the invariant of state s of a service in state σ. The invariant of state s for a RESTful web service is retrieved by invoking a GET method on all resources that are a part of resource configuration in σ in state s.
- $issubstate(s, s_1)$ evaluates to true if s is a substate of s_1.
- $trigger(t)$ gives the trigger method for transition t and is defined as a function, $trigger : T \rightarrow Trigger$, where $Trigger = \{PUT, POST, DELETE\}$
- $region(s_1, s_2)$ is a predicate that returns true if state s_1 and s_2 belong to the same region.

We make the following main design decisions in the construction of our behavioral interface for REST web service to address features of REST interface.

Uniform Interface

We only allow HTTP methods GET, PUT, POST and DELETE in our behavioral model. The GET method is idempotent and has no side-effects. GET method is used to retrieve the state of resources that constitute state invariants. The only allowed methods that can trigger a state change in our state machine are PUT, POST and DELETE.

Statelessness

When a method is invoked, it makes a transition from one service state to another. The statelessness feature requires that no hidden state or session information should be passed as a part of the method call such that each HTTP request is treated as an independent request. This features leads to scalability of web services. For this feature we require that all the information passed with the method call is either a part of URL or is in request parameters.

Also, in order to define statelessness of REST interface, we define states of a service as predicates over resources. A state is active when the resource configuration defined in its state invariant is true otherwise false. We, thus, define service states of a REST web service without violating the statelessness feature of REST interface.

In the next section, we discuss and translate the structure of a resource and behavioral model with state invariants over the sets representing resource definitions and states into OWL 2 DL.

7.5 From Resource and Behavioral Diagrams to OWL 2 DL

In order to check the satisfiability of resource definitions in a resource model, we need to first translate all resource definitions and their associations into OWL 2 ontology, and then validate the OWL 2 ontology using an OWL 2 reasoner. In this section we only present the translation of those concepts of a resource model which are required for the validation of the behavioral model such as: resource definitions, associations, multiplicity and attributes.

7.5.1 Resource Model in OWL 2

Each resource in a conceptual model is shown as a class in an ontology and an association as an object property. A class in OWL 2 is a set of individuals and *ObjectProperty* connects a pair of individuals[20]. According to the definition of a resource model given in Definition 1, we need to map these concepts in OWL 2 DL: resource definitions and their specializations, attributes, associations and association multiplicities.

Resource Specification and Hierarchy

A resource definition in a resource model represents a collection of resources which share the same features, constraints and definition. For each resource definition R_def in *RM*, we define an OWL 2 axiom:

```
Declaration(Class(R_def))
```

$issubresource(r_1, r)$ is true if r_1 is a subresource of resource r. We explicitly define the hierarchy of resources in OWL 2 between resources. The specialization of resources represented as classes is reduced to the set inclusion. We represent the fact that a resource definition R1 is a specialization of resource definition R2 with the condition $R1 \subseteq R2$. In this case we say that R2 is a super resource of R1, analogous to the superclass in a UML class diagram. If two resource definitions, R1 and R2, have a common super resource, or R2 is the super resource of R1 we say that they are in a specialization relation. The specialization relation $R_1 \subseteq R_2$ is translated in OWL 2 as:

```
SubClassOf( R1 R2 )
```

Each resource definition at the same hierarchical level in a resource model represent a different piece of information. We assume that a resource cannot belong to two resource definitions, except when these two resource definitions are in a specialization relation. In our semantic interpretation of a resource diagram, it is equally important to denote the facts that two resource definitions are not in a specialization relation. We represent the fact that two resources R1 and R2 are not in a specialization relation with the condition $R1 \cap R2 = \emptyset$. With this condition, a resource cannot belong to these two resource definitions simultaneously. Due to the open-world assumption used in Description Logic, we need to explicitly state this fact in OWL 2, i.e. for resource definitions R1...Rn at the same hierarchical level, we define disjointness in OWL 2 as:

```
DisjointClasses( R1..Rn )
```

Attributes

In our resource model, a collection resource does not have any attribute and a normal resource should have at least one attribute. So we define attribute *att* of a normal resource r of type D as $DataProperty(att)$ with domain as r and range as D. We do not need to give any attribute definition for collection resources because OWL 2 has an open world assumption and that which is not mentioned is not considered. So by simply not mentioning collection resources with any attributes is sufficient. However, for every normal resource, each of its attributes is defined in OWL 2. Attributes usually have a multiplicity restriction to one value. Hence, the attribute definition *att* in OWL 2 is given as:

```
Declaration(DataProperty( att ))
SubClassOf(C DataExactCardinality(1 att ))
DataPropertyDomain( att r )
DataPropertyRange( att D )
```

Association

An association is a relation between two resource definitions, r_1 and r_2 and the name of the association is its label l. For each association a in A, we define `ObjectProperty` axiom with label l and r_1 as its domain and r_2 as its range, i.e., for $a(r_1, r_2)$ with $l(a)$, we give OWL 2 definition as:

- l maps to the OWL 2 axiom `Declaration(ObjectProperty(l))`
- r_1 maps to the OWL 2 axiom `ObjectPropertyDomain(l` r_1`)`
- r_2 maps to the OWL 2 axiom `ObjectPropertyRange(l` r_2`)`

$min(a)$ and $max(a)$ give minimum and maximum cardinality of association a. It defines the number of allowed resources that can be part of the association. We represent a directed binary association A from resource definition R1 to R2 as a relation $A : R_1 x R_2$. The multiplicity of the association defines additional conditions over this relation $\#\{y|(x, y) \in A\} \geq min$, $\#\{y|(x, y) \in A\} \leq max$. If an association a has minimum cardinality min and maximum cardinality max for resource r, we define it in OWL 2 as:

```
SubClassOf( r ObjectMinCardinality( min a ) )
SubClassOf( r ObjectMaxCardinality( max a ) )
```

7.5.2 Behavioral Model in OWL 2

A behavioral model provides the behavioral interface of a web service and defines the sequence of method invocations, the conditions under which they can be invoked and their expected results. To check the satisfiability of state invariants in a behavioral model, we need to translate the states and their invariants into OWL 2. The translation of the state and the state invariant includes the reference of resources and their attributes so we translate a behavioral model in the same ontology that contains the OWL 2 translation of a resource model.

We need to cover following concepts of our behavioral model in OWL 2: state, state hierarchy, state disjointness and state invariant. The behaviour model is translated into OWL 2 by following the translation method discussed in the previous chapter.

7.6 Validation of RESTful Interfaces

We have defined earlier the satisfiability of our design models in Section 7.3. The consistency analysis of resource and behavioral models is reduced to the satisfiability of the conjunction of all the conditions derived from the model. In order to determine the satisfiability of the conditions represented in the design models, we first translate the resource and behavioral models into an OWL 2 ontology, then use an OWL 2 reasoner to analyze the satisfiability of translated concepts.

To translate the resource and behavioral models into OWL 2 ontology, we have implemented the translations of resource and behavioral diagrams in OWL 2, discussed in Section 7.5 in the form of a translation tool. We have used Python programming language for the implementation of the prototype of the translation tool. The implemented translation tool allows us to automatically transform a resource and behavioral model into OWL 2 DL. The translator takes these models and OCL state invariant as an input in the form of XMI. The XMI is generated by using a modeling tool, Magicdraw. The XMI generated by the modeling tool contains the source code of both resource and behavioral model in the form of XML. While modeling in a modeling tool we have used OCL to express the state invariants in a behavioral model. The state invariant written in OCL is also a part of a XMI generated by the modeling tool. Moreover, the output of an implemented translation tool is an ontology file, which contain the transformed resource model, behavioral model and state invariants in the from of OWL 2 functional syntax.

After translating the design models and state invariants into OWL 2 ontology by using the implemented translation tool, we will validate the output ontology by using an OWL 2 reasoner. The OWL 2 reasoner analyzes different facts presented as axioms in the ontology and infers logical consequences from them. When we give our ontology to the reasoner, it generates satisfiability report indicating which concepts are satisfiable and which not. If the ontology has one or more unsatisfiable concepts, this means that the instance of any unsatisfiable concept will make the whole ontology inconsistent, consequently, an instance of the resource definition describing an unsatisfiable concept in a resource diagram will not exist, or resources will not enter in a state describing an unsatisfiable condition, and vice versa. However, the ontology generated by the translation tool is in OWL 2 functional syntax, therefore, the satisfiability of the translated concepts can be checked by using any OWL 2 reasoner which supports OWL 2 functional syntax.

7.7 Conclusion

A REST interface can do more than simply creating, retrieving, updating and deleting data from a database. Designing behavioral interface for such web services that provide different states of the service and offer REST interface features is an interesting design challenge since it can involve many resources and resource configurations that define different states of the service. In this chapter, we address how to analyze the consistency of design models that create behavioral REST interfaces. We check the consistency of resource and behavioral diagrams with state invariants using OWL 2 reasoners. The structure of both the diagrams is formally defined and translated to OWL 2 ontology. The ontology is given to an ontology reasoner that checks the ontology for any unsatisfiable concepts. The unsatisfiable concepts indicate the design errors that can cause undesirable behavior in the implementation of the service. The approach is automated as we provide prototype tools that generate web service skeletons and OWL 2 ontology from the design models. Also, thanks to the existing OWL 2 reasoners the generation of satisfiability report for our OWL 2 ontology is also automated that is analyzed to check the consistency of design models.

In our future work, we plan to take into account guards on transitions in the consistency analysis of our design models.

Chapter 8

Consistency of Class and Object Diagrams

In this chapter we discuss the translations of UML object diagram concepts into OWL 2, and also discuss how to validate these concepts using OWL 2 reasoners. This chapter is based on the work presented in Articles I, III and IV.

8.1 Introduction

In MDE, each software model is described using a particular modeling language, such as the UML or DSLs, and this raises the question if each model conforms to its metamodel or not. In our context, conformance means that in a given UML class diagram containing classes and associations, and a UML object diagram containing objects and links, we want to know, first, if each object is a proper instance of a class and each link is an instance of an association depicted in the class diagram. Second, objects and links must preserve the uniqueness, multiplicity, source, target and composition constraints of their classes and associations. An example of a valid and an invalid object diagram and a class diagram is shown in Figure 8.1.

8.1.1 Overview of the Approach

In order to mechanically reason about model conformance we need to have a formal definition of UML. In this chapter we have chosen the Web Ontology Language version 2 for Description Logic (OWL 2 DL) [98] to formalize the UML class and object diagraming concepts. There are a number of reasons behind the selection of OWL 2 DL. Firstly, OWL 2 DL is the subset of OWL 2 that is decidable. Secondly, by using OWL 2 DL we will be able to use existing OWL 2 reasoners for model conformance. Finally,

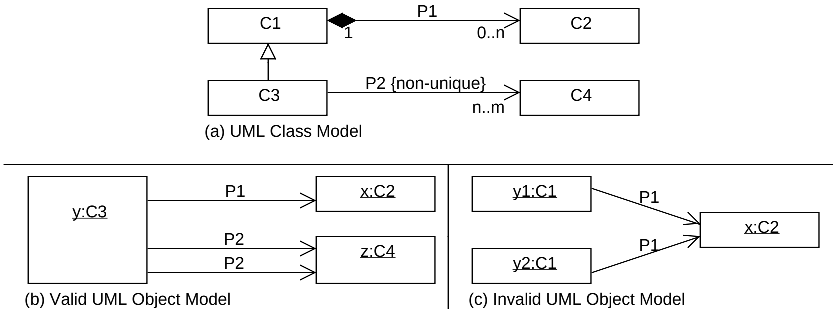

Figure 8.1: (a): A UML class diagram depicting a class hierarchy, a composition and a non-unique association. (b): A consistent UML object diagram conforms to the UML class diagram, and (c): An inconsistent object diagram due to the shared owner.

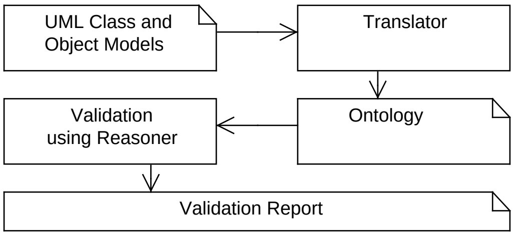

Figure 8.2: Automatic object and class diagram conformance process.

OWL 2 DL already provides constructs to represent many UML concepts such as classes, associations, objects, and links in a straightforward way.

Still, a translation of UML models to OWL 2 presents several challenges. Unfortunately, OWL 2 DL does not provide any constructs to represent UML concepts such as composition, ordering and non-unique associations. Also, OWL 2 uses open-world assumption, whereas UML operates under the closed-world assumption [59], where complete knowledge of the domain is assumed to be provided in a model. We assume that all existing classes and objects are known and depicted explicitly in the models.

The details about representing composition, ordering and non-unique associations and the closed-world environment in OWL 2 DL are the main contributions of this chapter and will be discussed later. To tackle the problem of model conformance, we propose a tool to translate UML models into OWL 2 DL, and then mechanically check the translated models for consistency using an OWL 2 reasoner. An overview of the process is given in Figure 8.2. The translator takes a UML class diagram and an object diagram as an input in the form of XMI [73] and produces ontology in OWL 2 format as an output. The translator contains the translations of UML concepts into OWL 2 axioms in the form of MOFScript [5]. The detail about the translation of a UML class and object diagraming concepts into OWL 2 will be discussed in Section 8.2, and the detail about the translation process will be discussed in Section 8.3.

Furthermore, the ontology generated by the translator contains the translation of the object diagram and the class diagram in the form of OWL 2 DL. The generated ontology will further be validated by using an OWL 2 reasoner.

- If a generated ontology is consistent, it means that the object diagram conforms to all constraints of the class diagram interpreted according to the semantics that we have given in our translation.
- If a generated ontology is inconsistent, it means that the object diagram does not conform to the constraints given in the class diagram.

8.2 UML Object Diagrams

In this section we give a formal definition of the UML class and object diagraming concepts that we treat in our approach. The definition is given in terms of predicate logic. In this section we also give the translation of UML Object diagraming concepts to OWL 2 and motivate our choice of features that are included in the definition.

8.2.1 UML Classes and Objects

A UML class represents a set of objects that have the same characteristics [77]. A UML class C is defined as a unary predicate C in predicate logic.

An instance of a class is called an object. In UML, every object in an object diagram must belong to a specific class in a class diagram. An object x in an object diagram belongs to a class C in a class diagram is defined in predicate logic as:

$$C(x)$$

A UML Class C in the class diagram is translated to OWL 2 as:

```
Declaration( Class( C ) )
```

Furthermore, every object that exists in an object diagram must belong to a specific class in a class diagram. In OWL 2 the UML object is represented as a class assertion and is called an *individual*. A UML object x of the class C is translated in OWL 2 as:

```
ClassAssertion( C x )
```

Furthermore, every object in a UML object diagram is by default different from another. Whereas, in OWL 2 due to the open-world assumption, we need to explicitly mention that all individuals are different from each other. For example: for objects $x1, .., xn$ in an object diagram, we use the OWL 2 axiom:

```
DifferentIndividuals(x1..xn)
```

8.2.2 Class Memberships

The UML and MOF semantics for instantiation follow closely the object-oriented paradigm i.e. the closed world assumptions [59]. In this context, when declaring that a model element m is of type C, it is required that the following holds:

1. m is a direct instance of the class C.
2. m is an indirect instance of all the superclasses of C.
3. m is not an instance of any other class.

However, OWL 2 follows the open world assumption. Therefore, declaring that an element m is of type C in OWL 2 asserts that:

1. m is a direct or indirect instance of the class C.
2. m is an indirect instance of all the superclasses of C.
3. m is not an instance of any classes explicitly declared disjoint to C.

Furthermore, these semantic differences require that we introduce additional axioms in our ontology to restrict object membership to the intended class hierarchies.

8.2.3 Class Memberships Within Inheritance Hierarchies

According to UML semantics of class membership, whenever we assert that a model element is an instance of a class, we also assert that it is not an instance of its subclasses. But in OWL 2 this is not the case. In order to overcome this semantic gap between OWL 2 and UML, we translate UML class into OWL 2 as a union of two disjoint concepts.

```
EquivalentClasses( C ObjectUnionOf( C_Direct C1..Cn ) )
DisjointClasses( C_Direct ObjectUnionOf( C1..Cn ) )
```

First concept *C_Direct*, represents the collection of all the objects which are direct instances of a class. And, the second concept represents all objects that belong to its subclasses $C1, .., Cn$.

8.2.4 UML Association and Links

A UML binary association defines a relationship between two classes [77]. A UML link is an instance of an association. A link l of an association P connecting objects x and y is represented in predicate logic as:

$$P(x, y, l)$$

We often do not need to differentiate what link is used to connect two objects. Therefore it is convenient for us to define the following:

$$\forall x, y, l.P(x, y, l) \rightarrow P(x, y)$$

A UML association is represented in OWL 2 as an object property. An association P from a class C1 to a class C2 is represented in OWL 2 as:

```
Declaration( ObjectProperty( P ) )
ObjectPropertyDomain( P C1 )
ObjectPropertyRange( P C2 )
```

A link in OWL 2 is represented as a property assertion. The link of an association P between the objects x1 and x2 in an object diagram is represented in OWL 2 as:

```
ObjectPropertyAssertion( P x1 x2 )
```

Moreover, due to the open-world assumptions of OWL 2, for a reasoner to be able to detect a violation of a minimum multiplicity constraint, we need to provide a definitive knowledge about the links, connecting or not connecting the individuals of a domain class and a range class of an association. Therefore, if there is no link between the objects of a domain class and a range class of a UML association, we need to explicitly declare that there is no connection between the individuals. The knowledge about the non-existence of a link between individuals is called a negative assertion. The negative assertion of an association P between the objects x1 and x2 in OWL 2 is written as:

```
NegativeObjectPropertyAssertion( P x1 x2 )
```

A negative assertion is required when there exists an association between the classes but their specific individuals are not connected with a link.

8.2.5 Unique and Non-Unique Associations

A UML association multiplicity can be unique or non-unique. A unique association does not allow two objects to be linked with each other more than one time, as shown in Figure 8.3c.

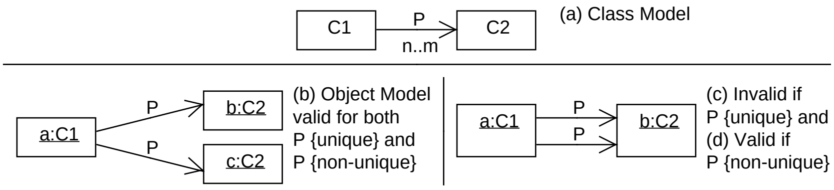
C1
P
n..m
C2
(a) Class Model
a:C1
P
P
b:C2
c:C2
(b) Object Model valid for both P {unique} and P {non-unique}
a:C1
P
P
b:C2
(c) Invalid if P {unique} and (d) Valid if P {non-unique}

Figure 8.3: (a): A class diagram depicting an association P connecting two classes, (b): A consistent object diagram for both unique and non-unique P, (c): An inconsistent object diagram if P is unique, and (d): A consistent object diagram if P is non-unique.

The restriction of unique association P is written in predicate logic as:

$$\forall x, y, l_1, l_2 . P(x, y, l_1) \wedge P(x, y, l_2) \rightarrow l_1 = l_2$$

In the case of non-unique associations this restriction does not apply.

In OWL 2 the UML unique association is treated as a normal object property, and the translation of a multiplicity constraint of a unique association is done by using OWL 2 axioms: ObjectMinCardinality and ObjectMaxCardinality. However, in case of a non-unique association, there can be multiple links between the objects of a domain class and a range class, as shown in Figure 8.3d. The OWL 2 reasoner considers all links which have a common source and target as one link, and to make the reasoner able to consider all those links as different links, we have introduced an intermediate class in between a domain class and a range class of a non-unique association. As a consequence, every non-unique association P is translated in OWL 2 as a combination of two object properties P_I and I_P. Where P_I connects a domain class to an intermediate class, and I_P connects an intermediate class to a range class of a non-unique association. The object property P_I is written in OWL 2 as:

```
InverseFunctionalObjectProperty( P_I )
```

An inverse functional object property restricts an individual of a domain class connecting with more than one individuals of an intermediate class. Furthermore, we put a cardinality restriction of n..m on P_I by using OWL 2 axioms: ObjectMinCardinality and ObjectMaxCardinality. Moreover, the object property I_P is written in OWL 2 as a normal object property. In order to ensure that the individuals of an intermediate class C_I connect one to one with the individuals of a range class, we have to put the exact cardinality of one on the property I_P as:

```
SubClassOf( C_I ObjectExactCardinality( 1 I_P ) )
```

Furthermore, a link of a non-unique association P from object a to b in an object diagram is translated as the assertions of the object properties P_I and I_P as:

```
ObjectPropertyAssertion( P_I a C_I_# )
ObjectPropertyAssertion( I_P C_I_# b )
```

Where $C\_I\_\#$ is an individual of an intermediate class C_I, and # is an auto generated unique number which is responsible to create a distinction between the identical links of a non-unique association. For example, the translation of identical links of the object diagram shown in Figure 8.3d is:

```
\\Link 1
ObjectPropertyAssertion( P_I a C_I_1 )
ObjectPropertyAssertion( I_P C_I_1 b )
\\Link 2
ObjectPropertyAssertion( P_I a C_I_2 )
ObjectPropertyAssertion( I_P C_I_2 b )
```

8.2.6 Ordered Properties

In UML ordering, the links of an ordered property are labeled with a unique numbered index, and it is required that the indexes are in order. An ordering would be used for example to preserve a sequence of a parameter in a function. A link of an ordered property P connecting object x of the source class and object y of the target class of the ordered property P having a label i is represented in predicate logic as:

$$P(x, y, i)$$

Furthermore, all links of an ordered property having identical index i are required to have an identical source and target, in predicate logic it is represented as:

$$\forall i \in N\ \forall x, y, a, b.P(x, a, i) \wedge P(y, b, i) \rightarrow (x = y) \wedge (a = b)$$

Where, x, y are the objects of a domain class and a, b are the objects of a range class of an ordered property P.

The UML ordered property is translated in OWL 2 as a normal objectproperty. The translation of basic constraints like domain, range and multiplicity is also the same as mentioned in Section 8.2.4. For example the translation of an ordered property P depicted on top of Figure 8.4 is as follows.

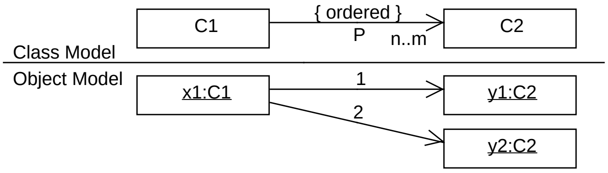

Figure 8.4: Top: A UML class diagram depicting ordered property. Bottom: A UML object diagram depicting ordered links

```
Declaration( ObjectProperty( P ) )
ObjectPropertyDomain( P C1 )
ObjectPropertyRange( P C2 )
SubClassOf( C1 ObjectMinCardinality( n P ) )
SubClassOf( C1 ObjectMaxCardinality( m P ) )
```

Moreover, in a UML object diagram, a link of an ordered property is labeled with an index, and requires that the index is unique and in order. In OWL 2 there is no specific axiom for the representation of a UML ordered property link or any link with a label. Due to this fact, we translate a UML ordered property link into OWL 2 in four steps. First, declare an *index* property for each ordered property link that exist in an object diagram. The *index* property is declared in OWL 2 as:

```
Declaration( ObjectProperty( index_P_# ) )
```

To make every ordered property link reachable while parsing a translated object diagram in OWL 2, the name of an *index* property comprises three parts. First, *index* refers that the link of this object property will represent an ordered property link. Second, *P* refers the name of an ordered property in a class diagram. Third, # is representing an index or a label on a link of an ordered property. The data type of an index can be ”xsd:integer” [97] or ”xsd:string” [97]. For example, for each labeled link 1 and 2 of the ordered property *P* shown at the bottom of Figure 8.4, we will declare an *index* property in OWL 2 as:

```
Declaration(ObjectProperty( index_P_1 ))
Declaration(ObjectProperty( index_P_2 ))
```

Second, a domain and a range of each *index* property in OWL 2 is the same as the domain and the range of an ordered property. Therefore, each `index` property in OWL 2 will be a subproperty of an ordered property *P*:

```
SubObjectPropertyOf( index_P_# P )
```

Third, all links of an *index* property must have an identical source and target. In OWL 2 all links having an identical source and target are considered as one link. Therefore, we have also made the *index* property:

1. FunctionalObjectProperty so that one link of an *index* property may not lead to two individuals.

   ```
   FunctionalObjectProperty( index_P_# )
   ```

2. InverseFunctionalObjectProperty so that two links of an *index* property may not lead to one individual.

   ```
   InverseFunctionalObjectProperty( index_P_# )
   ```

Last, each *index* property in OWL 2 is instantiated among the respective individuals of a domain and a range class of an ordered property. For example the UML ordered property links $P(x1, y1, 1)$ and $P(x1, y2, 2)$ as shown at the bottom of Figure 8.4, is represented in OWL 2 as:

```
ObjectPropertyAssertion( index_P_1 x1 y1)
ObjectPropertyAssertion( index_P_2 x1 y2)
```

8.3 Implementation of a Model Conformance Tool

8.3.1 Translator

We have implemented the translations of UML class and object modeling concepts as discussed in Section 8.2 into a translator by using the model-to-text translation tool MOFScript [5]. The implemented translator in MOFScript allows us to automatically transform the UML class and object diagrams into OWL 2. The translator takes a UML class diagram and an object diagram as an input in the form of UML XMI 2.1 [73], which is parsed according to UML 2. The output of the translator is an ontology, which contains a transformed object diagram and its class diagram in the from of OWL 2 functional syntax. The translator script can be downloaded from [4].

We have translated the class diagram and its inconsistent object diagram as shown in Figure 8.5, into an OWL 2 DL ontology, by using the implemented translator, the output ontology generated by the translator is as follows:

```
Declaration(Class(Operation))
SubClassOf( Operation_Direct Operation )
EquivalentClasses( Operation Operation_Direct )
```

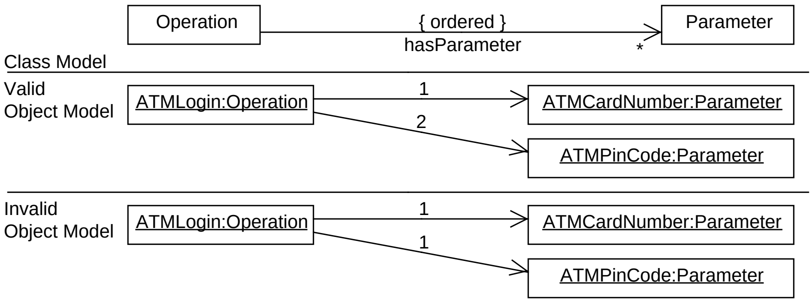

Figure 8.5: Top: UML class diagram depicting Automated Teller Machine (ATM) login operation by using ordered property, Middle: Consistent UML object diagram, Bottom: Inconsistent UML object diagram due to the non-unique index link.

```
DisjointClasses( Operation_Direct Parameter )
Declaration(Class(Parameter))
SubClassOf( Parameter_Direct Parameter )
EquivalentClasses( Parameter Parameter_Direct )
DisjointClasses( Parameter_Direct Operation )
Declaration(ObjectProperty( hasParameter ))
ObjectPropertyDomain(has_Parameter Operation)
ObjectPropertyRange(has_Parameter Parameter)
SubClassOf( ObjectOneOf( ATMLogin ) Operation_Direct )
SubClassOf( ObjectOneOf( ATMCardNumber ) Parameter_Direct )
SubClassOf( ObjectOneOf( ATMPinCode ) Parameter_Direct )
ObjectPropertyAssertion( hasParameter ATMLogin ATMCardNumber)
ObjectPropertyAssertion( hasParameter ATMLogin ATMPinCode)
SubObjectPropertyOf( index_hasParameter_1 hasParameter )
FunctionalObjectProperty( index_hasParameter_1 )
InverseFunctionalObjectProperty( index_hasParameter_1 )
ObjectPropertyAssertion( index_hasParameter_1 ATMLogin ATMCardNumber)
SubObjectPropertyOf( index_hasParameter_1 hasParameter )
FunctionalObjectProperty( index_hasParameter_1 )
InverseFunctionalObjectProperty( index_hasParameter_1 )
ObjectPropertyAssertion( index_hasParameter_1 ATMLogin ATMPinCode)
DifferentIndividuals( ATMLogin ATMCardNumber ATMPinCode )
```

8.3.2 Reasoning Engine

The conformance of an object diagram against a class diagram is done by validating an output ontology using an OWL 2 reasoner. The output ontology contains the transformed object diagram and its class diagram in the form of OWL 2. Since the translation tool produces an output ontology in OWL 2 functional syntax, validation of the output ontology can be done by using any ontology reasoner which supports OWL 2 functional syntax.

8.3.3 Conformance Report

A conformance report is an output of an OWL 2 reasoner. After the validation of a transformed object diagram against its class diagram, a reasoner will produce a conformance report. This report contains the details about the inconsistencies present in a transformed object diagram against its class diagram. The output of a translation tool which contains the transformed object diagram and its class diagram is then validated by a reasoner. The conformance report of the output ontology produced by a reasoner is as follows:

```
Consistent: No
Reason: Individual ATMLogin has more than one value
for the functional property index_hasParameter_1
```

The above conformance report is generated by using the Pellet OWL 2 reasoner version 2.3.0 [90]. The conformance report clearly indicates the inconsistency that exists in the object diagram that there exists more than one link of an ordered property *hasParameter* with an *index* label 1.

8.3.4 Tool Validation

In order to evaluate the proposed validation approach, the proposed model translation tool has been validated by a suite of test cases that covers all class and object modeling concepts discussed in Section 8.2. Each test case includes a class diagram, a valid object diagram, and an invalid object diagram. Each test case is transformed into OWL 2 using the proposed translation tool and validated by using an OWL 2 reasoner. The details of test cases and the summary of the conformance report is expressed in Figure 8.6.

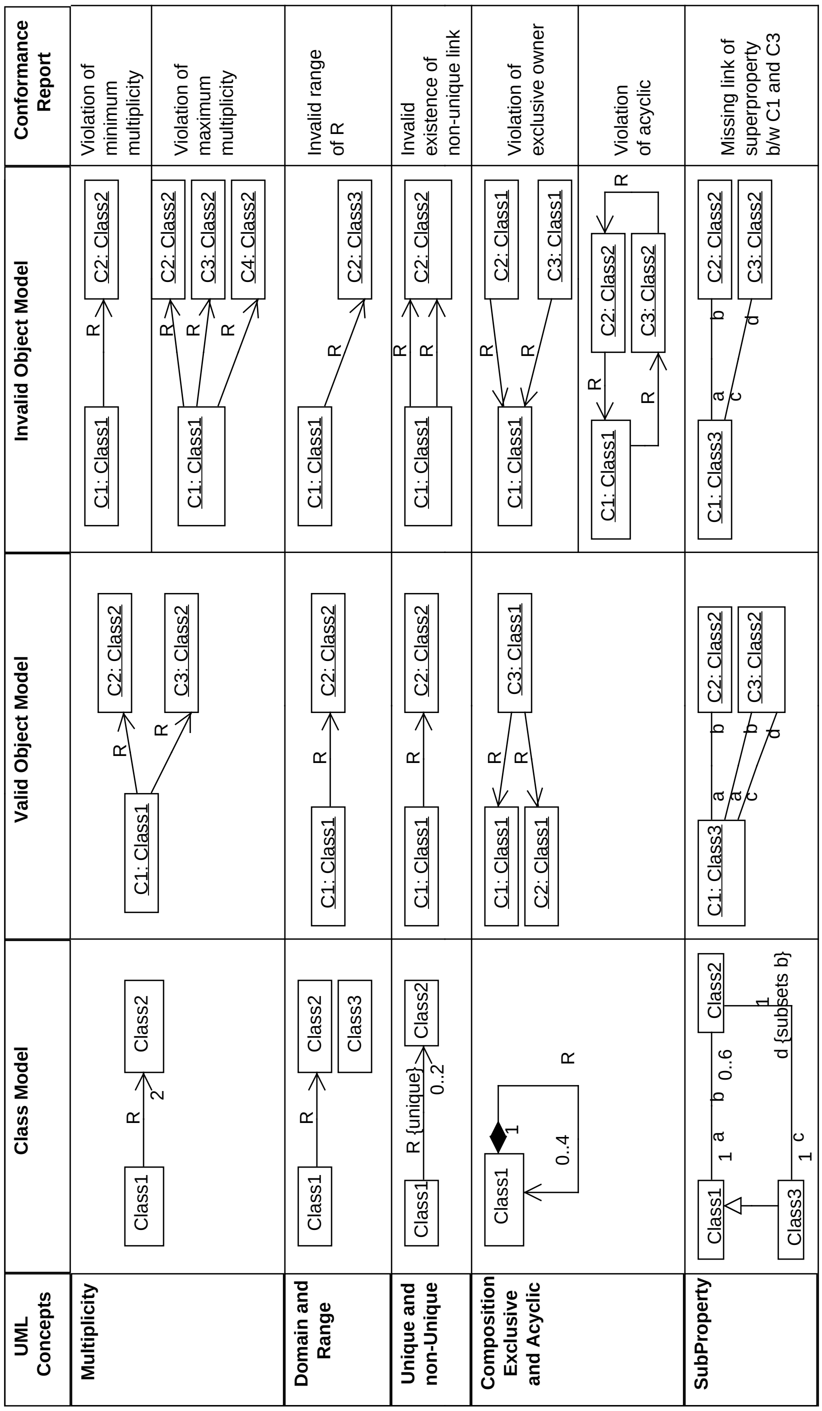

Figure 8.6: List of test cases and the conformance report summary of invalid object diagrams.

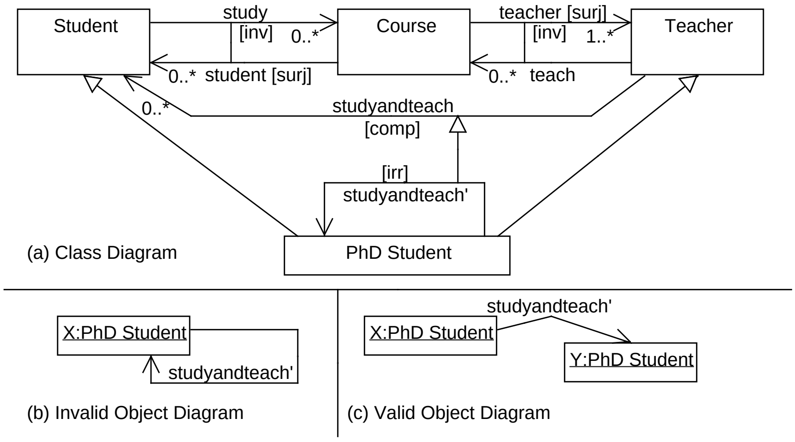

Figure 8.7: (a) The UML metamodel depicting a condition that a PhD student cannot enroll in a course that he is teaching by himself. (b) Invalid object diagram, (c) Valid object diagram.

8.4 Evaluation

8.4.1 Conformance of Objects against DPF Constraints

In order to check the conformance of an object diagram against a class diagram with DPF constraints(described in chapter 4), we first translate these diagrams and constraints into OWL 2, then by using OWL 2 reasoners we check whether the object diagrams conform to its class diagram and DPF constraints or not. For example, let us consider a university scenario in which PhD students are not allowed to enroll in a course which they are teaching themselves. The UML class diagram of this scenario representing classes, associations and constraints is shown in Figure 8.7(a) and the object diagram shown in Figure 8.7(b) depicts an object X that is linked with itself by using an instance of an irreflexive association $studyandteach'$, which violates the irreflexive constraint $[irr]$ and makes the object diagram invalid, whereas, the object diagram shown in Figure 8.7(c) is valid because it is not violating any constraints provided in the class diagram.

DPF Models to OWL 2

The OWL 2 translation of an object diagram(Figure 8.7(a)) and a class diagram(Figure 8.7(b)) is given in Figure 8.8 and Figure 8.9.

```
//Class Diagram
Declaration(Class(Student))
Declaration(Class(Course))
Declaration(Class(Teacher))
Declaration(Class(PhDStudent))
Declaration(ObjectProperty( study ))
ObjectPropertyDomain(study Student)
ObjectPropertyRange(study Course)
Declaration(ObjectProperty( student ))
ObjectPropertyDomain(student Course)
ObjectPropertyRange(student Student)
InverseObjectProperties( study student )
Declaration(ObjectProperty( teach ))
ObjectPropertyDomain(teach Teacher)
ObjectPropertyRange(teach Course)
Declaration(ObjectProperty( teacher ))
ObjectPropertyDomain(teacher Course)
ObjectPropertyRange(teacher Teacher)
InverseObjectProperties( teach teacher )
Declaration(ObjectProperty( studyandteach ))
ObjectPropertyDomain(studyandteach Teacher)
ObjectPropertyRange(studyandteach Student)
SubClassOf( PhDStudent
ObjectUnionOf( Student Teacher ) )
Declaration(ObjectProperty( studyandteach' ))
ObjectPropertyDomain(studyandteach' PhdStudent)
ObjectPropertyRange(studyandteach' PhdStudent)
SubObjectPropertyOf(studyandteach' studyandteach )
IrreflexiveObjectProperty(studyandteach')
//Object Diagram
SubClassOf( ObjectOneOf( X ) PhDStudent )
ObjectPropertyAssertion( studyandteach' X X)
```

Figure 8.8: The OWL 2 translation of models shown in Figure 8.7(a) and Figure 8.7(b).

```
<swrl:Variable rdf:ID="x"/>
<swrl:Variable rdf:ID="y"/>
<swrl:Variable rdf:ID="z"/>
<owl:ObjectProperty rdf:ID="teach"/>
<owl:ObjectProperty rdf:ID="student"/>
<owl:ObjectProperty rdf:ID="studyandteach"/>
<swrl:Imp>
<swrl:body rdf:parseType="Collection">
 <swrl:IndividualPropertyAtom>
  <swrl:propertyPredicate rdf:resource="#teach"/>
   <swrl:argument1 rdf:resource="#x"/>
   <swrl:argument2 rdf:resource="#y"/>
 </swrl:IndividualPropertyAtom>
 <swrl:IndividualPropertyAtom>
  <swrl:propertyPredicate rdf:resource="#student"/>
   <swrl:argument1 rdf:resource="#y"/>
   <swrl:argument2 rdf:resource="#z"/>
  </swrl:IndividualPropertyAtom>
</swrl:body>
<swrl:head rdf:parseType="Collection">
 <swrl:IndividualPropertyAtom>
  <swrl:propertyPredicate rdf:resource="#studyandteach"/>
   <swrl:argument1 rdf:resource="#x"/>
   <swrl:argument2 rdf:resource="#z"/>
  </swrl:IndividualPropertyAtom>
</swrl:head>
</swrl:Imp>
```

Figure 8.9: The SWRL rule for the [*comp*] constraint of the association *studyandteach* shown in Figure 8.7(a).

```
D:\pellet-2.3.0>pellet consistency -l OWLAPI D:\example.OWL2FS
Consistent: No
Reason: Irreflexive property nullstudyandteachPrime
```

Figure 8.10: The validation report of the OWL 2 ontology of the class and object diagram shown in Figure 8.8 and Figure 8.9.

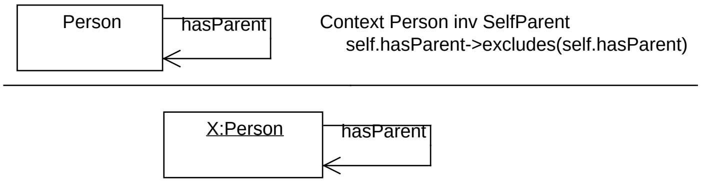
Person
hasParent
Context Person inv SelfParent
self.hasParent->excludes(self.hasParent)
X:Person
hasParent

Figure 8.11: Top: A UML class diagram with OCL constraints. Bottom: The object diagram of a class diagram depicted on top.

Reasoning

We validate the OWL 2 translations of DPF UML diagrams by using an OWL 2 reasoner. Since, the translations are in OWL 2 functional syntax (OWL2fs), we can use any OWL 2 reasoner for the purpose of validation which supports OWL2fs.

After the validation of the translated UML diagrams, the reasoner produces a validation report. The validation report contains detailed analysis of the inconsistencies in the translated diagrams. The validation report of the OWL 2 ontology of the class and object diagram shown in Figure 8.8 and Figure 8.9 is given in Figure 8.10.

The validation report shown in Figure 8.10 clearly indicates the violation of an irreflexive constraint of association *studyandteach′*, and results the whole ontology as inconsistent.

8.4.2 Conformance of Objects against OCL Constraints

In order to check the conformance of an object diagram against a class diagram with OCL constraints, we need to first translate these diagrams and OCL constraints into OWL 2 by following the translations discussed in previous chapters. The next step is to validate the OWL 2 translations by using an OWL 2 reasoner. As an example we have a class diagram shown in Figure 8.11 consisting of a class name $Person$, and the association name $hasParent$. Along with the class diagram there is an OCL constraint

```
//Class Diagram
Declaration(Class(Person))
Declaration(ObjectProperty( hasParent ))
ObjectPropertyDomain(hasParent Person)
ObjectPropertyRange(hasParent Person)

//OCL Constraint
IrreflexiveObjectProperty(hasParent)

//Object Diagram
SubClassOf( ObjectOneOf( X ) Person )
ObjectPropertyAssertion( hasParent X X)
```

Figure 8.12: The OWL 2 translation of models shown in Figure 8.11.

name SelfParent which has a context class *Person*. This OCL constraint is applying a restriction on the objects of a class person that they cannot link to them selves by using an instance/link of an association *hasParent*. At the bottom of Figure 8.11 we have an object diagram depicting an object *X* of the class *Person*. This object is linked with itself by using a link of the association *hasParent*. In order to determine that the object diagram confirms to the class diagram, we first translate these diagrams and OCL constraint into OWL 2.

OCL to OWL 2

The translation of a class diagram and an object diagram along with OCL constraints (Figure 8.11) into OWL 2 is shown in Figure 8.12.

Reasoning

After translating the class and object diagram with OCL constraint we pass the OWL 2 ontology to the OWL 2 reasoner. The reasoner parses the ontology and creates the validation report. The validation report of the ontology of the diagrams and constraint shown in Figure 8.11 is as follows:

```
D:\pellet-2.3.0>pellet consistency -l OWLAPI D:\OCL.OWL2FS
Consistent: No
Reason: Irreflexive property nullhasParent
```

The above validation report clearly shows the violation of the OCL constraint that an object cannot link with its own self by using a link of the association $hasParent$ i.e. $Irreflexive(hasParent)$. Due to the violation of the OCL constraint the reasoner finds the ontology inconsistent.

8.5 Conclusion

In this chapter, we have presented an approach to reasoning about the conformance of a UML object model against its class model using an OWL 2 reasoner. Furthermore, we have discussed the implementation of the translations as an automatic model translation tool.

The approach is fully automated thanks to the translation tool and the existing OWL 2 reasoners. Since the translation tools accept standard UML models serialized using the XMI standard, the approach can be easily integrated with existing UML modeling tools. Unfortunately, the validation report generated by OWL 2 reasoners is not always self-explanatory, because the relationship between UML concepts and OWL 2 axioms is not always obvious. As a consequence, it is not always possible to immediately point out the cause of the problem based on these violations without manual inspection of the validation report and the problematic object models. It would greatly add to the usefulness of the method to have some sort of automated discovery of the cause of violations.

Chapter 9

Consistency of Multiple UML Diagrams using OWL 2

In this chapter we discuss a method for the validation of multiple UML diagrams of the same metamodel using OWL 2 reasoners. This chapter is based on the Articles I, III, IV and V.

9.1 Introduction

UML is a language for modeling that is widely used during software development [77]. Each software development project involves the creation of many models. These models may represent different versions of the same software component, often designed in parallel by a number of designers. These different versions of a model may create contradictions when combined. This raises a need of a mechanism to semantically merge different versions of a UML model together and find out the possible contradictions and inconsistencies arise between model elements when they are viewed together.

The issue of merging UML models has been studied in the research literature previously [18, 7, 67, 64]. However, these approaches do not provide the mechanism to check the consistency of merged models. In this chapter, we propose an approach to study how different models of the same metamodel are merged, and how to validate the merged models. In the validation of merged models, we want to identify the possible inconsistencies that arise when the different models of the same metamodel are viewed together/merged. In this approach we use a decidable fragment of Web Ontology Language (OWL 2 DL) [20] to represent and merge UML models, and then use OWL 2 reasoning tools [90, 83] to determine the inconsisten-

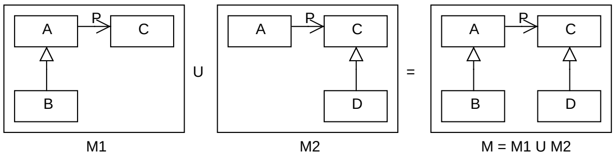

Figure 9.1: The merge or union of two versions of a Model.

cies in the merged model. We select OWL 2 DL to represent the UML models because we consider it well supported and adapted, and there exist several OWL 2 reasoners [90, 83] for checking concept satisfiability. The details about the translation of UML models into OWL 2 and the validation of UML models using OWL 2 reasoners has been discussed in chapters 4, 6 and 8.

9.2 Model Merging using OWL 2

In this section we show how to perform model merging. The merging of given models is performed by putting the union of all model elements of all models in to a single model, i.e. the merged model, such as, if $M1$ and $M2$ are given models then the merged model M represents the union of all model elements of all given models, i.e., $M = M1 \cup M2$. For example, in Figure 9.1 the merged model M is a union ($M1 \cup M2$) of given models $M1 = \{A, C, B\}$ and $M2 = \{A, C, D\}$.

In order to merge the UML models representing different versions of a UML model, we propose to use Description Logic [48]. Furthermore to detect the inconsistencies originating from the merging of different versions of a model, we propose to use the automated reasoning tools [90, 83]. A number of UML models representing different versions of a UML model are taken as an input. All the inputs are translated to OWL 2 DL and then merged into a single ontology. Next, the OWL 2 ontology of UML models is passed to a reasoner. Finally, the reasoner processes the ontology and produces a validation report. The validation report reveals the inconsistencies in the ontology representing a merged UML model.

In order to demonstrate the merging of different versions of a UML model, we first translate all UML models into OWL 2 DL by using the method discussed in previous chapters and merge the OWL 2 translations of all UML models into a single ontology. Since we translate all model elements of all UML models into a single ontology, the common elements of all models (i.e. $M1 \cap M2$ in our example Figure 9.1) will overlap. Additionally,

```
//M1
Declaration(Class(A))
Declaration(Class(C))
Declaration(Class(B))
SubClassOf( B A )
Declaration(ObjectProperty( P ))
ObjectPropertyDomain(P A)
ObjectPropertyRange(P C)

//M2
Declaration(Class(A))
Declaration(Class(C))
Declaration(Class(D))
SubClassOf( D C )
Declaration(ObjectProperty( P ))
ObjectPropertyDomain(P A)
ObjectPropertyRange(P C)
```

Figure 9.2: The OWL 2 ontology of models $M1$ and $M2$ shown in Figure 9.1.

due to the open-world assumptions [59] of OWL 2, where model elements are recognized by their names, all model elements having the same name are considered as a one model element or concept. Consequently, due to this assumption all models will merge or connect with each other by using common model elements. In Figure 9.2 we provide an example of the OWL 2 translation of models $M1$ and $M2$ given earlier in Figure 9.1.

The given models $M1$ and $M2$ represent six classes in total i.e., $M1 = \{A, C, B\}$ and $M2 = \{A, C, D\}$. We translate all six classes of $M1$ and $M2$ into a single ontology. Due to the unique name assumption of OWL 2, the reasoner recognises distinct classes, and counts all six classes, i.e, $M1 = \{A, C, B\}$ and $M2 = \{A, C, D\}$ as four classes, i.e., $M1 \cup M2 = \{A, C, B, D\}$.

The unique name assumption of OWL 2 is also applied on the relationships such as associations, generalization and on association constraints such as multiplicity, composition, domain and range constraints. For example, both models $M1$ and $M2$ represent four relationships in total, in which there are two associations and two generalization relationships. Both models depict association P from the class A to the class C, therefore, due to the unique name assumption of OWL 2, the reasoner recognises the distinct associations and count three relations instead of four. Within these relations there are two generalization relationships and one association.

In order to determine the number of elements in an ontology, the rea-

```
D:\pellet-2.3.0>pellet classify -l OWLAPI D:\Merging.owl2fs
Classifiying 7 elements
Classifiying:  100% complete in 00:00
Classifiying finished in 00:00

 owl:Thing
    merging.owl2fs:A
       merging.owl2fs:B
    merging.owl2fs:C
       merging.owl2fs:D
```

Figure 9.3: The classification report of the OWL 2 ontology shown in Figure 9.2 generated by Pellet.

soner produces a classification report of an ontology. The classification report of the ontology shown in Figure 9.2 is given in Figure 9.3. The classification report clearly shows that the reasoner recognises the distinct model elements of all models, i.e., $M1 \cup M2 = \{A, C, B, D\}$. It also shows the total number of seven model elements found in the ontology, which includes four classes $\{A, C, B, D\}$, two subclass relationships, and one association.

9.3 Consistency Analysis of Merged Models

To explain the consistency of a merged model, we assume that there is a nonempty set $\Delta^{\mathcal{I}}$ called the object domain containing all the possible objects in our domain. We propose that a merged model depicting a class diagram is interpreted as a number of subsets of $\Delta^{\mathcal{I}}$ representing each class in the merged model and as a number of conditions that need to be satisfied by these sets. The merged model is consistent, if each class in a merged model can be instantiated i.e, if C is a class in a merged model and $C \subseteq \Delta^{\mathcal{I}}$ then $C \not\equiv \perp^{\mathcal{I}}$ must hold.

Moreover, in order to check the model conformance (that an object diagram conforms to the class diagram), we want to know, if each object o in an object diagram is a proper instance of a class C in a class diagram i.e, $o \in C$. Moreover, we want to investigate that each link in an object diagram is an instance of an association depicted in a class diagram. Also, objects and links must preserve the constraints applied in a class diagram such as: uniqueness, multiplicity, source, target, ordering and composition constraints of their classes and associations.

9.3.1 Validation of Merged Models

The consistent UML models representing different versions of a UML model may have inconsistencies when merged. For example UML models $M1$ and

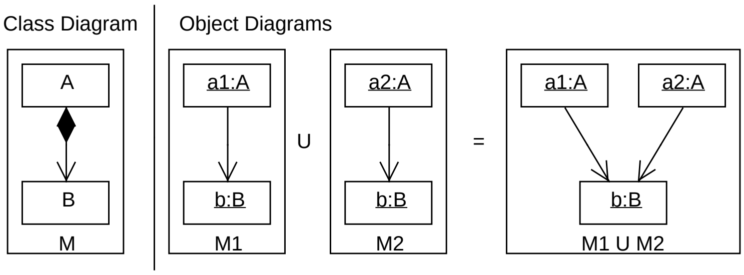

Figure 9.4: The invalid merge of two valid UML models.

$M2$ shown in Figure 9.4 are valid and consistent models, but when they are merged, the merged model $M1 \cup M2$ is inconsistent, because it is violating the exclusive ownership constraint of composition. Exclusive ownership means that an object can have only one owner.

In order to detect the inconsistencies occurred by the merging of models, we first translate the models into the OWL 2 ontology, and then use an OWL 2 reasoner to detect the inconsistencies in the translated ontology. To demonstrate our validation approach we have translated the models M, $M1$ and $M2$ shown in Figure 9.4 into an ontology. The generated ontology is shown in Figure 9.5.

9.3.2 Reasoner

In order to detect the inconsistencies in an ontology(Fiure 9.5) of a merged model(Figure 9.4), the OWL 2 ontology of the merged model is passed to the OWL 2 reasoner. Since the ontology is in OWL2fs format, we can use any reasoner which supports this format. In our example we have used Pellet [90] version 2.3.0. The reasoner processes the ontology and reveals the inconsistencies in the ontology in the form of a validation report.

9.3.3 Validation Report

The validation report indicates the contradictions and inconsistencies in the ontology caused by the model merge. The validation report of the ontology (shown in Figure 9.5) of models (Figure 9.4) is as follows:

```
D:\pellet-2.3.0>pellet consistency -l OWLAPI D:\METest.owl2fs
Consistent: No
Reason: Individual file:D:/METest.owl2fs#b has more than one
value for the functional property inv(file:D:/METest.owl2fs#owns)
```

```
//These are the global properties enforcing composition restrictions
IrreflexiveObjectProperty(rules:contains )
SubObjectPropertyOf( owns rules:contains )
InverseFunctionalObjectProperty( owns )

//Class Diagram M
Declaration(Class(A))
Declaration(Class(B))
Declaration(ObjectProperty( association_A_B ))
ObjectPropertyDomain(association_A_B A)
ObjectPropertyRange(association_A_B B)
SubObjectPropertyOf(association_A_B owns )

//Object diagram M1
SubClassOf( ObjectOneOf( A ) a1 )
SubClassOf( ObjectOneOf( B ) b )
ObjectPropertyAssertion( association_A_B a1 b)

//Object diagram M2
SubClassOf( ObjectOneOf( A ) a2 )
SubClassOf( ObjectOneOf( B ) b )
ObjectPropertyAssertion( association_A_B a2 b)

DifferentIndividuals( a1 a2 b )

// The SWRL rule for capturing transitivity for acyclic constraint of composition.
<rdf:RDF>
    <owl:Ontology rdf:about="compoRule"/>
    <swrl:Variable rdf:ID="x"/>
    <swrl:Variable rdf:ID="y"/>
    <swrl:Variable rdf:ID="z"/>
    <swrl:Variable rdf:ID="a"/>
    <swrl:Variable rdf:ID="b"/>
    <owl:ObjectProperty rdf:ID="contains"/>
    <owl:ObjectProperty rdf:ID="owns"/>
    <swrl:Imp>
        <swrl:body rdf:parseType="Collection">
            <swrl:IndividualPropertyAtom>
                <swrl:propertyPredicate rdf:resource="contains"/>
                <swrl:argument1 rdf:resource="#x"/>
                <swrl:argument2 rdf:resource="#y"/>
            </swrl:IndividualPropertyAtom>
            <swrl:IndividualPropertyAtom>
                <swrl:propertyPredicate rdf:resource="contains"/>
                <swrl:argument1 rdf:resource="#y"/>
                <swrl:argument2 rdf:resource="#z"/>
            </swrl:IndividualPropertyAtom>
        </swrl:body>
        <swrl:head rdf:parseType="Collection">
            <swrl:IndividualPropertyAtom>
                <swrl:propertyPredicate rdf:resource="contains"/>
                <swrl:argument1 rdf:resource="#x"/>
                <swrl:argument2 rdf:resource="#z"/>
            </swrl:IndividualPropertyAtom>
        </swrl:head>
    </swrl:Imp>
</rdf:RDF>
```

Figure 9.5: The OWL 2 ontology of the models M, $M1$ and $M2$ (Figure 9.4).

The validation report clearly indicates the violation of the mutually exclusive ownership constraint of composition is that the object *b* has more than one value for the inverse functional property *owns*. In our OWL 2 translations of a composition the inverse-functional object-property *owns* represents the exclusive ownership constraint of the composition. The details about the translation of composition constraints have already been discussed in Chapter 4.

9.4 Conclusion

In this chapter we have presented an approach to semantically merge UML models depicting different versions of a UML model. We have also discussed a mechanism to validate the merged models. Moreover, we have demonstrated our work using example models. The automatic discovery of inconsistencies in a merged model by the reasoner clearly shows the usefulness of the proposed merging and the validation approach.

This approach is purely based on the lessons learned from the work presented in the previous chapters. Therefore, on the basis of the results of the viability tests that we have shown in the previous chapters, we can claim that this approach is also viable, because it is using the same set of OWL 2 translations used in the previous approaches.

Chapter 10

Conclusions

This chapter discusses the conclusions of this thesis. The conclusions that we discuss here are categorized according to the research problems that are addressed in different chapters of this thesis.

Consistency of class diagrams: We addressed the problem of checking the consistency of class diagrams by translating class diagrams into OWL 2 ontology, and then use an OWL 2 reasoner for the reasoning of translated OWL 2 ontology. Also, we successfully demonstrated the viability of the proposed approach by validating more than 300 metamodels/class diagrams available at the Atlantic Metamodel Zoo, which comprises thousands of model elements. Out of these metamodels 49% has errors. This shows the significance of the problem of the validation of metamodels. Furthermore, these results affirm the valuable contribution of our approach in this validation area.

Consistency of diagrams expressed in different languages: The problem of checking the consistency of diagrams expressed using different language is addressed by using a specific type of interaction, i.e., the interaction of static and dynamic diagrams of a system. In this interaction a class diagram explaining the static content of a system interacts with a statechart diagram depicting a behavior of one of the class existing in the class diagram. In this case the interaction is done by using state invariants expressed using OCL. This whole system represented by a class diagram and a statechart diagram with state invariants is automatically translated into OWL 2 and then validated by using an OWL 2 reasoner. We have validated this approach using two different scenarios: the content management system and the REST web service interface. These scenarios have been validated by using both valid and mutated models, and the detection of mutants during the validation process clearly indicated the usefulness of the proposed approach.

Validation of OCL constraints: In general OCL is undecidable, and due to this it is not possible to do the reasoning of OCL constraints. However, previous research shows that if we restrict our approach to a limited known set of OCL constructs, then the undecidability can be avoided. Therefore, in our approach we restrict ourselves to the limited set of OCL constructs that comprise value, multiplicity and boolean constructs. These constructs are the basic OCL constructs, and are used to express the conditions in the states of a statechart diagram. We have validated the OCL constraints by translating them into OWL 2 and then used OWL 2 reasoners for the reasoning about the translated constraints. We have also shown the validation of the proposed approach using both valid and mutated OCL constraints. The detection of the mutated OCL constraints by the proposed approach is an evidence of the correctness of this approach. The performance of this approach was also tested by using both valid and mutated models consisting of 10 to 2000 model elements. The translation and the reasoning time on all models was less than 4.5 seconds in all cases. This shows once again that the proposed approach can process relatively large UML models in a few seconds.

Conformance of a model against its metamodel: We address the issue of the model conformance by expanding the approach that is used for the metamodel validation. The expansion consists of the translations of those model elements that are inadequately or not at all addressed in a related research. These include the OWL 2 translations of modeling concepts such as non-unique association, ordered property, open-closed world issues and composition (including unshearedness and acyclicity). Since the main approach has already been tested on a large scale by validating the metamodels of the Atlantic Metamodel Zoo, this time we have only tested this approach by using a set of test cases containing both valid and mutated models. The detection of the inconsistencies during the validation of test cases is the evidence of the viability of this approach.

Validation of multiple models: The validation of multiple models of the same metamodel is based on the approach of translating all models into a single ontology, and then use the OWL 2 reasoner for the reasoning. This approach is utilizing one of the open-world assumptions of OWL 2 that two concepts having the same name are considered as a single concept. The base of this approach is the same as the approach mentioned in the metamodel or model validation chapters. Therefore on the basis of the lessons learned from the viability test of the work of metamodel or model validation. We claim that this approach is also viable because it is using the same set of OWL 2 translations that are used in the work of metamodel or model validation.

Integration of our approach in existing tools: There are numerous tools available for the modeling of UML models. In these tools, each UML model is generally specified in two forms, one is a graphical form, and other is a syntactical form. The standard for the graphical form of UML models followed by the well known modeling tools is more or less the same. Unfortunately, the syntax of the XML version of UML models generated by different modeling tools are usually not the same. In our implementation we have followed the syntactic standard of UML 2 version 3.0.0. This allows us to integrate our approach with any standard compliant UML modeling tool that follows this UML standard. In order to show the integration of our approach with the existing standard UML modeling tools, we have used EMF plugin Topcased and Magicdraw for the modeling of the UML models that are presented in this thesis.

The approach we proposed in this thesis is fully automated thanks to the implemented translation tool and the existing OWL 2 reasoners. A very constructive potential improvement in the current work would be to enhance the way problems in ontologies are reported. In few cases, the relationship between UML concepts and OWL 2 axioms is not so obvious that it is possible to immediately point out the cause of the problem based on these violations without a manual inspection of the validation report and the problematic models. It would greatly add to the usefulness of the method to have some sort of automated discovery of the cause of violations. However, the metamodels or models that were not found to have problems are not guaranteed to be free of errors, they are just free from inconsistencies that our approach can detect.

Nonetheless, we consider that the use of languages and tools envisioned for the semantic web as a foundation for software modeling languages and tools is a promising proposition. The existing consistency issues of metamodels and models faced by current modeling tools could be addressed by reusing results from the semantic web community. However, the successful detection of the errors in the models present in the online model repository, and the results of the performance test of the proposed approach, is the evidence that the proposed approach is viable, and can be practised in the industry.

Appendices

Appendix A

Validation Results of the Metamodel at the Atlantic Zoo

A.1 Composition cycle errors between several classes

Metamodel	Classes involved
Agate	Site, GenericSite, ReusableObject
BusinessProcessModel	Task, CompoundTask
ChocoModel	Expression, Constraint
classDiagram	Class, Classifier, Interface
DotNET_SystemReflection	Type, MemberInfo
EclipseLaunchConfiguration	MapEntry, Attribute, LaunchConfiguration
HierarchicalSignalFlow	Base, Compound
QoS_Statement	QoSStatement, CompoundQoSStatement
RDFS	Graph, Resource
SignalFlow	CompoundComponent, BaseComponent
UML_withReuseContracts	ReuserClause, CompositeReuser
UnixFS	Directory, File
XUL-Interactorl	Container, Interactor

A.2 Unsatisfiable class due to multiplicity constraints

Metamodel	Classes involved
OCL_Expressions	VariableInitialization

A.3 Composition cycle errors involving a single class

Metamodel	Classes involved
Agate	Package
AntScripts	TaskElement
BusinessProcessModel	CompoundTask
CADM	MaterielItem
CPR	Plan, Action, Actor
deployment	Node
DoDAF	Performer
FiniteStateMachine	RootFolder
GeoTrans	GeoTransfo
KlaperPropCheck	Behavior, Step
MavenProject	Project
PDG	ExpressionNode
ProMarte	Step
SecureUML	Role
SimulinkStateFlow	System
UMLDI-Collaborations	Instance
UMLDI-UseCasesPropCheck	Instance

A.4 Classes forced to have multiple owners

Metamodel	Classes involved
ACG	Expression
ACME	System
Agate	ReusableObject
Amble	Action
Ant	Includes, InExcludes, PatternSet, FileSet
AnyLogic	Parameter, Point
Architectural_Description	ModelElement
AsmL	Main
ASM	Argument
ATL	Statement
ATOM	Generator
Bossa	SeqStmt
CADM	SoftwareItem
ChocoModel	Expression
ClassicModels	Date
Collaborations_Interactions_UML	AssociationEndRole
CPR	Action
CSM	Step
CWMRelationalData	QueryExpression
deployment	Node
DoDAF	Performer
DoDAF-OV5	OperationalRole
DoDAF-SV4	OperationalRolePA
DoDAF-SV5	SystemCapability
DOT	Compartment
DotNET_SystemReflection	Type
DSLModel	ModelElement
ebXML	Transaction
EclipseLaunchConfiguration	MapEntry
EclipsePlugIn	ExtensionPoint
EXPRESSb	ScopedId
EXPRESS	Express_metamodel::Core::Expression
GeoTrans	GeoTransfo
GraphML	Graph
GUI	Item, Row, Composant, Group
HAL	ReferenceBiblioType
HierarchicalStateMachine	CompoundState
HTML	TD, TR, BODY
ifc2x3	IfcMeasureWithUnit
IRL	expression
J2SE5	BodyDeclaration
JAVA3	Method
JavaAbstractSyntax	Type
JavaProject	Type

Jess	Expression
KDM	CodeItem
KM3	StructuralFeature, Class, ModelElement, Package
LaTeX	Label
M	Expression, WhereExpression, DerivedType
Mantis	Issue
Marte	Constraint
Matlab	Block, BlockDefaults
MavenMaven	ContentsGoal
METAH	Port
MoDAF-AV	Property
MonitorProgram	Expression
MSOfficeExcel_SpreadsheetMLBasicDef	ValueType
MSOfficeExcel_SpreadsheetMLPrintingSetup	ValueType
MSOfficeExcel_SpreadsheetMLStyles	ValueType
MSOfficeExcel_SpreadsheetMLWorkbookProp	ValueType
MSOfficeExcel_SpreadsheetMLWorksheetOpt	ValueType
MSOfficeExcel_SpreadsheetMLBasicDef	ValueType
MSOfficeWord_WordprocessingMLSimplified	BlockLevelElt
MSOfficeWord_WordprocessingMLStyles	BlockLevelElt
MSOfficeWord_WordprocessingMLTableElts	StringType
MSVisio_DatadiagramMLBasicDef	PageSheet
MSVisio_DatadiagramMLSimplified	ShapesCollection
MSVisio_DatadiagramMLTextFormat	ShapesCollection
MSVisio_DatadiagramMLXForm	ShapesCollection
MTRANS	Expression
News	Date
μOCCAM	Process
OCCAM	Constructor
OCL_Expressions	VariableInitialisation
ODP-CV	TerminationSignature, InteractionSignature
ODP-NV	Stub, EngineeringObject
OpenQVT	RootRule
OWL	PlainLiteral
Pascal	Parameter, Procedure
PDG	ExpressionNode
Perceptory	Characteristic
PluginEclipse	Version
PNML_basic	URI
PNML_modular	URI
PNML_simplified	URI
PNML_structured	URI
Program	Expression
ProMarte	Step
PRR	OclExpression

PtolemyII	ComponentEntity, Attribute
QoS	Parameter
QVT	Ocl_Expression
R2ML	Vocabulary
RSS-2.0	Category
SBVRvoc	PrimaryRepresentation
SCADE	NamedType
Scilab	Link
SEE_Design	Expression
Sharengo	BusinessRule
SignalFlow	CompoundComponent
SimulinkStateFlow	System
SPL	Place
SQLDDL	Value
SQLDML	Expression
SysML	FlowProperty
TextualPathExp	Path
UIML-3.0	Event
UML2	OutputPin
UMLDI	TaggedValue
UMLDI-ActivityGraphs	Action
UMLDI-StateMachines	Action
USECASE1	Parameter
vb	expression
WebApplications_ConceptualModel	LogicElement
WorkDefinitions	Phase, WorkDefinition
WSDL	BindingOperation
XHTML	LinkTypes
XPDL-1.14	Activity
XQuery	XPath, BooleanExp
XSchema	AbstractContent
XUL-Interactorl	Interactor
yUML	Note, Model, ClassReference

A.5 Multiple occurrences of a class name in a package

Metamodel	Classes involved
Conference.owl	Abstract
confious.owl	Abstract
confOf.owl	Abstract
crs_dr.owl	Abstract
edas.owl	Abstract
ekaw.owl	Research_Topic
iasted.owl	Research_Topic
MICRO.owl	Research_Topic
OpenConf.owl	Research_Topic
paperdyne.owl	Research_Topic
PCS.owl	Research_Topic
sigkdd.owl	Research_Topic

Bibliography

[1] AtlanMod metamodel zoo, available at http://www.emn.fr/z-info/atlanmod/index.php/Zoos.

[2] Dresden OCL Toolkit, available at http://www.dresden-ocl.org.

[3] Formal Modeling Using Logic Programming and Analysis, available at http://research.microsoft.com/en-us/projects/formula/

[4] Model Validation MOFScript, available at http://users.abo.fi/akhan/model_validation.m2t.

[5] MOFScript Homepage, available at http://www.eclipse.org/gmt/mofscript/.

[6] Semantic Web and Model Driven Development, available at http://code.google.com/p/twouse/

[7] UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML. Lecture Notes in Computer Science, vol. 2863. Springer (2003)

[8] MOFScript User Guide (2009), document Available at http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide-0.8.pdf.

[9] Alanen, M., Porres, I.: A Metamodeling Language Supporting Subset and Union Properties. Springer International Journal on Software and Systems Modeling 7(1), 103–124 (2007), available at http://www.springerlink.com/content/8k67436222447147/.

[10] Anastasakis, K., Bordbar, B., Küster, J.M.: Analysis of Model Transformations via Alloy. In: Proceedings of MoDeVVa 2007. pp. 47–56 (2007)

[11] Artale, A., Calvanese, D., Ibáñez García, A.: Full satisfiability of UML class diagrams. In: Proceedings of the 29th international conference on Conceptual modeling. pp. 317–331. ER'10, Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1929757.1929788

[12] Audi, R.: The Cambridge Dictionary of Philosophy. Paw Prints (2008)

[13] Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York, NY, USA (2003)

[14] Babenyshev, S., Rybakov, V.: Reasoning and Inference Rules in Basic Linear Temporal Logic BLTL. In: Setchi, R., Jordanov, I., Howlett, R., Jain, L. (eds.) Knowledge-Based and Intelligent Information and Engineering Systems, Lecture Notes in Computer Science, vol. 6277, pp. 224–233. Springer Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-15390-7_23

[15] Balaban, M., Maraee, A.: Consistency of UML Class Diagrams with Hierarchy Constraints. In: Etzion, O., Kuflik, T., Motro, A. (eds.) Next Generation Information Technologies and Systems, Lecture Notes in Computer Science, vol. 4032, pp. 71–82. Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/11780991_7

[16] Balaban, M., Maraee, A.: A UML-based method for deciding finite satisfiability in description logics. In: Description Logics (2008)

[17] Barros, A., Duddy, K., Lawley, M., Milosevic, Z., Raymond, K., Wood, A.: Processes, Roles, and Events: UML Concepts for Enterprise Architecture. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000 - The Unified Modeling Language, Lecture Notes in Computer Science, vol. 1939, pp. 62–77. Springer Berlin Heidelberg (2000), http://dx.doi.org/10.1007/3-540-40011-7_5

[18] Bendix, L., Emanuelsson, P.: Diff and merge support for model based development. In: Proceedings of the 2008 international workshop on Comparison and versioning of software models. pp. 31–34. CVSM, ACM (2008)

[19] Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artif. Intell. 168(1-2), 70–118 (2005)

[20] Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U., Smith, M.: OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. W3 Recommendation (2009), http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/

[21] Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U., Smith, M.: OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax, w3 Recommendation Available at http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/

[22] Boris, M., et al.: OWL 2 web ontology language structural specification and functional-style syntax. W3 Recommendation http://www.w3.org/TR/owl2-syntax/

[23] Boris, M., Peter F, P.S., Bernardo, C.G.: OWL 2 Web Ontology Language Direct Semantics, http://www.w3.org/TR/owl2-direct-semantics/

[24] Boris Motic, Peter F. Patel-Scheneider, Ian Horrocks: OWL 1.1 Web Ontology Language Structural Specification and Functional-Sytle Syntax (May 2007), available at http://www.webont.org/owl/1.1/owl_specification.html.

[25] Broy, M., Cengarle, M.V., Granniger, H., Rumpe, B.: Considerations and Rationale for a UML System Model, pp. 43–60. John Wiley and Sons, Inc. (2009)

[26] Cabot, J., Clariso, R., Riera, D.: Verification of UML OCL class diagrams using constraint programming. IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08) pp. 73–80 (2008)

[27] Cao, X., Miao, H., Xu, Q.: Verifying service-oriented requirements using model checking. In: Proceedings of the 2008 IEEE International Conference on e-Business Engineering. pp. 643–648. ICEBE '08, IEEE Computer Society, Washington, DC, USA (2008)

[28] Chen, W.: Tactic-based theorem proving and knowledge-based forward chaining: An experiment with Nuprl and Ontic. In: Kapur, D. (ed.) Automated Deduction CADE, Lecture Notes in Computer Science, vol. 607, pp. 552–566. Springer Berlin Heidelberg (1992), http://dx.doi.org/10.1007/3-540-55602-8_191

[29] Clark, T., Evans, A., Kent, S.: The Metamodelling Language Calculus: Foundation Semantics for UML. In: Proceedings of the Fundamental Aspects of Software Engineering (FASE). pp. 17–31 (2001)

[30] Clavel, M., Egea, M., da Silva, V.T.: The MOVA Tool: A Rewriting-Based UML Modeling, Measuring and Validation Tool. In: JISBD. pp. 393–394 (2007)

[31] David H. Akehurst and Stuart Kent and Octavian Patrascoiu: A relational approach to defining and implementing transformations between metamodels. Software and System Modeling 2(4), 215–239 (2003)

[32] Dobing, B., Parsons, J.: Current Practices in the Use of UML. In: Akoka, J., Liddle, S., Song, I.Y., Bertolotto, M., Comyn-Wattiau, I., Heuvel, W.J., Kolp, M., Trujillo, J., Kop, C., Mayr, H. (eds.) Perspectives in Conceptual Modeling, Lecture Notes in Computer Science, vol. 3770, pp. 2–11. Springer Berlin Heidelberg (2005), http://dx.doi.org/10.1007/11568346_2

[33] Ebbinghaus, H., Flum, J., Thomas, W.: Mathematical Logic. Undergraduate Texts in Mathematics, Springer (1994)

[34] Egyed, A.: UML Analyzer: A Tool for the Instant Consistency Checking of UML Models. In: Proceedings of ICSE 2007. pp. 793 –796 (may 2007)

[35] Epstein, R.: Predicate logic: the semantic foundations of logic. Wadsworth Thomson Learning (2001)

[36] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, Citeseer (2000), http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[37] Fowler, M.: Domain-Specific Languages. Addison-Wesley Signature Series, Pearson Education (2010)

[38] Garcia, M., Shidqie, A.J.: OCL Compiler for EMF. In: Eclipse Modeling Symposium at Eclipse Summit Europe 2007, Stuttgart, Germany (2007)

[39] Gašević, D., Djurić, D., Devedžić, V.: MDA-based Automatic OWL Ontology Development. Int. J. Softw. Tools Technol. Transf. 9(2), 103–117 (2007)

[40] Gogolla, M., BÃijttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Science

of Computer Programming 69(1-3), 27 – 34 (2007), http://www.sciencedirect.com/science/article/pii/S0167642307001608, special issue on Experimental Software and Toolkits

[41] Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press (1993), http://www.cs.ox.ac.uk/tom.melham/pub/Gordon-1993-ITH.html

[42] Heckel, R., Voigt, H., Kuster, J., Thone, S.: Towards Consistency of Web Service Architectures. In the Proceedings of the 7th World Multiconference on Systemics, Cybernetics, and Informatics (2003)

[43] Henderson-Sellers, B.: On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages. Springer Briefs in Computer Science, Springer (2012)

[44] Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)

[45] Hnatkowska B., Huzar Z., M.J.: Consistency Checking in UML Models. In: ISM'01 (2001)

[46] Hoglund, S., Khan, A.H., Liu, Y., Porres, I.: Representing and Validating Metamodels using OWL 2 and SWRL. In: JCKBSE'10. pp. 133–144. Kaunas University of Technology (2010)

[47] Holovaty, A., Kaplan-Moss, J.: The Definitive Guide to Django: Web Development Done Right. Apress (2007)

[48] Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible $\mathcal{SROIQ}$. In: Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006). pp. 57–67. AAAI Press (2006), download/2006/HoKS06a.pdf

[49] Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML (2004), availible at http://www.w3.org/Submission/SWRL/

[50] Horrocks, I., Patel-Schneider, P.F., Harmelen, F.V.: From SHIQ and RDF to OWL: The Making of a Web Ontology Language. Journal of Web Semantics 1, 2003 (2003)

[51] Horrocks, I., Sattler, U., Tobies, S.: Practical reasoning for expressive description logics. In: Proceedings of LPAR 1999. pp. 161–180. Springer-Verlag, London, UK (1999)

[52] Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems. Bologna, Italy (2006)

[53] Kandé, M.M., Strohmeier, A.: Towards a UML profile for software architecture descriptions. In: Proceedings of the 3rd international conference on The unified modeling language: advancing the standard. pp. 513–527. UML'00, Springer-Verlag, Berlin, Heidelberg (2000), `http://dl.acm.org/citation.cfm?id=1765175.1765230`

[54] Kaneiwa, K., Satoh, K.: Consistency Checking Algorithms for Restricted UML Class Diagrams. In: Proceedings of FoIKS2006, Springer. Springer (2006)

[55] Kent, S.: Model Driven Engineering. In: Proc. of IFM International Formal Methods 2002. LNCS, vol. 2335. Springer-Verlag (2002)

[56] Khan, A.H., Rauf, I., Porres, I.: Consistency of UML Class and Statechart Diagrams with State Invariants. In: Filipe, J., Neves, R.C.d. (eds.) First International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2013. vol. 1, pp. 1–11. SciTePress Digital Library (http://www.scitepress.org/DigitalLibrary/). SciTePress is member of CrossRef (http://www.crossref.org/). (2013)

[57] Khan, A.H., Suenson, E., Porres, I.: Class and Object Model Conformance using OWL2 Reasoners. In: Penjam, J. (ed.) SPLST'11. pp. 126–137. TUT Press (2011)

[58] Khan, A.H., Suenson, E., Porres, I.: Representation and Conformance of UML Models Containing Ordered Properties Using OWL2. In: OrdRing'11. CEUR-WS Proceedings., vol. 2011. ISWC 2011 (2011)

[59] Knorr, M., Alferes, J.J., Hitzler, P.: Local closed world reasoning with description logics under the well-founded semantics. Artificial Intelligence 175(9-10), 1528 – 1554 (2011)

[60] Krogstie, J.: Model-Based Development and Evolution of Information Systems - A Quality Approach. Springer (2012)

[61] Lamo, Y., Wang, X., Mantz, F., MacCaull, W., Rutle, A.: DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment. In: Computer and Information Science 2012, vol. 429, pp. 37–52. Springer (2012)

[62] Liu, L., Özsu, M.T. (eds.): Encyclopedia of Database Systems. Springer US (2009)

[63] Liu, Y., Höglund, S., Khan, A.H., Porres, I.: A Feasibility Study on the Validation of Domain Specific Languages Using OWL 2 Reasoners. In: Proceedings of TWOMDE 2010. CEUR Workshop Preceedings, CEUR (2010)

[64] Lutz, R., Wurfel, D., Diehl, S.: How Humans Merge UML-Models. pp. 177–186. ESEM '11, IEEE (2011)

[65] Machines, I.B., Group, O.M., Software, S.: Ontology definition metamodel (ODM), OMG Document ad/2003-02-23. Available at http://www.omg.org/.

[66] Malgouyres, H., Motet, G.: A UML model consistency verification approach based on meta-modeling formalization. In: Proceedings of SAC2006. pp. 1804–1809. SAC '06, ACM, New York, NY, USA (2006)

[67] Maoz, S., Jan Oliver, R., Bernhard, R.: Summarizing semantic model differences. In: Proceedings of The International Workshop on Models and Evolution. ME (2011)

[68] Maoz, S., Ringert, J.O., Rumpe, B.: Semantically configurable consistency analysis for class and object diagrams. In: MoDELS. pp. 153–167 (2011)

[69] Mendelson, E.: Introduction to mathematical logic; (3rd ed.). Wadsworth and Brooks/Cole Advanced Books & Software, Monterey, CA, USA (1987)

[70] Nentwich, C., Emmerich, W., Finkelstein, A.: Static Consistency Checking for Distributed Specifications. In the Proceedings of the 16th IEEE international conference on Automated software engineering (2001)

[71] OMG: Meta Object Facility, version 1.4 (April 2002), document formal/2002-04-03. Available at http://www.omg.org/.

[72] OMG: Unified Language Specification (March 2003), version 1.5, Document formal/03-03-01, available at http://www.omg.org/.

[73] OMG: XML Metadata Interchange (XMI) Specification, version 2.1 (September 2005), document formal/05-09-01, available at http://www.omg.org/.

[74] OMG: OCL Specification, Version 2.0 (2006), http://www.omg.org/spec/OCL/2.0/

[75] OMG: ODM Ontology Definition Metamodel (2009), http://www.omg.org/spec/ODM/1.0

[76] OMG: UML 2.2 Infrastructure Specification (February 2009), available at http://www.omg.org/.

[77] OMG: UML, Superstructure Specification, Version 2.4.1. Tech. rep. (2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/

[78] Parreiras, F.S.: Marrying model-driven engineering and ontology technologies: The TwoUse approach. Ph.D. thesis (2011)

[79] Parreiras, F.S., Staab, S., Winter, A.: On marrying ontological and metamodeling technical spaces. In: ESEC-FSE '07: Proceedings. pp. 439–448. ACM, New York, NY, USA (2007)

[80] Porres, I., Rauf, I.: Modeling Behavioral RESTful Web Service Interfaces in UML. Accepted in 26th Annual ACM Symposium on Applied Computing Track on Service Oriented Architectures and Programming(SAC 2011) (2011)

[81] Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: A decidable (yet expressive) fragment of OCL. In: Proc. of the 25th Int. Workshop on Description Logics (DL 2012). CEUR Electronic Workshop Proceedings, http://ceur-ws.org/, vol. 846, pp. 312–322 (2012)

[82] Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: Finite Reasoning on UML/OCL Conceptual Schemas. Data and Knowledge Engineering 73, 1–22 (2012)

[83] R, S., B, M., I, H.: HermiT: A highly-efficient OWL reasoner. Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008) (2008)

[84] Rahmani, Oberle, Dahms: An Adjustable Transformation from OWL to Ecore. In: MODELS 2010, Oslo, Norway, October 3-8, 2010. Proceedings. LNCS, Springer (2010)

[85] Rasch, H., Wehrheim, H.: Checking Consistency in UML Diagrams: Classes and State Machines. In: Formal Methods for Open Object-Based Distributed Systems, LNCS, vol. 2884, pp. 229–243 (2003)

[86] Rauf, I., Khan, A.H., Porres, I.: Analyzing Consistency of Behavioral REST Web Service Interfaces. In: Silva, J., Tiezzi, F. (eds.) The

8th International Workshop on Automated Specification and Verification of Web Systems. pp. 1–15. Electronic Proceedings in Theoretical Computer Science (EPTCS). (2012)

[87] Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A formal approach to the specification and transformation of constraints in MDE. In: Journal of Logic and Algebraic Programming. vol. 81, pp. 422 – 457. ELSEVIER (2012)

[88] Sekerinski, E.: Verifying statecharts with state invariants. In: ICECCS. pp. 7–14 (2008)

[89] Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: Proceedings of MoDeVVa 2009. pp. 4:1–4:10. MoDeVVa '09, ACM, New York, NY, USA (2009)

[90] Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web 5, 51–53 (2007)

[91] Tarski, A., Helmer-Hirschberg, O.: Introduction to Logic and the Methodology of Deductive Sciences. Oxford (1946)

[92] Tsai, W.T., Wei, X., Chen, Y., Paul, R.: A robust testing framework for verifying web services by completeness and consistency analysis. In: Proceedings of the IEEE International Workshop. pp. 159–166. SOSE '05, IEEE Computer Society, Washington, DC, USA (2005)

[93] Tsai, W., Wei, X., Chen, Y., Xiao, B., Paul, R., Huang, H.: Developing and assuring trustworthy Web services. In the Proceedings of Autonomous Decentralized Systems (2005)

[94] Van Der Straeten, R.: Inconsistency Management in Model-driven Engineering. An Approach using Description Logics. Ph.D. thesis, Vrije Universiteit Brussel, Brussels, Belgium (September 2005)

[95] Varró, D., Pataricza, A.: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling 2(3), 187–210 (October 2003), `http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2003/sosym2003_vp.pdf`

[96] Vitus S. W. Lam, J.A.P.: Consistency checking of statechart diagrams of a class hierarchy. In: ECOOP. pp. 412–427 (2005)

[97] Vlist, E.V.D. (ed.): RELAX NG: A Simpler Schema Language for XML. O'Reilly, California, CA, USA (2003)

[98] W3: OWL 2 Document Overview (October 2009), available at http://www.w3.org/TR/owl2-overview.

[99] W3C: Extendable Markup Language XML Version 1.0 (September 2005), available at http://www.w3.org/TR/xml-id/.

[100] Walter, T., Parreiras, F., Staab, S.: An ontology-based framework for domain-specific modeling. Software and Systems Modeling pp. 1–26, http://dx.doi.org/10.1007/s10270-012-0249-9, 10.1007/s10270-012-0249-9

[101] Walter, T., Parreiras, F.S., Staab, S.: OntoDSL: An ontology-based framework for domain-specific languages. In: Schürr, A., Selic, B. (eds.) MoDELS. Lecture Notes in Computer Science, vol. 5795, pp. 408–422. Springer (2009)

[102] Wang, S., Jin, L., Jin, C.: Ontology Definition Metamodel based Consistency Checking of UML Models. pp. 1 –5 (may 2006)

[103] Wilke, C., Demuth, B.: UML is still inconsistent! How to improve OCL Constraints in the UML 2.3 Superstructure. ECEASST 44 (2011)

[104] Yeung, W.L.: Checking Consistency between UML Class and State Models Based on CSP and B. J. UCS 10(11), 1540–1559 (2004)

[105] Yin, Y., Yin, J., Li, Y., Deng, S.: Verifying consistency of web services behavior using type theory. In: Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE. pp. 1560–1567 (dec 2008)

[106] Zhao, Y., Pan, J.Z., Jekjantuk, N., Henriksson, J., Groner, G., Ren, Y.: Most project - definition of language hierarchy (2008), availible at https://www.most-project.eu/admin/xinha/plugins/ExtendedFileManager/images/Deliverables/MOST_Deliverable_D3.1.pdf

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms

168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners

Turku Centre *for* Computer Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www. tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Division for Natural Sciences and Technology
- Department of Information Technologies

ISBN 978-952-12-2985-5
ISSN 1239-1883

Ali Hanzala Khan

Consistency of UML Based Designs Using Ontology Reasoners