A Natural Language Processing Approach To Generate SBVR and OCL

By

Imran Sarwar Bajwa

A thesis submitted to

The University of Birmingham

for the degree of

DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham
June 2012

Abstract

The Object Constraint Language (OCL) is a declarative language. It is used to make well-defined models of the Unified Modeling Language (UML) through defining a set of constraints. However, it is a common knowledge that OCL is the least used language among the 13 UML languages. The main cause of less use of OCL is attributed to complex syntax of the language, overly expressive nature of OCL, and difficult interpretation of large OCL expressions. Since, a single OCL expression can be written in multiple possible ways, the expressive nature of OCL confuses OCL writers. The complexity of its syntax makes the writing of OCL code difficult. Complex syntax and descriptive nature of OCL result in very lengthy and complicated OCL expressions which are too complex to interpret for a user. Such issues make it difficult for one to write OCL constraints especially for the novice users.

A natural language based interface can be useful in making the process of writing OCL expressions easy and simple. However, the translation of natural language (NL) text to object constraint language (OCL) code is a challenging task on account of the informal nature of natural languages as various syntactic and semantic ambiguities make the process of NL translation to formal languages more complex. SBVR is the OMG's recent standard introduced to overcome the inherent ambiguity of natural languages. SBVR not only provides natural languages a formal abstract syntax representation but it is also close to OCL syntax as both languages (SBVR & OCL) are based on formal logic.

In this research, the major contribution is a novel approach that aims at presenting a method based on *natural language processing* and *model transformation* technology to improve OCL usability. The aim of the method is to produce a framework so that the user of UML tools can write constraints and pre/post conditions in English. The framework is useful in translating such English expressions to the equivalent OCL statements. The proposed approach is implemented in Java as an Eclipse plugin named the NL2OCL*via*SBVR. The tool generates OCL from NL constraints via SBVR and is a proof of this concept. The NL2OCL*via*SBVR allows software modelers and developers to generate well-formed OCL expressions that result in valid and precise UML models. An evaluation of the OCL constraints is also performed to test the performance of the tool. For this purpose, three famous case studies have been done using the NL2OCL*via*SBVR tool. The results of the case studies manifest that a natural language based approach to generate OCL constraints can not only help in significantly improving usability of OCL but also outperforms the most closely related techniques in terms of effectiveness and effort required in generating OCL.

The designed system NL2OCL*via*SBVR is always capable of producing the wrong analysis but that in such circumstances the produced formal representation is correct for a particular, valid and potentially correct interpretation and can be corrected by manual intervention.

KEYWORDS

Object Constraint Language, Semantics of Business Vocabulary and Rules, Natural Language Processing

Acknowledgments

First, I am extremely thankful to Allah Almighty for giving me the strength and fortitude to achieve this milestone.

I owe my deepest gratitude to my supervisors Dr. Mark Lee and Dr. Behzad Bordbar, for their enthusiasm, their encouragement, and their resolute dedication to the strangeness of my research. Where, I cannot forget the support of Dr. Mark Lee in understanding the field of Natural Language Processing (NLP), at the same time, I want to express my gratitude to Dr. Behzad for introducing me to Semantics of Business Vocabulary and Rules (SBVR) standard, Model Driven Architecture (MDA) standard and role of Object Constraint Language (OCL) in Unified Modeling Language (UML) based software modeling. I would also like to thank Dr. Peter Hancox and Dr. Rami Bahsoon for providing criticisms on the progress of the work during the thesis group meetings. I also thank to my father for proof reading my thesis and improving its English.

This work would not have been possible without the financial support of The Islamia University of Bahawalpur (IUB) and Higher Education Commission (HEC), Pakistan. I would also like to thank the School of Computer Science, University of Birmingham for providing me financial support during my research.

Last but not the least I would not have been standing at the finish line had it not been for the selfless love and prayers of my parents and wife. Especially, I would like to express gratitude to my father for proofreading and polishing English of my PhD thesis. I dedicate this thesis to my kids Muhammad Rafay Imran and Enaya Imran.

ACRONYMS

EMF	Eclipse Modelling Framework
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MOF	Meta Object Facility
NLP	Natural Language Processing
OCL	Object Constraint Language
OMG	Object Management Group
POS	Parts-of-Speech Tag
QUDV	Quantities, Units, Dimensions and Values
QVT	Query/View/Transformation
SBVR	Semantics and Business Vocabulary & Rules
SRL	Semantic Role Labeling
SiTra	Simple Transformer
UML	Unified Modeling Language
USE	UML-based Specification Environment Tool
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Published Work

1. Imran Sarwar Bajwa, Behzad Bordbar, Mark Lee. (2010). "OCL Constraints Generation from Natural Language Specification", in IEEE/ACM 14th International EDOC Conference 2010, Vitoria, Brazil, October 2010, pp:204-213

2. Imran Sarwar Bajwa, Mark Lee, Behzad Bordbar. (2011)."SBVR Business Rules Generation from Natural Language Specification", in AAAI Spring Symposium 2011 – Artificial Intelligence for Business Agility (AI4BA), San Francisco, USA, March 2011, pp:2-8.

3. Imran Sarwar Bajwa, Mark Lee. (2011). "Transformation Rules for Translating Business Rules to OCL Constraints", in ECMFA 2011 - 7th European Conference on Modelling Foundations and Applications, Birmingham, UK, June 2011, pp:132-143

4. Imran Sarwar Bajwa, Behzad Bordbar, Mark G. Lee (2011), “SBVR vs OCL: A Comparative Analysis of Standards”, in 14thIEEE International Multi-topic Conference (INMIC 2011), Dec 2011, Karachi, Pakistan, pp:261-266

5. Imran Sarwar Bajwa, Mark G. Lee, Behzad Bordbar. (2012). "Resolving Syntactic Ambiguities in NL Specification of Constraints using UML Class Model" in 13thInternational Conference on Computational Linguistics and Intelligent Text Processing (CICLing 2012), Delhi, India, , March 2012, pp:178-187

6. Imran Sarwar Bajwa, Mark Lee, Behzad Bordbar, (2012). “Semantic Analysis of Software Constraints”, The 25th International FLAIRS Conference, Florida, USA, May 2012, pp:8-13

7. Imran Sarwar Bajwa, Mark Lee, Behzad Bordbar, Ahsan Ali (2012). “Addressing Semantic Ambiguities in Natural Constraints”, The 25th International FLAIRS Conference, Florida, USA, May 2012, pp:262-267

8. Imran Sarwar Bajwa, Mark Lee, Behzad Bordbar. (2012). “Translating Natural Language Constraints to OCL”, Journal of King Saud University - Computer and Information Sciences, June 2012, 24(2): Elsevier

9. Imran Sarwar Bajwa, Behzad Bordbar, Kyriakos Anastasakis, Mark Lee (2012). “On a Chain of Transformations for Generating Alloy from NL Constraints”, 7th IEEE International Conference on Digital Information Management (ICDIM 2012), Macau, August 2012, pp:93-98

10. Imran Sarwar Bajwa, Behzad Bordbar, Mark Lee, Kyriakos Anastasakis (2012). “NL2Alloy: A Tool to Generate Alloy from NL Constraints”, JDIM, Dec 2012, (10)6: , pp:365-372

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

This chapter presents the overview of the research area addressed in this thesis. The research problem has also been described in detail along with the research motivations, the major research objectives and the published work.

1.1 Research Problem

In object-oriented software engineering, the Unified Modelling Language (UML) is used to visually represent software models. UML has been adopted as the de-facto standard for the design, modelling and documentation of software systems [Gogolla, 2007]. There are lots of tools which allow not only modelling and design but also support creation of code, reverse engineering, versioning [Engles, 2001]and much more. However, it is a well-known fact that the least used of all UML languages is OCL. This is often attributed to complex syntax of the language [Wahler, 2008].

The developer's ability to use OCL is very important. Faultless OCL constraints and pre/post conditions quite significantly improve the clarity of software models and make models more precise [Wahler, 2008]. Users have to translate manually the NL representations of the constraints to OCL syntax. The manual effort to create an OCL constraint usually results in inaccurate and inconsistent constraints specification [Gogolla, 2007] for many reasons. First of all, the OCL syntax is very hard to code [Wahler, 2009]. Secondly, OCL is a highly expressive language that results in multiple possible OCL expressions from a single NL representation. This multiplicity of OCL expressions confuses the user during manual creation [Cabot, 2009].

Thirdly, there are no means available for semantic verification of the OCL constraints so that it is not easy to decide if they mean exactly what they were written for.

Besides, constraints specification, OCL can be used for specifying models for analysis purposes as shown in the UML2Alloy project [Shah, 2008]. Improving the usability of OCL will also assist developers who are not experts in formal methods for producing specifications in automatic analysis.

The OCL usability issue becomes more critical on account of the absence of a tool that is capable of automatically creating OCL constraints from NL specification. This is critical because the existing OCL tools, e.g., Dresden OCL Toolkit [Demuth, 2009], IBM OCL Parser [IBM, 2009], USE [Gogolla, 2007], ArgoUML [Rompaey, 2007], Cybernetic OCL Compiler [Emine, 2008] are just limited to syntax verification and type checking.

There is need of an approach that allows development of tools and techniques that provide assistance in writing OCL. Wahler has tried to tackle this problem using a template based approach [Wahler, 2008]. However, this thesis is adopting a radically new approach by bringing together two main domains of computer science: model transformation and Natural Language Processing (NLP). Using natural languages and transformation to OCL seems like an intuitive approach. However, we adopt a systematic way to use SBVR to restrict the domain of NL text and generate OCL code from the SBVR representation. The OCL usability can be increased through automatically generating accurate and consistent code for OCL constraints.

Modelling Software is a process in software engineering, in which information or knowledge is represented in a structure. Since the emergence of object-oriented software engineering, visual models are very common to represents software schemas.In the context of the above described scenario, the proposed research addresses the following four key scientific questions:

- How can the natural language text be analysed to understand meaning and extract the required knowledge from the text?
- How can SBVR be useful in making natural language syntactically restricted and semantically formulated?
- How can the informal representation i.e. natural language text be transformed into formal representation i.e. Object Constraint Language.

- What is the effect of Natural Language based software modeling in the software design process?

1.2 Scope of the Research

The contributions expected from this work are mainly centered in two areas: (1) NLP and (2) model-driven Software Engineering. The conversion of natural language text into OCL constraints requires syntactic and semantic knowledge. Here, a translation process is required that is robustly capable of dealing with natural language ambiguity and vagueness. The development of new methods for syntactic and semantic analysis is one of the main contributions of this research. Methodologies like Model Transformation and Markov Logics have been proposed to develop these new NLP methods. There are also other contributions in the NLP area, which is the study and modelization of language pragmatics and ambiguity resolution of NL.

The researcher is aware of the fact that a NLP based solution cannot be 100% accurate due to informal nature of NL. The researcher has made pair usage of NL and automated generated SBVR. These can help in resolving ambiguities and clarifying vagueness by pointing them out. However, this will not be a 100% solution either and the researcher is aware of it. The designed system NL2OCL*via*SBVR always tends to produce the wrong analysis but in such circumstances the produced formal representation is correct for a particular, valid and potentially correct interpretation and can be corrected by manual intervention.

The other main contribution of this research is the development of new ways of helping software engineers to develop software models, making their work more productive and efficient. The most familiar language for humans is natural language. So a tool that could help translating what the software engineer wants into what the machine understands is a valuable tool for software companies.

1.3 Research Motivation

The use of natural langauge processing in software enigneering and specifically the translation of natural languages to formal specifications is not a brand new proposal. Automated translation of natural language (e.g. English) specifications to formal specification (e.g., E-R models, object oriented analysis, UML models, high-level languages code, database queries) have already been achieved. Overmyer [2001] has presented conceptual modeling of natural language (NL) requirement specification and generation of Entity-Relationship models. Further, Osborne [1996], Mich [1996], Delisle [1999] have presented automated processing of NL based requirement specifications. Natural language processing was incorporated to automate object oriented analysis and modeling of NL based software requirement specifications is presented by Juristo [2000], Brown [2002], Perez-Gonzalez [2002], Cockburn [2002], Li [2005]. Natural language based requirement engineering and generation of UML models has also been presented by a few researchers. Rolland [1992] used a linguistic approach to process natural languages statements and generation of conceptual specification.

A few Natural language based tools have also been introduced to generate formal specifications, e.g., NL-OOPS [Mich, 1996], LIDA [Overmyer, 2001], GOOAL [Perez, 2002], CM-Builder [Hermain, 2003], Rebuilder [Gomes, 2006], UML-Generator [Bajwa, 2008], R-TOOL [Vinay, 2009], etc. None of these tools provides support for translating NL specification to OCL constraints. The aim of the proposed research is to extend this work by automating the process of annotating UML models with OCL constraints. An English based user-interface to create OCL constraints for UML models can make not only the OCL more adaptable but also assist in automatic analysis of object-oriented models.

The main idea of the research is to propose the idea of writing constraints specification for a UML model in English and then transforming the English specification to OCL syntax. The automatic transformation of English specification to OCL syntax involves semantic analysis of English specification and its mapping with OCL constraints. The direct transformation of English specification to OCL constraints is difficult due to the informal syntax and inconsistent semantics of English. The researcher proposes the use of the Semantics of Business Vocabulary and Rules (SBVR) [OMG, 2008] to reduce complexity involved in the processing of natural languages as SBVR has already been used to represent Business designs [Raj, 2009]. SBVR is used as an intermediate language for English to OCL transformation as SBVR is a formal representation of English and close to OCL syntax. A methodology is designed for the automatic transformation of English to SBVR and SBVR to OCL constraints that will be based on a set of transformation rules. During English to SBVR transformation, SBVR specification is also mapped with the given UML model to semantically verify the defined constraints before it is finally transformed to OCL syntax. The proposed automated transformation will not only soften the process of creating OCL but also enhance the adoptability of OCL by providing automatic mechanism of semantic verification with a UML model and semantic validation with user input. The researcher also aims at addressing OCL usability problem in this thesis.

1.4Published Work

During three years of PhD research, the major contributions by the researcher were published in reputed conferences and journals. An overview of the work published during PhD research is given below.

The first paper was published in EDOC 2010, held in Brazil. The paper was based on the main idea of research that NL constraints can be automatically translated to OCL constraints [Bajwa, 2010]. In this paper, the researcher presented how SBVR can play a useful role in translation of NL specification of constraints to OCL invariants and OCL pre/post conditions.

In 2011, a paper was published in AAAI spring symposium, held in USA that was addressing the SBVR rules aspect of this research [Bajwa, 2011a]. In this research, the researcher is generating SBVR rules from NL constraints and then mapping such SBVR rules to OCL. In this paper, the researcher presented an automated approach that can process NL constraints to extract SBVR vocabulary and generate a complete SBVR rule. Another paper was published in ECMFA 2011 conference, held in UK. This paper presented the set of transformation rules used to transform a SBVR rule to an OCL constraint [Bajwa, 2011b]. In 2011, another paper by the researcher was published in 14th IEEE INMIC, held in Pakistan. This paper presented the key findings during a study of SBVR and OCL standards. A comparison of both standards is also presented in this paper to highlight various similarities and differences in both standards [Bajwa, 2011c]. This study helped the researcher in SBVR to OCL transformation.

A few more papers were published in 2012. A paper was published in CICLing 2012 (held in India) to present a set of identified syntactic ambiguities in NL specification of constraints and an approach to resolve such syntactic ambiguities [Bajwa, 2012a]. The presented approach was using metadata of UML Class Model to resolve identified syntactic ambiguities. Two more papers were published in 25th edition of FLAIRS, held in USA: the focus of one paper was the semantic Analysis of Software Constraints [Bajwa, 2012b] while the other paper highlights the identified set of semantic ambiguities in NL constraints. An approach is also presented in the paper to address the identified set of semantic ambiguities in NL constraints [Bajwa, 2012c]. The present approach helps in improving semantic role labeling and quantifier scope resolution in terms of accuracy. Another paper presenting the results of Royal and Loyal modal case study was presented in Journal of King Saud University (JKSU) - Computer and Information Sciences [Bajwa, 2012d]. This paper highlights that the researcher's NL-based approach is more accurate than the pattern based approach [Wahler, 2008]. An extension of this work is accepted in IEEE ICDIM 2012 (held in Macau), that focuses on generation of Alloy code from NL constraints [Bajwa, 2011e]. This work is also used in qualitative analysis of our approach presented in Chapter 7, Section 7.4 as the researcher generates Alloy of OCL (generated by our NL approach) and if Alloy is generated correctly, it means that OCL is also correct. The details of this evaluation are given in Section 7.4 of this thesis.

1.5 Thesis Contribution

In Section 1.1, a set of questions are raised that are core of this research thesis. To answer these questions various investigations were performed and new results were achieved. We have provided the details of each answer in various chapters of this thesis.

1.5.1How the informal representation by natural language text can be transformed into a formal representation using the Object Constraint Language.

To transform natural language representation of constraints to formal constraints, the NL2OCL approach is presented in this thesis. The NL2OCL approach is a fundament thrust of the presented research and is explained in Chapter 3, Section 3.3. The NL2OCL approach works in two phases: the NL to SBVR translation, explained in Chapter 4 and SBVR to OCL transformation, explained in Chapter 5.

1.5.2 How a natural language text can be analysed to understand its meaning and extract the required knowledge from the text?

To extract meaningof NL constraints, we had to deal with various syntactic and semantic ambiguities. The approach used to deal with syntactic ambiguity is explained in Chapter 4, Section 4.2,and while as the approach used to address the semantic ambiguity is explained in Chapter 4, Section 4.3.

1.5.3 How SBVR can be useful in making natural languages syntactically restricted and semantically formulated.

In the NL2OCL approach, SBVR plays a key role as the researcher uses SBVR based intermediate representation in NL to OCL transformation.A detail of such intermediate representation is given in Chapter 4, Section 4.3. A set of model transformation rules to transform a SBVR rule to an OCL expression are explained in Chapter 5, Section 5.3.

1.5.4How well the NL approach works for generating formal representation such as OCL?

To evaluate the performance in terms of correctness of NL2OCL approach, three case studies are done. The details of these case studies with results are discussed in Chapter 7.

1.6 Thesis Organization

The remaining thesis is structured into a set of chapters. Each chapter describes a distinct part of research. Brief overview of each chapter is given below:

Chapter 2 presents the preliminaries of the presented research. In this chapter we provide an overview of the fundamental concepts, such as OCL, SBVR, NLP, MDA and model transformations. The work related to the field of NL-based automated software modelling and automated transformations for SBVR and OCL is presented in the second part of the chapter to highlight the significance of the presented research.

Chapter 3 presents the thesis statement of this research. A set of hypotheses are also highlighted in the chapter which the researcher has tried to prove in the rest of the thesis. Moreover, a sketch of the NL2OCL approach has also been provided.

Chapter 4 presents the first part of the NL2OCL approach that deals with the processing of natural language constraints and generate a SBVR based logical representation. The core of this chapter is syntax and semantic analysis of NL constraints. The researcher also presents major innovations and contributions of the research in the same chapter.

Chapter 5 presents the approach used to transform SBVR business rules to OCL constraints. This chapter provides the details of SiTra based model transformation and the transformation rules involved in SBVR to OCL transformation.

Chapter 6 highlights the key features and implementation details of the NL2OCLviaSBVR tool. Moreover, the architecture of the tool as well as the used libraries in transformation of NL to OCL has also been discussed.

Chapter 7 presents evaluation criteria and also provides the details of three case studies solved to validate the performance of the tool discussed in Chapter 6. The results of all three cases studies have also been discussed under the headings of quantitative and qualitative evaluation. The chapter also presents a set of limitations of the presented approach.

Chapter 8 discusses the key contribution in thesis. In addition, we conclude the presented work by highlighting the overall contributions of the research presented in this thesis. The chapter ends with pointed out areas for future research.

Chapter 9 presents the concluding remarks of the thesis.

CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter presents background of the research and a brief introduction to preliminary concepts which have been used later on in this thesis. The second half of the chapter presents the work related to translation of a Natural Language (NL) to formal and software engineering languages. Automated transformation and tools for OCL and SBVR are also discussed at the end of the chapter.

2.1 Preliminaries

The standards like UML, OCL and SBVR are involved in modern information systems to ensure quality and correctness. UML is a de-facto standard. UML provides a graphical notation to represent software conceptual schema or models. Here, OCL based textual constraints are used to restrict UML based conceptual schemas. Similarly, SBVR provides a formal representation for software and business requirements. On the other hand, SBVR based requirements are not only easy to read for users but also simple for machine-process.

In this section, an introduction is presented of various concepts that are involved in the present research and have been used throughout the thesis. First, an introduction of the Object Constraint Language (OCL) along with its role in UML-based modelling is presented. Then the basics of SBVR and role of SBVR in modern software Engineering are discussed. Afterwards, the researcher describes fundamentals of Natural Language Processing (NLP) and elaborates typical phases involved in NLP. Finally, the concept of Model Driven Architecture (MDA) is elaborated to highlight the basics of model transformation with Simple Transformers (*SiT*ra) [Akehust, 2007] that plays a key role in this research.

Before going into details of OCL and SBVR, it is pertinent to define meta-models. A metamodel [OMG, 2010] is a key part of standards such as OCL and SBVR and metamodel is a simplified descriptive model (blueprint) of another descriptive model.

2.1.1 Object Constraint Language (OCL)

The Object Constraint Language (OCL) is an adopted standard of the Object Management Group (OMG) and typically used to annotate UML models with constraints. Constraints specified in OCL help to restrict UML models [Cabot, 2009] but they also increase maturity level of a UML model [Wahler, 2008]. OCL is a declarative language that can also be used with the Meta-Object Facility (MOF) standard. Recently, OMG proposed the role of OCL in the Queries/Views/Transformations (QVT) specification to be used for model transformations [OMG, 2010]. OCL is a side-effect free language. That means OCL does not introduce any new object in a UML model but completes the meanings of the existing objects. In addition, an OCL constraint always conforms to the OCL meta-model. In the UML standards, OCL is also used for expressing constraints to satisfy the well-definedness criteria.

In this thesis, we target the ability of OCL to write invariants for a UML class model. The following sections bring out a brief overview of OCL syntax with examples and highlight various features of OCL.

A. OCL Syntax

To define the basic structure of an OCL expression, OCL syntax is given in OCL 2.0 [OMG, 2006] document. A simplistic view of OCL meta-model is shown in Figure 2.1. Abstract syntax typically deals with the grammar and structure of the OCL statements. OCL abstract syntax is further defined into OCL types and OCL expressions. Common OCL types are data types, collection types, message types, etc. While, OCL expressions can be call expression, if expression, literal expression, variable expression [OMG, 2006], etc. We have used a selected set of OCL abstract syntax for implementation that does not include OCL types.

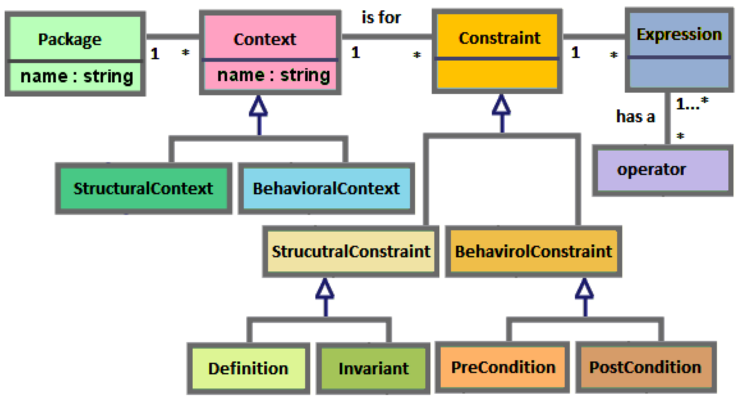


Figure 2.1:Generic view of OCL metamodel

The focus of the present research is the OCL expressions. Figure 2.2 shows the abstract syntax meta-model [OMG, 2006] of OCL expressions. It is shown that there are four possible types of an OCL expression such as `CallExpression`, `IfExpression`, `LiteralExpression` and `VariableExpression`. Here, a `CallExpression` can be a `LoopExpression`, and `OperationCallExpresion`. However, `IteratorExpression` can be a type of `LoopExpression`. Moreover, in the OCL expression meta-model, there is a `VariableDeclaration` object for a`VariableExpression`.

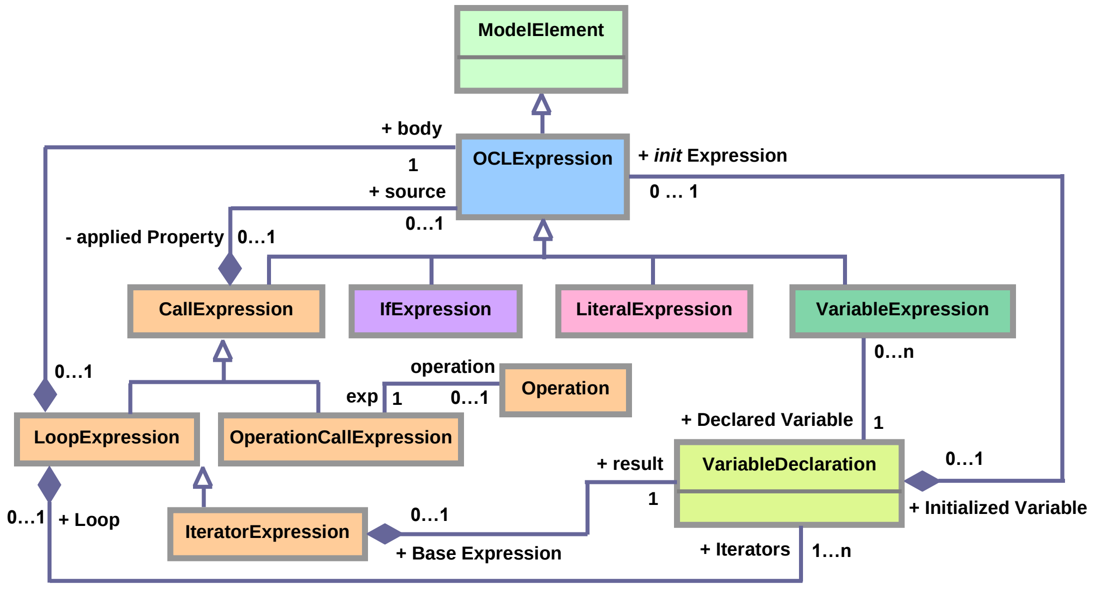


Figure 2.2: Elements of selected OCL Expression metamodel

The syntax of a typical OCL expression is composed of the four components as explained below:

i. Context: A context [OMG, 2006] in OCL constraint specifies the scope of that constraint as shown in Figure 2.3. The OCL context limits a world of an expression in which it is valid. Context can be different for an invariant and pre/post condition such as for an invariant, a context is a name of a class whereas for a pre/post condition a context is combination of a class and targeted operation of that particular class. Typical context is a class name which it belongs to. Keyword "`self`" is used to refer to a context in an OCL constraint.

```
- context Person                          -- for invariant
- context Person::setAge(newAge:int)      -- for pre/post condition
```

Figure 2.3: An OCL constraint with a context

ii. Property: An OCL property [OMG, 2006] represents an attributes or operation of a class. The "." operator is used to specify such properties. A possible OCL property can be an attribute of a class, a method of a class or an association between two classes. An example of OCL property is shown in Figure 2.4.

```
- person.age  or  self.age-- an attribute as OCL property
- person.isAdult()  or  self.isAdult()-- a method as OCL property
```

Figure 2.4: OCL examples with properties

iii. Operation: An OCL operation [OMG, 2006] manipulates or qualifies a property on an attribute or set of attributes related to a class (see Figure 2.5). Collection types are also a part of OCL typically used to handle set of attributes. To perform various functions, various operations are available such as to return number of elements operation `size()` is used. Similarly, operation `exists()` returns that an object exists in a domain or not. Examples of OCL operations are shown in Figure 2.5.

```
- self.items->size()            -- operation returns number of items
- self.order->exists()          -- operation returns True if objects exists
```

Figure 2.5: OCL example with an operation

iv. Keyword: The OCL keywords [OMG, 2006] are also important parts of any OCL constraint. Typically, used OCL keywords are `if`, `then`, `else`, `and`, `or`, `not`, `implies`, etc. Some keywords like `and`, `or`, `not`, etc. are used to represent the conditional expressions in a constraint, while some other keywords like `inv`, `result`, `pre`, `post`, `def`, etc. are used to represent various sections of an OCL constraint.

B. Types of OCL Constraints

An OCL constraint defines a restriction on state or behaviour of an entity in a UML model. A restriction on the state of an entity is represented using an OCL invariant, while behaviour of an entity is expressed using OCL pre/post condition. The OCL constraint defines a Boolean expression that always results in True or False. If the constraint results true, the system is in valid state. There are three types of OCL constraints:

i. Invariants: An OCL invariant [OMG, 2006] is a constraint that must always be convened by all instancesof the class. An invariant is a condition that has to be TRUE always. Invariants typically represent structural information and used to restrict an entity in a model. An example of an invariant is shown in Figure 2.6.

```
context Person
inv: self.age<18 implies self.car -> forAll(v | not v.oclIsKindOf(Car))
```

Figure 2.6: An example of OCL Invariant

ii. Precondition:A precondition [OMG, 2006] is a restriction on a method of a class. A precondition is a constraint that should be TRUE always before the execution of a method starts. Preconditions typically represent behavioural information. An example of a precondition is shown in Figure 2.7.

```
context Person::setAge(newAge:int)
pre: newAge >= 0
```

Figure 2.7: An example of OCL Precondition

iii. Postcondition:An OCL postcondition [OMG, 2006] is a constraint that should be TRUE always after the execution of a method has finished. Similar to OCL precondition, an OCL postcondition also represents behavioural information. An example of a postcondition is shown in Figure 2.8.

```
context Person::setAge(newAge:int)
post: self.age=newAge
```

Figure 2.8: OCL Postcondition

C. Why Writing OCL is Difficult?

During the literature review, the researcher found various reasons which make it difficult to write OCL, especially for the novice users. The researcher has compiled the reasons identified by various scientists into following three dimensions:

i. OCL Syntax is Complex: In a typical software modelling scenario, a modeller has to manually process natural language constraints by extracting various OCL elements from NL constraints and then writing those elements in OCL syntax. However, writing OCL code manually is a difficult and cumbersome job due to many reasons. One of the reasons is complex nature of OCL syntax [Gogolla, 2007] that ultimately results in very lengthy and complex OCL expressions which are difficult to write manually [Wahler, 2008]. Moreover, various issues have been identified that are source of ambiguity in OCL postconditions [Cabot, 2006]. Such issues confuse a user in writing OCL postconditions. Another feature of OCL, that makes it difficult to write OCL, is expressive nature of OCL. Owing to expressive nature of OCL, there can be multiple possible ways to write an expression in an OCL constraint [Cabot, 2007] and this feature of OCL also confuses a user and makes it difficult to write an OCL constraint. On the basis of these facts, we can conclude that writing OCL is difficult especially for the novice users.

ii. OCL is too Implementation Oriented: Natural Rule Language (NRL) is another initiative for providing a user-friendly alternative to languages like OCL, XSLT, etc. [Nentwich and James, 2010]. It is identified by NRL community that OCL is too implementation-oriented and not well-suited to conceptual modelling [Vaziri, 1999]. NRL community also presents an English-like syntax based language CLiX to specify constraints for XML. Later on, CLiX was extended in 2006 to replace OCL in an environment as OCL was found inappropriate and difficult to write specifically for novice users [NRL Community, 2005].

iii. Declarative Nature of OCL:OCL is a declarative language. OCL constraints describe what it wants to accomplish. While writing OCL a user's focuses is to write declarative statements to show a relationship between various parts of an OCL constraint [Correa, 2007]. The declarative nature of OCL makes it difficult to understand the use and application of various OCL expressions such as navigation expression. Actually, it is difficult to decide that when to use navigation and when not to do. Moreover, it is also complex to identify a left hand side for a right hand side of a navigation expression. Typically, '→' is used in a OCL navigation expression to express a relationship between two elements.

A complex example of OCL constraint for the following NL specification is shown in Figure 2.9. This constraint exhibits most of the issues which make OCL difficult to write. First of all, there are three possible OCL representations for the NL constraint discussed in [Cabot, 2007]. Secondly, this constraint involves three navigation expressions, as well. The NL representation of this constraint is also given in [ibid] that is "The maximum salary of a junior employee those with an age lower than 25 cannot earn more than the maximum junior salary value defined for their department" [Cabot, 2007].

```
context  Department
inv:  MaxSalary: Department.allInstances() -> forAll(d|not d.employee ->
     select (e|e.age < 25) -> exists (e|a.salary > d.maxJuniorSal))
```

Figure 2.9:A complex OCL constraint

Similarly, a more complex example of an OCL constraint is given in [Kleppe and Warmer, 2003] that is shown in Figure 2.10: "In the `enroll()` operation of `LoyaltyProgram`, the postcondition specifies that there is now one more customer than before and that the new customer's loyalty account has no points and no transactions."

```
context LoyaltyProgram::enroll(int c:Customer)
pre:   not customer -> includes(c)
post:  customer = customer @ pre -> including(c)
post:  membership -> select(customer=c) -> forAll(
      loyaltyAccount -> notEmpty() and
      loyaltyAccount.points = 0 and
      loyaltyAccount.transations -> isEmpty)
```

Figure 2.10: A more complex OCL constraint

2.1.2Semantic Business Vocabulary and Rules (SBVR)

Semantic Business Vocabulary and Rules (SBVR) [OMG, 2008] is a recently introduced standard by OMG. The latest version SBVR v1.0 was introduced in January 2008. SBVR can be used to capture specifications in English and represent them in formal logic so that they can be machine-processed. SBVR can be effective in formal representation of information in multiple dimensions [Linehan, 2008], i.e. the production of the business vocabulary and rules, development for multilingual support, support for format interchange capabilities, formalizing syntactic and semantic structures, etc. An example of SBVR rule is "It is obligatory that nationality of a **customer** *should be* British".

SBVR has two major elements: SBVR business vocabulary and SBVR business rules. A brief description of both elements of SBVR is given below.

A. SBVR Vocabulary

SBVR business vocabulary, also called SBVR vocabulary, consists of the specialized terms and concepts typically incorporated in the definition of a business domain in a particular organization [OMG, 2008]. There are various types of SBVR concepts. However, we are interested in three basic types of SBVR business vocabulary: Object Type, Individual Concept, and Fact Types. Figure 2.11 shows the SBVR meaning meta-model that highlights various types of SBVR concepts.

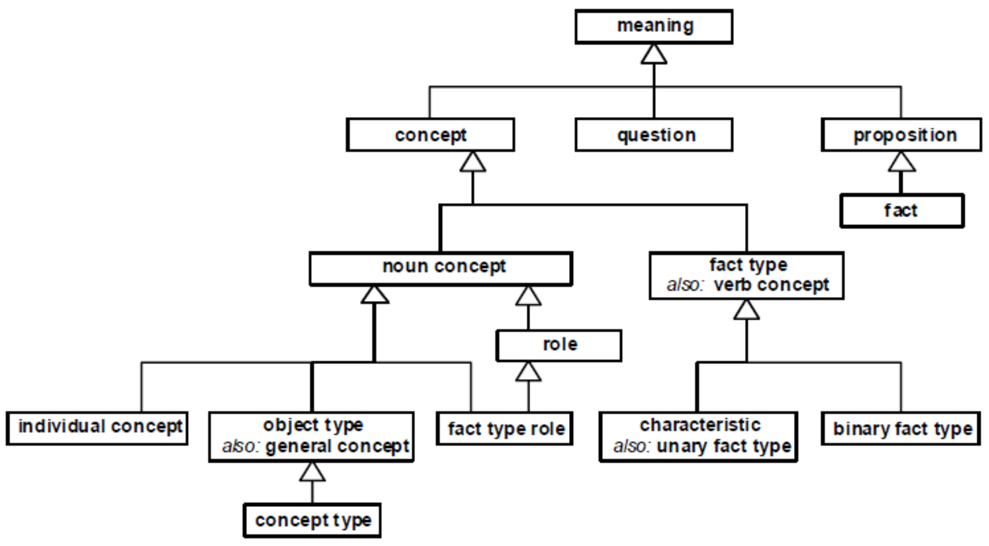


Figure 2.11: SBVR meaning metamodel [OMG, 2008]

Following is the overview of the used three types of SBVR concepts:

i. Object Type: In SBVR 1.0, an Object Type is a Noun Concept that is also called a General Concept. An Object Type is a Noun Concept that categorizes things on the basis of their common properties [OMG, 2008: Section 8.1.1]. Typically, in natural languages, the common nouns can be represented as Object Types. In the example discussed above, '**customer'** is an Object Type. In UML class models, an Object Type can be mapped to a UML class. Simple examples of an Object Type can be '**account'**, '**customer'**, '**student'**, '**book'**, etc.

ii. Individual Concept: In SBVR 1.0, an Individual Concept corresponds to only one object [OMG, 2008: Section 8.1.1]. In the example discussed above, 'British' is an Individual Concept. However, an Individual Concept cannot be an Object Type or Fact Type role. In English, proper nouns or quantified nouns are classified as Individual Concepts for example 'Silver Account', 'London', 'Commercial Bank' etc. In UML class models, an Individual Concept can be mapped to an object.

iii. Fact Type: In SBVR, a Fact Type is based on a verb phrase that involves one or more Noun Concepts and whose instances are all actualities [OMG, 2008: Section 8.1.1]. In SBVR 1.0, a Fact Type is also called a Verb Concept. In the example discussed above, '**customer** *should be* British' is a Fact Type. A role of the Fact Type is one point of involvement of something in that instance for each instance of a Fact Type. A Fact Type can be of many various types. We have used following types of Fact Types in our research:

a. Characteristic: A Characteristic is a type of Fact Type. A Characteristic always has exactly one role, but it can be defined using Fact Types having multiple roles. Basically, a Characteristic is is-property-of Fact Type. In the example discussed above, nationality is a Characteristic. In this example, nationality is property of Object Type **customer**. In English, a Characteristic can be an adjective or an associative noun. In UML class models, a Characteristic can be mapped to an attribute of a class.

b. Binary Fact Type: A Binary Fact Type consists of exactly two roles those can be General Concept or Individual Concept. Typically, in a Binary Fact Type there is also a verb phrase that relates two roles. In example, "A **customer** *opens* an **account**", **customer** and **account** are two roles and *opens* is a verb phrase that relates both roles with each other. In English, combination of subject, verb and object form a Binary Fact Type. In UML class models, a Binary Fact Type can be mapped to an association in two classes. In SBVR 1.0, associative Fact Type [OMG, 2008: section 11.1.5.1] is a concept very close to binary fact type.

c. Partitive Fact Type: A Partitive Fact Type [OMG, 2008: Section 11.1.5.1] is a type of Fact Type that represents a composition of a given whole. In English, constructs such as "is-part-of", "included-in" or "belong-to" are used to represent a Partitive Fact Type. For example, "Edgbaston is included in Birmingham". Here 'Birmingham' is composed of many areas where "Edgbaston" is one of them. In UML class models, a Partitive Fact Type can be mapped to a UML aggregation.

d. Categorization Fact Type: A categorization Fact Type [OMG, 2008: Section 11.1.5.2] represents a particular concept that is a type or category of another concept. Here, each instance of the fact type is an actuality. In English, a categorization fact type is identified by various constructs such as "is-category-of" or "is-type-of", "is-kind-of". An example of a categorization fact type can be "Gold account is a special account". Here 'Gold account' is category of 'special account'. In UML class models, a categorization fact type can be mapped to UML generalizations.

B. SBVR Business Rules

SBVR business rules or SBVR rules are used to represent the text logic. A SBVR rule can be formally defined as "an element of guidance that introduces an obligation or necessity" [OMG, 2008]. A SBVR rule is typically based on various SBVR vocabulary items and represents relationship among these SBVR vocabulary items used in the SBVR rule. A typical SBVR rule can represent a piece of structural or behavioural information. On the basis of information represented in a SBVR rule, the SBVR rules can be classified into two types; definitional rule and behavioral rule:

i. Definitional Rule: A definitional rule is used to define an organization's setup and it represents the structure of the organization. A definitional rule is also called structural rule and typically a definitional rule is a claim of necessity [OMG, 2008: Section 12.1.2]. An example of a definitional rule is "**It is necessary** that **each** <u>customer</u> *has* **at least** one <u>bank account</u>", and it explains that a customer should have at least one account, though she can have more than one account.

ii. Behavioural Rule: A behavioural rule expresses the conduct or behaviour of an entity. A behavioural rule is also called an operative rule. Formally, a behavioural rule is a claim of obligation [OMG, 2008: Section 12.1.2]. For example, in a behavioural rule "**It is obligatory** that **each** customer *can withdraw* **at most** GBP200 per day", the behaviour of a customer's account is explained by saying that a customer cannot withdraw more than GBP200 from his account.

C. Semantic Formulation

In SBVR 1.0, logical formulations are used to semantically formulate the SBVR rules. A semantic formulation that shapes a proposition is called a logical formulation [OMG, 2008: Section 9.2]. There are five semantic formulations given in SBVR 1.0. A selective subset of SBVR meta-model including various semantic formulations is shown in Figure 2.12.

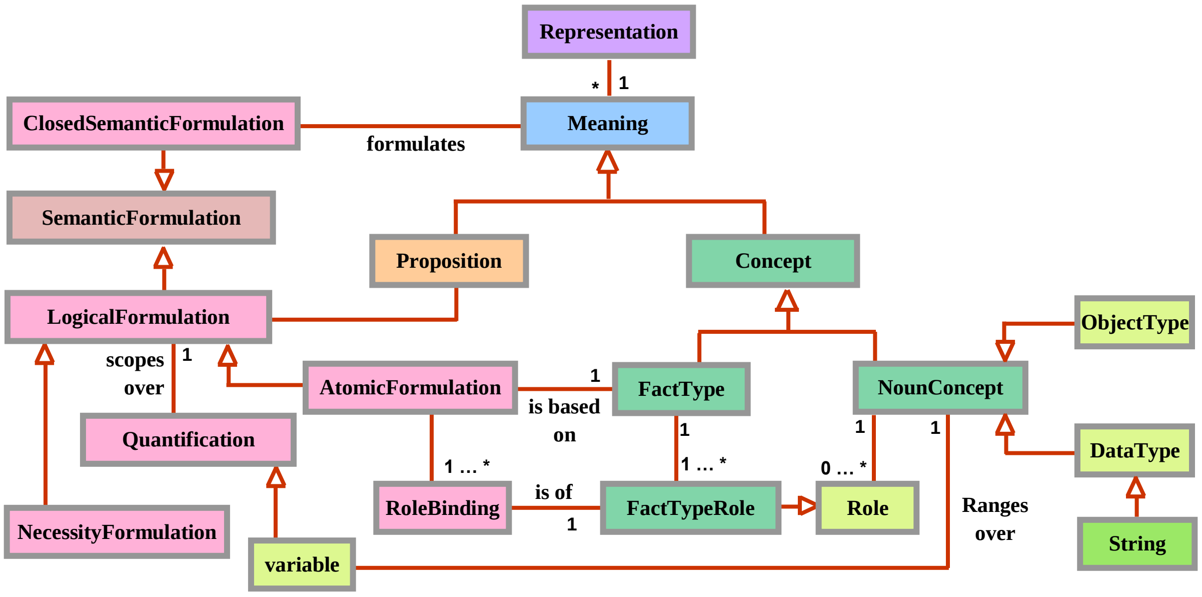


Figure 2.12: Elements of selected SBVR metamodel

The researcher is interested in three semantic formulations (such as Modal Formulation, Logical Formulation and Quantification) considering the scope of this research. Following is the brief description of these three SBVR semantic formulations:

i. Modal Formulation: Modal formulations [OMG, 2008: Section 9.2.4] are logical formulations that are used to specify meanings of the other logical formulations. There are four basic types of modal formulations.

a. Necessity Formulation: If a Logical Formulation is true in all possible worlds, it is represented as necessity formulation. In English, a sentence having words like 'need', 'may' and 'might' can be mapped to necessity formulation. In SBVR 1.0, a necessity formulation is represented using phrase "**It is necessary** that" [ibid].

b. Obligation Formulation: If a Logical Formulation is true in all acceptable worlds, it is represented as obligation formulation. In English, a sentence having words like 'should', 'must', 'ought', and 'have to' can be mapped to obligation formulation. In SBVR 1.0, an obligation formulation is represented using phrase "**It is obligatory** that" [ibid].

c. Permissibility Formulation: If a Logical Formulation is true in some acceptable worlds, it is represented as permissibility formulation. In English, a sentence having words like 'is' can be mapped to permissibility formulation. In SBVR 1.0, a permissibility formulation is represented using phrase "**It is permitted** that" [ibid].

d. Possibility Formulation: If a Logical Formulation is true in some possible worlds, it is represented as possibility formulation. In English, a sentence having words like 'would', 'can' and 'could' can be mapped to possibility formulation. In SBVR 1.0, a possibility formulation is represented using phrase "**It is possibility** that" [ibid].

ii. Logical Operations: Logical operations are used to combine one or more expressions, known as logical operand to produce complex Boolean expressions [OMG, 2008: Section 9.2.5]. We have incorporated these logical operations to map NL phrases to more complex logical expression. We are currently supporting the following six types of the logical expressions which are defined in SBVR v1.0 document [ibid]:

a. Conjunction: In SBVR, a conjunction is a binary logical operation and it is used to formulate the meaning of a logical decision of two operands that each operand is true [ibid]. A conjunction can be represented, i.e., `p AND q`.

b. Disjunction: In SBVR, a disjunction is a binary logical operation and it helps to formulate the meaning of a logical decision of two operands that at least one operand between two operands is true [ibid]. A conjunction can be represented, i.e., `p OR q`.

c. Equivalence: Equivalence in SBVR is another binary logical operation and it comprehends a logical decision that among two operands, the first operand is equal to the second operand [ibid]. An equivalence can be represented, i.e., `p is equal to q` or `p is q`.

d. Implication: In SBVR, an implication is a binary logical operation that is used to formulate the meaning of a logical decision of two operands that second operand is true if first operand is true [ibid]. An equivalence can be represented, i.e., `if p then q`.

e. Negation: In SBVR, negation is a unary operation for logical decision of one operand that formulates the meaning that the operand is false, i.e., `NOT p` [ibid].

iii. Quantification: Quantification is a logical formulation that uses a variable to specify the scope of a concept [OMG, 2008: Section 9.2.5]. Six basic types of quantifications have been defined in SBVR 1.0. Quantification types are briefly described below:

a. Universal Quantification: A universal quantification in SBVR 1.0 document is defined as a reference for each element in a domain [ibid], e.g., "each item".

b. Existential Quantification: In SBVR 1.0, an existential quantification represents minimum cardinality one represented by a thing [ibid], e.g., "at least one item".

c. At most-n-Quantification: At most-n-quantification shows maximum cardinality represented by number n represented by a thing [ibid], e.g., "at most 5 items".

d. At least-n-Quantification: At least-n-quantification shows a minimum cardinality represented by number n represented by a thing [ibid], e.g., "at least 3 items".

e. Numeric Range Quantification: A numeric range quantification in SBVR exhibits both minimum and maximum cardinality represented by a thing [ibid], e.g., "3 to 5 items".

f. Exactly n Quantification: This quantification shows the exact cardinality is represented as exactly as n quantification represented by a thing [ibid], e.g., "4 items".

Here two quantifications "exactly one Quantification" and "at most one quantification" are not mentioned as we deal them under "exactly-*n* Quantification" and "at most-*n* quantification", respectively. In the researcher's approach, all these logical formulations are employed to transform English text to a SBVR rule. For further details of SBVR the reader is referred to [OMG, 2008].

D. SBVR Notation

In SBVR 1.0 document [OMG, 2008], the Structured English is proposed, in Annex C, as a possible notation for the SBVR rules. The Structured English provides a standardized representation for formalizing the syntax of natural language representation [Kliener, 2009]. In this thesis, the researcher has used the following Structured English specification:

- <u>Object Type</u> is underlined e.g. <u>customer</u>
- *Verb phrase* is italicized e.g. *should be*
- **Keyword** is given in bold frame i.e. SBVR keywords e.g. **each**, **at least**, **at most**, etc.
- <u>*Individuals Concept*</u> is underlined and italicized e.g. <u>*London*</u>.

E. SBVR Rules vs. OCL Constraints

In business modeling, a business rule defines or constrains one aspect of business that aims to emphasize on the structure or behavior of the business [Ambler, 2003]. A comparison of SBVR rule and OCL constraints is presented in [Bajwa, 2011c]. Here, the researcher presents a few commonalities in a SBVR rule and OCL constraint. The different features of SBVR an OCL are not discussed here. For detail, we recommend the reader [Bajwa, 2011c].

i. Rules vs. Expression: The Rules [OMG, 2008: Section 12.1.2] in SBVR represent the specifications or the meanings of business constraints. Similar to Rules in SBVR, there are Expressions [OMG, 2010: Section 7.3] in OCL that make up a basic OCL constraint. Similarly, SBVR rules can be of two types: structural rules and behavioural rules. Similarly, OCL expressions are also of two types: structural constraints (such as invariant) and behavioural constraints (such as precondition or postcondition). Relation of a SBVR rules with OCL expressions can be explained as per two dimensions given below:

ii. Structural Rule vs. Invariant: The SBVR structural rules [OMG, 2008: Section 12.1.2] represent the structure of a business models and their underlying entities. The SBVR structural rules supplement definitions by using conditions and restrictions. Similar to SBVR structural rules, in OCL invariants [OMG, 2010: Section 7.3.3] are used to represent a structural constraint. OCL invariants typically specify structural information of UML models.

iii. Behavioural Rule vs. Pre/Post Condition: The SBVR behavioural rules [OMG, 2008: Section 12.1.2] govern the behaviour of business activities and operations. The behavioural or operative rules are ones that direct the activities involved in the business affairs. Akin to behavioural rules in SBVR, OCL's behavioural constraints such as pre/post conditions [OMG, 2010: Section 7.3.4] are particularly specified to handle behaviour of respective methods of classes and objects. The OCL pre/post conditions also specify state change.

F. Why SBVR is Suitable for Intermediate Representation?

Recently, in many approaches, SBVR is used as an intermediate representation [Cabot, 2010; Pau, 2008]. The following Characteristics of SBVR make it a suitable option for intermediate representation in translation of one language to another language, especially if one language in the translation is a natural language and the other is a formal language. Following is an overview of such Characteristics of SBVR [OMG, 2010;Chapin, 2008]:

- SBVR vocabulary is concept centric, not word centric. That makes SBVR shareable in various communities.
- SBVR can be used to produce business vocabularies and rules which can be shared by more than one domain.
- SBVR supports precise Fact-oriented Modeling [OMG, 2008] in Formal Logic.
- SBVR is portable due to the use of OMG's Meta-Object Facility [OMG, 2008] (MOF) and it's supports of XML schema. With XML support, the business vocabularies and business rules can be interchanged among organizations and between software tools.
- SBVR supports multilingual development as it keeps symbols separate from their meanings.
- SBVR has support of textual notations such as Structured English and RuleSpeak [OMG, 2008]. Such notation can also be mapped to the SBVR metamodel.

- SBVR is based on Formal Logic with natural language interface.
- Since, SBVR has a meta-model, SBVR can be model transformed to other standards using model transformation technology.
- SBVR supports rule-based Application Software Development and Configuration [Chapin, 2008]. SBVR rules in combination with system design decisions are typically used to produce set of execution rules for software components.

On the basis of these Characteristics of SBVR, we are interested in SBVR to use it as an intermediate representation in the presented translation.

2.1.3 Natural Language Processing (NLP)

Processing of a Natural Language (NL) has been an area of interest for researchers for many decades. In the late nineteen sixties and seventies, researchers like Noam Chomsky [Chomsky, 1965], Chow and Liu [Chow, 1968] contributed in the areas of analysis and understanding of natural languages. However, automated processing of natural languages (such as English) is still a challenging task for NL community. Many contributions have been presented in the area of NLP but still there are many open questions to answer. Automated processing of NL is difficult due to reasons such as

- English is vast and has no domain boundaries.
- English sentence structure is ambiguous.
- Most English words have multiple meanings.
- A single meaning can be represented in multiple ways.

Typically processing a natural language involves a series of actions such as text segmentation, morphological analysis, Parts-of-Speech (POS) tagging, syntactic analysis, semantic analysis, and pragmatic analysis [Jurafsky and Martin, 2000]. In this research, the researcher will focus on a few of these, as pragmatic analysis is not involved in this research. Owing to the scope of our research, handling of discourse using pragmatic analysis is part of the future work. Currently, we are interested in lexical, syntax and semantic analysis of a natural language. In the following section, these phases are introduced as they are core of the presented research and will be discussed in forthcoming chapters.

A. Text Segmentation

Text segmentation is a primary phase in processing of a natural language. Text segmentation is a concept of linguistics typically used in computer science to break a stream of text up into meaningful elements. These meaningful elements are called tokens, lexicons or symbols. In computer science, the process of text segmentation is a part of lexical analysis in NLP.

For text segmentation, first of all a piece of text is segmented into sentences and this process is called sentence splitting. Afterwards, each sentence is further segmented into tokens and this process is called tokenization. The researcher is interested in text segmentation as a NL constraint can be composed of a multiple sentences and to apply syntax and semantic analysis to each sentence, the researcher needs tokenized form of the input NL constraint. Here, a Java sentence splitter has been used for splitting sentences and Java Tokenizer for text segmentation.

B. Morphological Analysis

Once the text is tokenized, next phase is morphological analysis and it is also a part of lexical analysis in NLP. Morphological analysis typically deals with the study of words (tokens) formation from smaller meaningful units called morphemes. Morphemes have two major types called: stems and affixes. The 'Stem' is the main morpheme of the word, which supplies the main meaning, and 'Affixes' add some additional meanings of various kinds. Affixes are further divided into prefixes, suffixes, infixes and circum-fixes. Prefixes precede the stem, suffixes follow the stem, circum-fixes do both and infixes are inserted inside the stem.

Lemmatization is a key phase of typical morphological analysis. In lemmatization, the lemma or stem of each token is identified. In this research, the researcher is interested in lemmatization as he needs core part of each token to compare with UML class model. Lemmatization as mapping of un-lemmatized text with UML class model can result in mismatch and can affect accuracy of the translation. In our research, a token 'customer' is treated differently than a token 'customers'. To consider both the tokens same, the researcher has to trim affix's' of the second token. Actually, during lemmatization, the affixes which are trimmed from lemma are used for grammatical purpose and in mapping with UML class model, grammatical information is not required.

The simplest methodology used for the morphological analysis is Stemming. The stemmers are based on stemming algorithms presented by Lovins [1968] and afterwards enhanced by Porter [1978] for suffix stripping. Another simple approach is to use lexical databases that associate lemmas and word-forms with inflectional information i.e. MULTEXT [Bel, 1996] system provides lexical list of lemmas. CELEX [Baayen, 1991] is another large multilingual database with extensive lexicons of English, Dutch and German languages. However, only MULTEX is available for downloading but it was not able to integrate with our implementation in Eclipse. Moreover, a root of an inflected form of word returned by a stemmer is not in a 'proper' dictionary word, while the researcher is interested in a word with 'proper' dictionary form.

In this research, the researcher aims to develop his own rule-based module for lemmatization.

C. Part- of-Speech (POS) Tagging

The Part-of-Speech (POS) tagging is a process of assigning a grammatical category (such as a noun, verb, determiner, etc.) to each token in a sentence. Each POS category is represented using a set of POS tags like NN (common noun), NNS (plural noun), NNP (proper noun), CD (cardinal number), VB (verb base form), VBZ (verb present, 3d person), VBD (verb past), MD (modal), RB (adverb), JJ (adjective), DT (determiner), IN (preposition), POS (possessive ending), CC(coordinating conjunction)[Toutanova, 2000], etc.

We are interested in generation of POS tags as syntax analysis uses POS tags in forming a parse tree and generating dependencies. In this research, POS tags can be involved in identification of various SBVR vocabulary items. As, the identified SBVR vocabulary items ultimately become the part of logical representation in semantic analysis, POS tagging is a key part of this research.

There are many examples of automatic POS taggers those can be used in building automatic word-sense disambiguation algorithms. The ENGTOWL tagger [Karlsson, 1995] is based on rule based architecture. Illieva [2005] introduced a methodology that can perform POS tagging using a tabular representation to identify subject, verb an object. Similarly, Li [2005] developed a methodology that does POS tagging using predefined rules. An additional support for POS tagging is WordNet [Fellbaum, 1998] that is a database of lexical relations that helps in extracting the lexical relations. Another example of POS tagger is the Stanford POS tagger was originally written by Kristina Toutanova [2000]. The Stanford POS tagger is an entropy-based POS tagger that uses of cyclic dependency network [Toutanova, 2003]. The Stanford POS tagger is 97% accurate in POS tagging [Manning, 2011]. Owing to its high accuracy, the researcher has used the Stanford POS tagger in his approach.

An issue, in typical POS tagging, is assignment of wrong tags to a token due to ambiguous tokens as there are many tokens in English which can be assigned multiple POS tags at a time e.g. a token ‘books’ can be identified as a noun as well as a verb in English. However, wrong POS tagging can be very dangerous as the accuracy of syntax and semantic analysis totally relies on accuracy of POS tagging. In this thesis, the researcher aims at dealing with such issues in POS tagging.

D. Syntax Analysis

In syntax analysis, syntactic or grammatical relationship is identified in various parts of a sentence. A parse tree is a typical way of graphically representing the grammatical relationships in a sentence. Traditionally, there are two types of parsers; top-down (goal-directed) parser and bottom-up (data-directed) parser. A top-down parser searches for a parse tree by trying to build from the root node *S* down to the leaves. On the other hand, a bottom-up parser starts with the leaves (words of the input) and tries to build a tree by applying rules from grammar one at a time. A typical parse tree involves various phrasal categories such as Noun Phrase (NP), Verb Phrase (VP), Preposition Phrase (PP), and Quantificational Phrase (QP). A parse tree is also the base of dependencies [Marneffe, 2006] that is the target of our syntax analysis. The syntactic dependencies help the researcher to identify possible relationships among various syntactic constituents of a NL constraint.

The researcher is interested in syntax analysis of NL text due to multiple reasons: (1) syntax analysis can provide the researcher with a parse tree and a set of dependencies. Such dependencies are actually relationships in various syntactic structures of a NL constraint. In this research, the researcher aims at mapping such relationship into equivalent relationship in SBVR and OCL; (1) during syntax analysis, the researcher can sort out voice of a sentence as in this research an active-voice sentence is treated differently than a passive-voice sentence; (3) with the help of syntax analysis, the researcher can deal with the logical operators in English as such logical operators play key role in our research.

A common way of parsing is by*dependency grammar* [Tesniere, 1959; Johansson, 2008]. In dependency grammar based parsing, structures are determined by the relations between a word and its dependents. Another way of syntactic parsing is *phrase structure grammar* proposed by Noam Chomsky [1968]. It was introduced by Gawron [1982] that phrase structure rules can be used in phrase structure grammar based parsing. Another type of parsers is probabilistic parsers (such as the Stanford parser). The Stanford parser is a lexically driven probabilistic parser. The Stanford parser is a Java implementation of a probabilistic natural language parser based on Probabilistic Context-Free Grammars (PCFG). The Stanford parser provides two outputs: (1) a phrase structure tree; and (2) a Stanford dependencies output. The researcher aims at using the Stanford parser in our approach for syntax analysis. To best of the researcher's knowledge the Stanford parser provides higher accuracy than do the other available parsers. The Stanford parser can be up to 84.1% [Cer, 2010] accurate. An additional benefit of the Stanford parser is that it also provides the typed dependencies. Here, typed dependencies are compact form of typical dependencies.

E. Semantic Analysis

Semantic analysis is used to identify different constituents of a sentence and analyse the input text to extract its explicit meanings, i.e., direct or apparent meanings of a sentence. During semantic analysis, the apparent meanings of a sentence are represented using a logical form.

The researcher is interested in semantic analysis for a number of reasons. First, semantic analysis can help us to identify SBVR based semantic roles. Secondly, quantifications, implication and negation can be processed with the help of syntactic analysis. Thirdly, a logical representation can also be generated as an end-product of semantic analysis. Such logical representation can be mapped to other formal representations such as SBVR, OCL, Alloy, etc.

There are different ways of analyzing semantics of NL text. Typically, semantic analysis is performed into two phases: shallow semantic parsing and deep semantic parsing. A brief description of both phases is given below:

In semantic analysis, a key phase is shallow semantic parsing in which the semantic or thematic roles are typically assigned to easy syntactic structure in a NL sentence. This process is also called *Semantic Role Labelling*. Typically used semantic roles are agent, action, patient, beneficiary, etc. However, in this research, the researcher has used SBVR based semantic roles as the researcher aims at generating SBVR based logical representation from NL constraint. Frame nets are commonly used for semantic role labelling [Fillmore, 2003]. The actual purpose of semantic role labelling is identifying relationship of participants (semantic arguments) with the main verb (semantic predicate) in a clause. SRL is a most common way of representing lexical semantics of NL text. Semantic labelling on a substring (semantic predicate or a semantic argument) in a constraint (NL sentence) 'C' can be applied. Every substring 's' can be represented by a set of indices as following:

$$S \subseteq \{1, 2, 3, \ldots., n\}$$

Formally, the process of semantic role labelling is mapping from a set of substrings from 'C' to the label set 'L'. Where 'L' is a set of all argument semantic labels,

$$L = \{a_1, a_2, a_3, \ldots., m\}$$

The semantic roles can act as an intermediate representation in NL to SBVR translation. Croft (1991) explained that exact number of roles cannot be specified. Various scientists have defined various semantic roles. Similarly, it was investigated that it is difficult to define boundaries in various role types [Dowty, 1999].

The typical resources required for the automatic role-semantic analysis can be lexicons and corpora. Common examples of a corpora is FrameNet that is a English lexical database [Baker, 1998] and it consists of a list of lexicons and a frame ontology that helps in identifying the semantic roles for each frame and frame-to-frame relations. Similarly, SALSA [Burchardt, 2006] is another corpus of German. SALSA is efficient for statistical systems. Another example is VerbNet [Kipper, 2000] that identifies semantic roles for lexicon in hierarchal categories. VerbNet uses set of semantic roles for syntactic transformations. PropBank [Palmer, 2005] is another support for semantic role annotation. However, in this research, the researcher aims at using UML class model as a lexical knowledge base due to the fact that a UML class model is context of its constraints.

Deep semantic analysis typically involves actions like word sense disambiguation, handling quantifications, quantifier scope resolution, anaphora (i.e. pronouns) resolution, and generating a logical representation. In our research we are interested in all of them except anaphora resolution as typically pronouns are not part of NL constraints and resolution of pronouns is not part of current scope of this research. Word sense disambiguation has a very wide scope. In word sense disambiguation, various issues such as polysemy, coherence, inference and discourse analysis are addressed. In this research, the researcher has employed word sense disambiguation at a very limited level as during labeling of semantic roles, there is possibility that a token may be assigned more than one semantic roles at a time. The researcher aims at resolving such issues in this thesis. Similarly, the researcher also aims working on quantifications and quantifier scope resolution as quantifications are always an important part of NL constraints. Final aim of deep semantic analysis is to generate a logical representation from NL text. This research aims at exploring the role of SBVR in a logical representation.

In semantic analysis another important field is handling of discourse and dealing with a donkey sentence. A donkey sentence is a classical NLP problem, e.g., "Every farmer who owns a donkey beats it". Donkey sentences cannot be solved using first-order logic as one has to deal with a certain kind of anaphora (statements about other statements). One approach to deal with such issues is Discourse Representation Theory (DRT) [Kamp,1981]. DRT is typically based on dynamical databases called Discourse Representation Structures (DRS). The DRS's are associated with a particular sense. Another approach associated with the dynamic semantics is Dynamic Predicate Logic (DPL) presented by Mus [1991]. DPL helps in specifying meanings of an action that modifies the receiver's information state. Semantic Frame [Fillmore, 1992] is a concept used to share the same set of roles using a set of predicates. The concept of semantic frame also assists in sorting out the definitional problems of semantic roles in universal sets. Frame semantics are helpful in machine translation [Boas, 2005] due to their ability of having similar frame-to-frame relations in the source and target languages. Pinkal [1991] proposed a revised semantic binding condition to address this issue by permitting the binding of indefinite NP in accordance with intuitions. However, Pinkal proved that in donkey sentences, the anaphoric relations are not solely specified by syntactic components. The work, discussed above proposes a solution for donkey sentences. However, processing of the donkey sentences is not part of scope of this research.

2.1.4 Model Driven Architecture

Model Driven Architecture (MDA) [OMG, 2010] is a flavor of model-driven development (MDD) proposed by the OMG. MDA is a software design approach and its typical application is development of software systems. Using MDA, platform independent models can be mapped to domain specific code. However, in this thesis, the researcher complements that functionality by using model transformations to generate formal language's code from NL specifications.

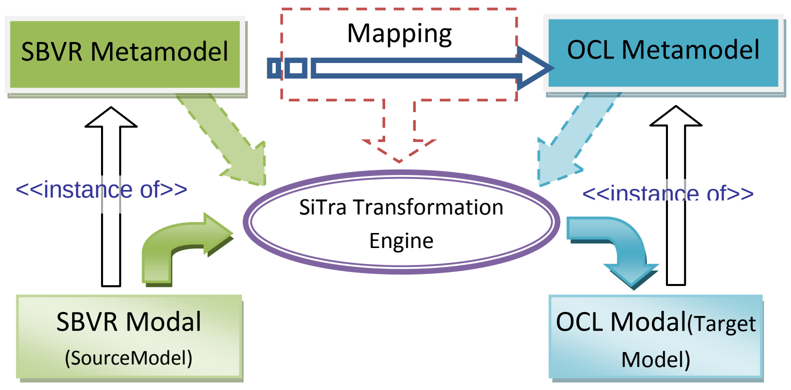


Figure 2.13: An overview of MDD

Model transformation is the core process in MDD and MDA that involves automated creation of new models, depicted in Figure 2.13, can be described briefly as follows:

> Model Transformations rely on the "instanceof" relationship between models and meta-models to convert models [Dang, 2009]. Model Transformations define the mappings rules between two modeling languages meta-models. Rules typically define the conversion of element(s) of the source meta-model to equivalent element(s) of the destination meta-model. The Model Transformation frameworks execute the Model Transformation implementations on models. Upon execution with a given model, the necessary rules are applied by the transformation framework applying rules to generate an equivalent model in the destination modeling language.

There are different types of model transformations such as model-to-model, model-to-text and text-to-model transformations [Cabot, 2007]. The Model-to-Model Transformation is used to transform a model into another model e.g. transforming UML/OCL to Alloy [Anastasakis, 2007], SBVR to UML [Raj, 2008], [Hina, 2011] UML/OCL to SBVR [Cabot, 2009], Alloy to UML/OCL [Shah, 2009], SBVR to SQL [Moschoyiannis, 2010], SBVR to BPMN [Steen, 2010], etc. The Model-to-Text Transformation is used to translate a model to a natural language representation e.g. transforming OCL/UML to NL [Raquel, 2008]. The Text-to-Model

Transformation talks over interpreting the natural language text and creates a model from the interpretation.

In this research, the researcher aims at using model-to-model transformation for automated transformation of SBVR to OCL. A Typical Model Transformation can be employed by creating abstract syntax of source model and then converting it into the target model representation using the model transformation rules. To achieve this goal, the researcher aims at using a set of transformation rules to perform the proposed transformation of SBVR to OCL.

Basic approaches used to perform a model transformation are Graph Transformation and Relational Model Transformation [Kuster, 2004]. Graph Transformation is employed by creating abstract syntax of source model using typed attributed graphs [ibid] and then finally converting them into the target model representation using graph transformation rules. On the other hand Relation Model Transformation approach proposes the use of QVT (Query/View/Transformation) [ibid] approach.

A. Transformation Rules

A typical model transformation is carried out by using a rule based approach to translate source text or a model conforming to its meta-model into a target text or model conforming to its meta-model. Rule based model transformations employ set of transformation rules to map source model to a target model. A transformation rule *r* maps one component of the source model using a source transformation rule *rs* with one component of target model using a target transformation rules *rt*. It can be represented as *r*: S → T.

Transformation rules were individually defined for SBVR to OCL transformation. Defined transformation rules were based on If-then-Else structure. Each rule consists of a component from the source model (such as SBVR) and one component from the target model (such as OCL) inspects source input and the mapping.

The researcher has defined a number of states[1] for the source model, e.g. $Y = \{y_1, y_2, \ldots.., y_n\}$ is a set of states for source model. Similarly, a number of states for the target model have been defined, e.g. $Z = \{z_1, z_2, \ldots.., z_n\}$ is a set of states for target model. For mapping, the states of

[1] State is an element of the metamodel

input source model are matched with possible states of the target model. An occurrence of X from the source model is looked within the all occurrences of Z from the source model and if the match is found, the matched state of source model is given concrete syntax of the target model.

B. Simple Transformer (SiTra)

Simple Transformer (*Si**T*ra) has been developed by Akehurst et al. [2008]. It is a simple and lightweight implementation of an extensible transformation engine. A conceptual outline of the *Si**T*ra framework is shown in Figure 2.14. There are two interfaces in the*Si**T*ra transformation framework: the `Transformer` interface and the `Rule` interface. The `Transformer` interface provides the skeleton of the methods to achieve the transformation. The `Transformer` interface consists of two key methods: the `transform()` method and the `transformAll()` method. On the other hand, the `Rule` interface is a set of mapping rules (defined by the user), which need to be implemented by the modeller according to the transformation rules.

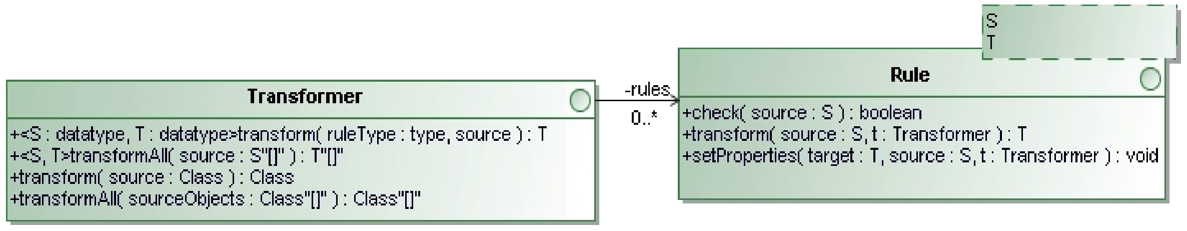

Transformer
+<S : datatype, T : datatype>transform(ruleType : type, source) : T
+<S, T>transformAll(source : S"[]") : T"[]"
+transform(source : Class) : Class
+transformAll(sourceObjects : Class"[]") : Class"[]"
-rules
0..*
S
T
Rule
+check(source : S) : boolean
+transform(source : S, t : Transformer) : T
+setProperties(target : T, source : S, t : Transformer) : void

Figure 2.14: Explanation of SiTra Model

We have defined such transformation rules in Section 5.3. However, the use of SiTra is very simple as modeller needs to implement the `Rule` interface by using defined set of transformation rules. The `Rule` interface consists of three methods as depicted in Figure 2.14. First method is `check()` that is involved in the rule interface. The second method `build()` method is executed to generate the target model element. The third method `setProperties()` is involved in setting the attributes and links of the newly created target element.

2.2 Related Work

In this section, related work in the area of NLP application in software engineering is presented. Moreover, an account of related work in the area of model transformations and tool support for OCL and SBVR is given in this section. The related work discussed in this section not only emphasizes the significance of the research but also highlights the motivations of the research.

2.2.1 NLP for Automated Software Engineering

Applications of NLP in the field of software engineering are significant especially to improve accuracy, productivity, flexibility, multilinguality and robustness [Leidner, 2003]. An example of an application of NLP is automated software modeling (such as automated object-oriented analysis and automated generation of UML models) from NL software requirements specifications. Similarly, NLP has been applied to provide NL interfaces for automatically generating E-R models and SQL queries as well. Automated generation of OCL constraints for software (UML) models is also related to this area of research. An account of the work related to above mentioned fields is given below:

A. NLP for Automated Object Oriented Analysis

The role of NLP in the field of object oriented software modeling has been very important. One of the various contributions for automated object oriented analysis of natural language software requirements specifications and extraction of object oriented information from the NL specification was presented by Mich [1996] and the presented approach was implemented in a tool LOLITA [ibid]. Delisle [1999] and Perez-Gonzalez [2002] also presented NL based tools which could be used for object oriented analysis of NL specifications of software requirements. Similarly, linguistic information was used to analyse syntactically and semantically informal specifications and employ a semiformal procedure to extract object-oriented information [Juristo, 2000] that could be used to construct a model. Similarly, Li presented his work in which he addressed various issues in NL based automated object oriented analysis [Li, 2003]. MOVA [Clavel, 2007] is another tool that models, measures and validates UML class diagrams. Such techniques consume less time and require less human effort and expertise in analysis of NL software requirements. A similar approach can save time and resources when we analyse NL constraints to extract the SBVR vocabulary that will be used to construct a SBVR rule or an OCL constraint.

B. NLP for Automated UML Modelling

Various approaches and tools have been presented to automatically generate software models from NL specifications. An example of such work is the semi-natural language (4WL) presented to automatically generate object models from natural language text [Hector, 2002]. Its prototype tool GOOAL [Perez-Gonzalez, 2002] produces object oriented static and dynamic model views of the problem. Much research has been done on analysis of NL requirement specification [Bryant, 2008] and their translation to object oriented models [Seco, 2004], and programming languages [Bajwa, 2006]. A significant contribution by Harmain and Gaizauskas was their NL based CASE tool named CM-Builder [Harmain, 2003]. This CASE tool was restricted to create a primary class model. Such tools are real motivation for automated software modeling. NOESIS (Natural Language Oriented Engineering System for Interactive Specifications) is a WordNet based NL text analysis module [Nuno, 2003]. A tool REBUILDER based on NOISES is introduced by Gomes [2004] to generate class diagrams from NL specification. To generate the class diagrams, NOESIS first performs basics steps of NLP such as morphology, syntax and semantics analysis. Then a CBR (Case Based Reasoning) engine is used to retrieve cases from the case library based on the similarity with the target problem.

The work discussed above addresses only generation of UML models from NL specification of software requirements. However, no work is presented for automated generation of OCL from NL specification. Since OCL is an important part of UML models, there is a gap in research that demands an automated approach and a tool that can generate OCL constraints for the UML models. However, the presented techniques are a motivation for a NL based approach that can facilitate generation of OCL constraints.

C. NLP for Automated E-R Modelling

In addition to automated translation of NL requirements specifications to UML models is translation of NL specification to ER models [Omar, 2006]. Semantic heuristics were used to extract the relevant ER elements such as entities, attributes, and relationships from the specifications. The used approach is an extension of syntactic heuristic based tool ER-Converter.

ER-Converter provides 85% precision and to improve its accuracy the concept of semantic heuristics were employed. E-R Generator [Gomez, 1999] is another rule-based designed tool that performs semi-automatic generation of E-R Models from NL specifications. A heuristic based parsing algorithm was used to parse NL statements and then the linguistic structures were transformed into ER concepts.

The work we have discussed above is a real encouragement for the development of a NL approach that can simplify the generation of OCL.

D. NLP for Automated SQL Query Generation

Another application of NLP are natural language interfaces for databases. A model is presented for auto analysis of user requirement using NLP and custom model database generation [Al-safadi, 2009]. A CASE tool DBDT (Database designer in the Database development) is also presented. Similarly, Popescu [2003] and Nihalani [2011] presented a natural language interface for databases. In the same way natural language interfaces are also presented to communicate with data warehouses [Kuchmann-Beauger, 2011;Naeem, 2012].

The researcher's approach to generate OCL constraints automatically from NL specification of constraints is really motivated by such work as discussed above.

2.2.2 Automated Generation of OCL

Though usability of OCL is a long standing challenge for research community, not very much work has been presented so far to facilitate the writing of OCL. One effort has been done by Wahler [2008] to generate OCL using a pattern based approach. However, there are two issues with Wahler's approach. The first issue relates to the applications of Wahler's approach as it is semi-automatic [ibid] and the user has to extract manually the required information from NL constraints and then he/she has to select manually one or more patterns required to generate a specific OCL constraint. The second issue is that Wahler's approach is 69% accurate [ibid]. Here accuracy means failed vs. successful generations of constraints. Hence, it can be deduced that Wahler's approach cannot process almost one third of given number of constraints in a scenario. However, there are plenty of margins for improvement in accuracy.

In light of above mentioned facts, we can conclude that there is need of an approach that is not only fully autonomic but also more accurate in generating OCL expressions.

2.2.3 OCL Transformations

Various automated transformations have been presented from OCL to other formal languages. Examples of such transformation are OCL to JML [Hamie, 2004], OCL to Alloy [Anastasakis, 2007], OCL to NL [Burke, 2007], [Raquel, 2008], OCL to SQL [Heidenreich, 2008], OCL to SBVR [Cabot, 2009], etc. A brief description of these transformations is given below:

OCL is transformed to Java Modeling Language (JML) by Hamie [2004], where JML is a specification language to state Java classes and interfaces. It was further presented that OCL to JML transformation can assist in automated mapping of object-oriented models expressed in UML and OCL to Java classes and interfaces.

OCL has also been transformed to natural language (English) by Burke [2007]. The presented work is the part of the Key Project and the major emphasis of the presented work was to integrate the formal software specification and verification into the industrial software engineering process. A Grammatical Framework (GF) is used that is based on a grammar formalism and toolkit. GF grammars separate abstract from concrete syntaxes. A similar contribution was presented by Raquel [2008].

OCL to Alloy transformation is presented by Anastasakis [2007]. The presented transformation was used for automated analysis of UML models. The reverse of this transformation Alloy to OCL is presented by Shah [2009]. The presented work uses *S*i*T*ra [Akehurst, 2007] library to transform OCL/UML to Ally and back.

A transformation from OCL to SQL was presented by Heidenreich [2008]. The focus of research was to provide an automated way of generating SQL queries from integrity constraints specified in OCL. The presented transformation not only decreases development costs but also increases software quality.

OCL was transformed to SBVR by Cabot [2010]. In OCL to SBVR transformation, all possible textual objects in constraint language (OCL) that complement the UML graphical elements were mapped to SBVR. Such a transformation presents several benefits and applications and opens the door to representing the initial UML/OCL specification in a variety of different languages and notations for which a predefined mapping from SBVR has been already proposed. This work can help in generating business vocabularies from the already designed software models.

The work the researcher has discussed above highlights various transformation for OCL or transformation to OCL. However, presently, there is no approach that can transform NL or SBVR to OCL. The gap in current research really motivates for research and development of an approach automated transformation of NL and SBVR to OCL.

2.2.4 OCL Tool Support

A continuous research is in practice for designing tools to automate the process of OCL type checking. One of such tools is IBM OCL Parser [IBM, 2009] that is the first OCL tool written in Java by IBM. A model or OCL expression was given as input in a special file format. OCL parser was able to perform syntax checking and partial type checking. Dresden OCL Toolkit [Demuth, 2009] is another OCL compiler. This OCL toolkit was using a compiler that parses and semantically analyses the OCL expression to validate logic or meanings of an expression. Another famous tool is USE (UML-based Specification Environment) [Gogolla, 2007] used for the validation of UML models and OCL constraints. USE validation tool is comprised of two main components: a UML model simulator and an OCL parser and interpreter for constraint checking. OCL interpreter supports validation of OCL expression syntax and performs strong type-checking. ArgoUML [Rompaey, 2007] is an open source CASE tool which provides typical OCL syntax and type checking. This tool is based on Dresden OCL compiler. ModelRun [Akehurst, 2001] is one of the tools that not only provide support for direct execution of OCL expression but also are endowed with OCL based query execution shore up. ModelRun is product of Boldsoft and it has integrated support for the creation of model prototypes. Cybernetic OCL Parser [Emine, 2008] is a complete OCL compiler that provides syntactic and type checking of the OCL expressions was introduced by Cybernetic Inc. Cybernetic is a software company that is working on logical consistency checks in OCL expressions.

None of the tools, discussed above, provide support for automated generation of OCL constraints. However, a semi-automated tool Copacabana is presented [Wahler,2008] to enhance maturity level of UML class models. However, the tool is not available for download and it is not possible to decide up to what extent the tool can solve OCL usability problem.

Most of the tools the researcher has mentioned above are related to syntax checking, semantic checking, dynamic validation, test automation, code verification and synthesis. However, there is none of the tools available that can automatically generate OCL from NL constraints. Absence of any tool for automated generation of OCL from NL constraints is another motivation factor for the presented research.

2.2.5 SBVR Transformations

SBVR can be used for capturing natural language software requirements specifications. Since, SBVR is easy to machine process, software requirement specifications represented using SBVR can be automatically translated to other formal specifications. Example of such transformation is model transformation of SBVR to UML [Raj, 2008], [Hina, 2011], SBVR to R2ML [Nicolae & Wagner, 2008], SBVR to SQL [Moschoyiannis, 2010], SBVR to BPMN [Steen, 2010], SBVR to Ontologies [Karpovic, 2010], etc. A brief description of such transformation is given below:

SBVR was introduced by OMG to provide a formal representation to capture software and business requirements. The business models represented in SBVR are mapped to UML models by Raj [2008]. Similarly, the software requirements represent using SBVR rules are model transformed to UML class models by Hina [2011]. The focus of both transformations was to facilitate the generation of UML models from business and software requirements specifications represented in SBVR. A transformation from SBVR to R2ML is presented by Nicolae and Wagner [2008]. The purpose of this transformation was to generate R2ML language for existing SBVR rules to get higher semantic representation that can improve the level of business logic abstraction. Moreover, SBVR is transformed to Structured Query Language (SQL) by Moschoyiannis [2010]to generate automatically SQL queries from existing SBVR rules in a business domain. Another example of SBVR transformation is automated transformation of UML and OCL to SBVR [Cabot, 2009], explained in previous sections. To facilitate the automated generation of ontologies from SBVR business rules a transformation was presented by

Karpovic [2010]. One more contribution was by Friedrich [2011] to generate Business Processes (expressed in Business Process Modelling Notation) from SBVR business rules.

The present work draws attention to the gap in research on transformation of NL to SBVR and SBVR to OCL. To fill the gap in the existing research an approach is required for NL to SBVR and SBVR to OCL transformation.

2.2.6SBVR Tool Support

The SBeaVeR is a business model editor developed as an Eclipse plugin [Tommasi, 2006]. The SBeaVeR assists business modellers and analysts to create SBVR based business models and rules. The SBeaVeR uses a SBVR linguistic engine to validate the sentences representing fact types and business rules. The SBeaVeR also provides support to formalize the semantics of business knowledge in the form of business rules represented using the Structured English notation [ibid]. Another SBVR editor that provides syntax highlighting and auto-completing facility is presented by Marinos [2011].

The work discussed above highlights that currently there is no tool available that can generate SBVR rule representation from NL specification of constraints. This gap in research motivates for developing a tool that provides facility of automatic generation of SBVR rules from NL specifications.

2.3 Summary

In this chapter the researcher has presented an overview of the basic concepts such as OCL, SBVR, NLP and model transformations. These concepts have been used throughout the thesis. In the second section of the chapter, related work in the area of automated transformations has been presented. It is found that various researches have been conducted to translate NL specifications to UML diagrams, E-R models, SQL queries, etc. However, no work has been presented to translate NL specifications to OCL constraints. Similarly, OCL is mapped to Alloy, SBVR and other standard using model transformation technology. On the other hand, SBVR is mapped to UML, BPMN, SQL, and other formal languages. However, no research has been presented to model transform SBVR to OCL.

The presented related work identifies not only a gap in research for automated generation of OCL from NL specification but also highlights the need and want for an automated approach to generate OCL from NL specifications so that the software/business modelers may be assisted in the modeling of software/business models. The presented work related to model transformation highlights that such automated transformation can facilitate the writing of OCL and can improve the usability of OCL.

CHAPTER 3
PROPOSED SOLUTION

This chapter presents the thesis statement that reflects on the challenges undertaken in this research and sketches the solution provided. Moreover, a set of hypotheses are also stated that the researcher aimed at addressing in the rest of the thesis.

3.1 Thesis Statement

Two major factors can be identified contributing to low adaption of OCL: (1) usability of OCL and (2) absence of tool support to facilitate OCL writing. On the basis of the research discussed in the previous chapter, the researcher could identify various aspects that play a role in usability of OCL and make writing OCL difficult. A primary aspect is the complex syntax of OCL [Gogolla, 2007] because OCL is a declarative language and focuses on establishing relationships among various elements. Wahler [2008] presented a template based approach to contribute to OCL adaption by providing a simple interface for automated generation of OCL constraints. Wahler's approach allows the user to choose a required template from a wide range of OCL templates, assign the parameters and use them. Such approach can help an expert user. However, the key challenge for a novice user is the selection of a correct template and if a constraint involves more than one template, the scenario becomes more complex. The second aspect of OCL's usability problem is the ambiguous nature of OCL constraints as several equivalent implementations for a constraint are possible in OCL [Cabot, 2008]. Cabot proposed an approach for automatic disambiguation of the constraints by means of providing a default interpretation for each kind of ambiguous expression. But a designer has to be aware of all the possible states while writing an OCL constraint to avoid the identified ambiguities. The third aspect of OCL's usability problem is understandability of overly complex OCL expressions commonly used in large software models [Correa, 2007]. The refactoring techniques are used to improve the understandability of OCL specifications but the employment of refactoring technique can be an overhead in the process of software modeling.

In parallel to difficult syntax of OCL, absence of a tool (that facilitates writing OCL) also contributes to least adoptability of OCL. None of the currently available tools is capable of assisting users in writing OCL constraints. The available OCL tools do not provide any assistance in writing OCL expressions syntactically correct and simple enough to interpret semantically. Examples of such tools are Dresden OCL Toolkit [Demuth, 2009], IBM OCL Parser [IBM, 2009], USE [Gogolla, 2007], ArgoUML [Rompaey, 2007], Cybernetic OCL Compiler [Emine, 2008], etc. All these tools are limited to syntax verification and type checking of the already written OCL constraints. As discussed in Chapter 2, Section 2.2.2, Wahler's approach is semi-automatic and less accurate as well. To the best of this researcher's knowledge, there is currently no tool that can automatically generate OCL from NL.

In the context of the above described scenario, the proposed research will address the problem of easing adaption of OCL by providing a NL based user interface to write OCL. The following are the key scientific issues involved in the proposed solution:

- The syntactic and semantic analysis of the NL constraint to understand the meaning of the given text and extract the OCL constraints related knowledge from that text.
- How the informal representation (such as NL constraint) can be transformed into formal representation (i.e. SBVR and OCL).The generated SBVR is checked and verified by the SBeaVeR tool and generated OCL is checked by the USE tool.
- Investigating how SBVR can be incorporated in making natural languages syntactically and semantically restricted and also exploring how SBVR can be used as an intermediate representation for NL to OCL transformation to generate OCL constraints.

The facts presented above highlight the need of an approach that allows development of tools and techniques to provide assistance in writing OCL. In this thesis, the researcher presents a radically new approach by bringing together two main domains of computer science: (1) natural language processing and (2) model transformation. Using natural language processing for transformation of NL constraints to OCL is a novel work. But the researcher adopted a systematic way that transforms NL constraints to OCL by using SBVR as an intermediate representation. Here, use of SBVR not only helped in dealing with ambiguities of NL constraints but also assisted in transformation to OCL due to its basis on formal logic.

3.2 Hypothesis and Assumptions

This thesis aims at extending the existing work in the field of NLP by defining a model for understanding and analyzing natural language constraints and translating them into a formal specification such as OCL. This work aims at using NLP to translate NL constraint to SBVR rules and then using model transformation technology to transform the SBVR representation of constraints to OCL. The hypotheses for the presented work are stated below:

1. That it is possible to build a tool using a model transformation-based approach in the NLP domain that can translate informal specification (i.e., English constraints) into a formal specification (i.e., OCL invariants, preconditions and post-conditions). The presented tool can also be used with the existing Eclipse platform as an Eclipse plugin and the generated OCL constraints can be directly used in major CASE tools.

2. That by using the presented approach and tool, it is also possible to assist the software designers in generating syntactically accurate, semantically precise and consistent OCL constraints that can be incorporated to annotate UML models. This will allow the software designers to solely concentrate on the software quality issues rather than designing details.

3. Having two versions of text (i.e., NL constraint and SBVR rule); NLP and the SBVR standard can help in validating a part of the transformation to ensure correct interpretation of the NL constraint.

4. We assume that the UML model is a suitable representation of the domain and the statements are made about this model and not the actual world. This is to say we assume models are reasonable representation of the domain. All models are wrong whereas some are useful.

5. There are sufficient consistencies in the language used in this approach that are amenable to automated NLP. For example, we assume most inherently ambiguous NL statements have a

default specific interpretation and truly ambiguous sentence for which there are several competing likely interpretations are rare.

6. The ability to map to OCL gives us sufficient assurance that we can map to other formalisms. Because, OCL is rather complex has three value logic with classic login embedded.

Additionally, this work aims at evaluating the effectiveness of such systems by seeing how well it meets the needs of software designers. Moreover, various case studies will also be used to evaluate the performance of the presented approach.

It is pertinent to mention here that the researcher is aware of the fact that the presented solution to generate OCL from NL constraints cannot be 100% accurate due to the informal nature of NL and infinite size of NL (such as English) vocabulary. Since, the researcher has used NL and automated generated SBVR in pair to resolve NL ambiguities and to clarify vagueness by pointing them out, this will not be a 100% solution either and the researcher is aware of it.

3.3 Used Approach

The NL2OCL is a NL-based approach that generates OCL from NL specification of constraints with respect to a target UML class model, where SBVR plays a role of an intermediate representation. The NL2OCL approach takes two inputs: (1) a NL statement (that is a specification of a constraint) and (2) a UML class model (that is the target of the NL constraint). Figure 3.1 depicts various phases of the NL2OCL approach.

The NL constraint is transformed to OCL in multiple phases. First of all, the NL constraint is linguistically analysed by the NL module. Linguistic analysis of the NL constraint involves syntactic analysis and semantic analysis and the output of the NL module. Then, another UML module parses the UML class model and extracts the SBVR vocabulary, e.g., Object Types, Characteristics, Fact Types, etc. The SBVR module maps the output of the NL module and the UML module to ensure that the output of the NL module should comply with the output of the UML module. Here, if any part of NL module's output does not comply with the UML module's output, the unmatched part does not become the part of the SBVR rule and user is given a message about the inconsistency. Finally, the OCL module maps the SBVR rule to an OCL constraint

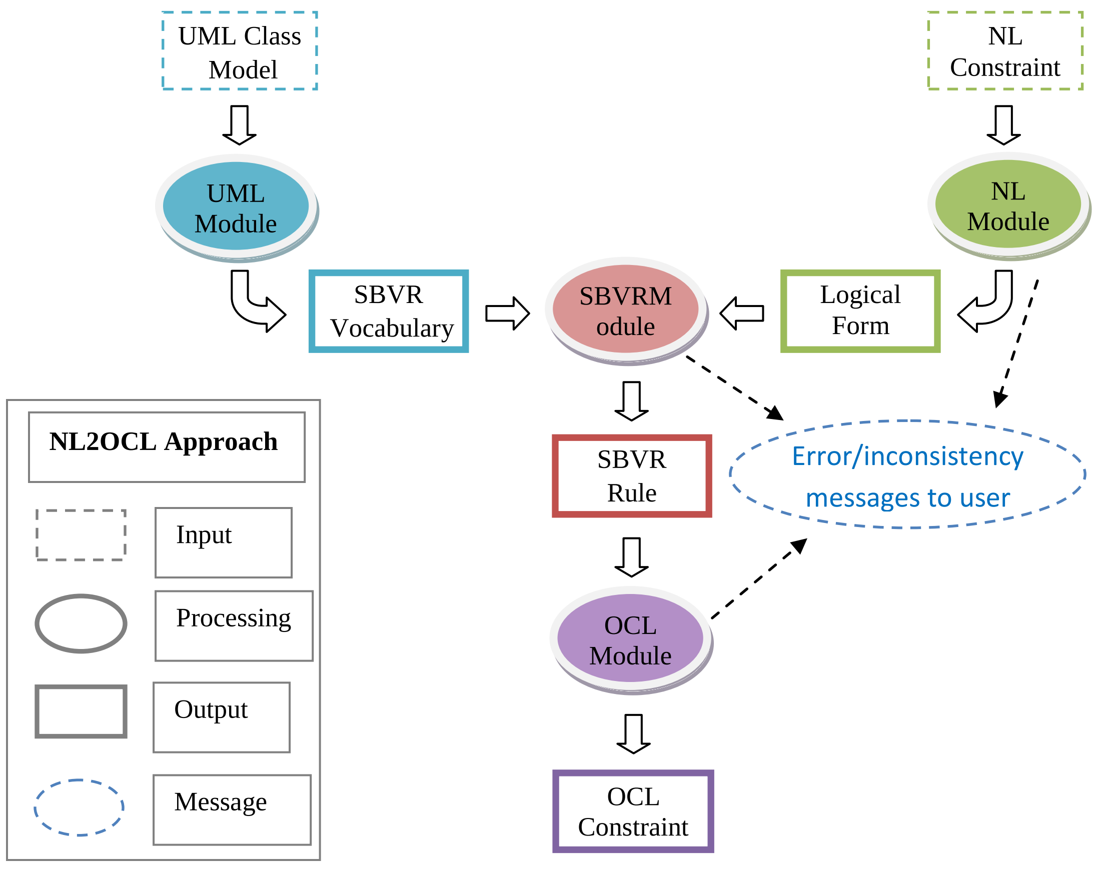


Figure 3.1: The NL2OCL Approach

All the steps involved in the NL2OCL approach (shown in Figure 3.1) for translating NL constraints to OCL are expressed in the form of an algorithm. The algorithm on which the NL2OCL approach is based is given below:

1. Give as input a text document that contains the NL description of a constraint
2. Give as input a UML class model that is the target of the NL constraint.
3. Parse the UML class model to extracts SBVR vocabulary, e.g., Noun Concepts, Object Types, Individual Concepts, Verb Concept, Characteristics, etc.
4. Pre-process the NL constraint to get rid of un-necessary text and prepare text for detailed syntax and semantic analysis
5. Perform syntax analysis to identify structural relationship among various syntactic parts of the NL constraint. If there is error/inconsistency with the SBVR vocabulary, give message to the user.

6. Use the identified structural relationship for SBVR based semantic role labeling of each part of the NL constraint. If there is error/inconsistency with the SBVR vocabulary, give message to the user.
7. Map the SBVR vocabulary with the output of the semantic analysis of the NL constraint. If there is error/inconsistency with the SBVR vocabulary, give message to the user
8. If the mapping is successful, generate the SBVR rule by applying semantic formulations or else notify the user to correct the NL constraint by removing the extra contextual[2] information from the NL constraint.
9. Identify the type of the SBVR rule, e.g., a structural rule or a behavioural rule.
10. If the type of a SBVR rule is a structural rule then it is translated to an OCL invariant.
11. If the type of a SBVR rule is a behavioural rule then it is mapped to an OCL pre-condition or post-condition.
12. Generate OCL context. If context is not given in NL constraint, give error message to the user and restart processing.
13. Generate body of OCL constraints involving expression and navigations. If there is error/inconsistency with the UML model or some information is missing, give message to the user.
14. Integrate output of step 12 and 13 to generate a complete OCL constraint.

Algorithm 3.1: Algorithm to Translate NL constraints to OCL

The steps of Algorithm 3.1 represent a generalized form of the actions performed in the NL2OCL approach. However, a single step in the algorithm can have sub-steps, as well. Theoretical detail of all these steps is explained in Chapter 4 and Chapter 5 with examples, while the implementation details are provide in Chapter 6.

The proposed solution to automatically generate OCL from NL specificationis always capable of producing the wrong analysis but that in such circumstances the produced formal representation is correct for a particular, valid and potentially correct interpretation and can be corrected by manual intervention.

[2]Any piece of information that is not part of the target UML class model

3.4 Summary

In this chapter, thesis statement has been discussed in a detail. Moreover, the solution to address the problem of OCL usability has also been presented in this chapter with a set of hypothesis. Additionally, the NL2OCL approach has been presented based on an algorithm discussed in Section 3.3. The details of the NL2OCL approach are given in following chapters.

CHAPTER 4
TRANSLATING NATURAL LANGUAGE TO SBVR

As the researcher said in the previous chapter, the Semantics of Business Vocabulary and Rules (SBVR) standard is used as a pivotal representation in Natural Language (NL) to Object Constraint Language (OCL) transformation. SBVR is chosen as a pivotal representation due to its peculiar features that is SBVR is not only easy to understand for the natural language readers but also is simple to transform to other formal languages such as OCL. Moreover, a SBVR based representation is easy to interchange among multiple platforms and tools due to the support of XMI (XML Metadata Interchange) and MOF (Meta-Object Facility) [OMG, 2008]. Since, SBVR has already been used as an intermediate representation [Cabot, 2010], [Pau, 2008], we aim to exploit the strength of SBVR in NL2OCL translation. There are many features (discussed in Chapter 2, Section 2.2.3) of SBVR that make it a suitable option for intermediate representation in translation of one language to another language, especially if one language in the translation is a natural language and the other is a formal language.Though, the approach uses automated generated SBVR in pair with NL representation to resolve NL ambiguities and clarify NL vagueness by pointing them out; even then the NL2OCL approach cannot be 100% correct. In this chapter, the first half of the NL2OCL approach is presented that deals with NL to SBVR translation.

To generate SBVR representation from NL constraints, two things are required: (1) SBVR vocabulary (such as Object Types, Individual Concepts, Fact Types, etc.) and (2) relationships among various SBVR vocabulary items. We use NL to SBVR translation to extract both these types of elements from NL constraints. In NL to SBVR translation, the researcher applied typical

NLP techniques such as syntax and semantic analysis to extract the required information. The syntax analysis provides the researcher with a parse tree and set of dependencies while semantic analysis uses these dependencies to generate a SBVR vocabulary based logical representation that contains both SBVR vocabulary and relations among various SBVR vocabulary items.

NL to SBVR translation is an automated approach based on NLP. In NL to SBVR translation, there are two key challenges: (1) analyzing NL constraints to generate a SBVR vocabulary based logical representation and (2) mapping the logical representation to SBVR rule representation. In the researcher's approach, analysis of NL constraints involves three sub-phases such as pre-processing, syntax analysis, and semantic analysis. The used framework for analysis of NL constraints is shown in Figure 4.1, where the researcher has shown two layers: (1) logical layer and (2) user interface layer. Here, the user interface layer provides both the inputs and receives the output, while the logical layer handles actions like pre-processing, syntax and semantic analysis.

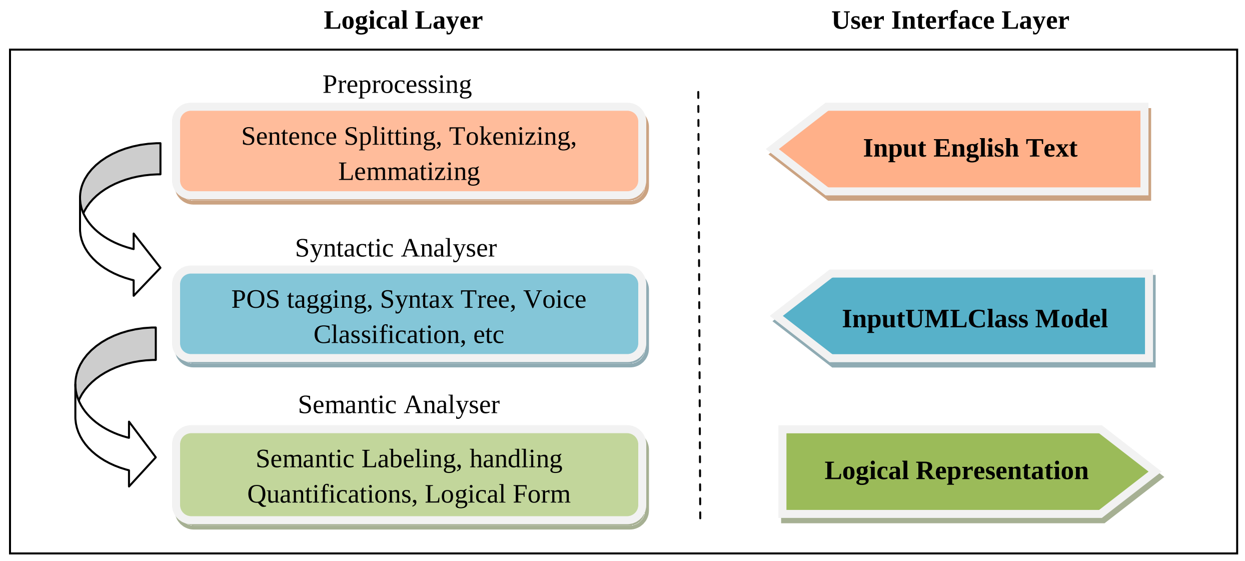

Logical Layer
User Interface Layer
Preprocessing
Sentence Splitting, Tokenizing, Lemmatizing
Syntactic Analyser
POS tagging, Syntax Tree, Voice Classification, etc
Semantic Analyser
Semantic Labeling, handling Quantifications, Logical Form
Input English Text
InputUMLClass Model
Logical Representation

Figure 4.1: A framework for analysis of natural language constraints

The NL2OCL approach is always capable of producing the wrong analysis but that in such circumstances the produced formal representation is correct for a particular, valid and potentially correct interpretation and can be corrected by manual intervention.

In the remaining part of the chapter, Section 4.1, 4.2, and 4.3 provide the details of the three steps involved in processing of NL constraints and Section 4.4 highlights the details of generation of SBVR rule representation.

4.1 Pre-processing

In the pre-processing phase, the input NL text is prepared for the detailed processing such as syntactic and semantic analysis. The input text contains a natural language specification of a constraint that is specifically defined for a UML class model. Major steps involved in pre-processing phase are splitting the sentences, tokenizing the words, and lemmatization. Following are the brief description of these three sub-phases of pre-processing.

4.1.1 Sentence Splitting

If the input text contains multiple sentences, each sentence is considered as a separate entity. During sentence splitting, the margins of a sentence are identified and each sentence is separately stored and is treated as a separate constraint. The Stanford POS tagger is used for the sake of sentence splitting.

4.1.2 Tokenization

After sentence splitting, each sentence is further processed to identify tokens. The purpose of the tokenization phase is to identify tokens in a given piece of text for the detailed syntactic analysis. A simple example of the tokenized text is shown in Figure 4.2:

English:	**An increase is awarded to all workers with injury.**
Tokens:	[An] [increase] [is] [awarded] [to] [all] [workers] [with] [injury] [.]

Figure 4.2: Tokenized text using Stanford Parser

Here, Figure 4.3 shows a complex example of tokenization as it involves *'s* that should be treated as a separate token. The researcher has used the Stanford POS tagger that can handle such difficult cases.

English:	**A customer’s age cannot be more than 18 years.**
Tokens:	[A] [customer] [’s] [age] [can] [not] [be] [less] [than] [18] [years] [.]

Figure 4.3: Tokenized text using Stanford Parser

4.1.3 Lemmatization

In lemmatization, the morphological analysis of words is partially performed to remove the inflectional endings and it returns the base form or dictionary form of a word. The base form of a word is typically represented as ‘lemma’. We identify lemma (base form) in the tokens by removing various affixes attached to the tokens. Here we store two copies of each sentence: one copy with the original tokens and the second copy contain the lemmatized tokens. The copy of NL constraints with original list of tokens is important to save as the removed parts are used to identify POS tags in the syntactic analysis phase. An example of lemmatization is representation of a token “awarded” as “award+ed”. Similarly, in Figure 4.3, another token “workers” is processed as “worker + s”.

4.2 Syntax Analysis

In syntax analysis phase, the pre-processed text is processed to extract grammatical structure and possible dependencies between particular grammatical structures. Grammatical structure of a sentence is pertinent to identify as sentences with different grammatical structures are treated differently. For example, the algorithms used to process the active-voice sentences cannot be used to process a passive-voice sentence.

The output of a typical syntax analysis phase is a parse tree. A parse tree can be represented using a textual representation or a graphical representation. The example parse trees discussed in this chapter are represented using the textual representation generated by the Stanford parser. Besides parse tree generation, the researcher also performs some additional steps for robust extraction of detailed information required in the semantic analysis, such as classification of active-voice and passive-voice analysis of logical operators, etc. It is a fact that the accuracy of syntax analysis affects the semantic analysis and rest of the processing phases as the output of the syntax analysis is input of semantic analysis phase. Hence, any mistake or misinterpretation during syntax analysis phase propagates in rest of the processing such as semantic analysis, SBVR rule generation, and OCL generation. Considering the importance of syntax analysis, following steps are performed to syntactically analyse a NL constraint:

1. POS Tagging
2. Generating Parse Tree and Dependencies
3. Voice Classification
4. Processing Conjunctions an Disjunctions
5. Generating an Intermediate Representation

The description of all the five steps involved in the syntax analysis of NL constraints is given below.

4.2.1 Part-of-Speech (POS) Tagging

POS-tagging is the first phase of syntax analysis. In POS tagging, each token is assigned a part-of-speech such as noun, verb, preposition, etc. A set of name abbreviations such as NN, NNS, CD, VB, VBZ, etc., are the output of POS-tagging. The researcher has used the Stanford POS tagger for the sake of POS tagging due to its accuracy that is 97% [Manning, 2011]. The Stanford POS tagger was originally written by Kristina Toutanova [2000]. The Stanford POS tagger is an entropy-based POS tagger that laterally involved the use of cyclic dependency network [Toutanova, 2003].

The researcher has used the Stanford POS tagger version 3.0.3 that can identify 44 various POS tags. An example of POS tagging of a simple NL constraint is shown in Figure 4.4 that involves one determiner 'a', two singular nouns 'customer' and 'age', one possession ''s', one Modal verb 'can', one negation 'not', one verb 'be', one comparative adjective 'less', one subordinating conjunction 'than', one cardinal number '18' and one plural noun 'years':

English: A customer's age cannot be more than 18 years.

Tokens: [A/DT] [customer/NN] ['s/POS] [age/NN] [can/MD] [not/RB] [be/VB] [less/JJR] [than/IN] [18/CD] [years/NNS] [./.]

Figure 4.4: Parts-of-Speech tagged text

Despite the high accuracy of the Stanford POS tagger, the researcher has identified a few cases where the Stanford POS tagger identifies wrong tags for a token. Identification of wrong tags is due to lexical ambiguity [Uejima, 2003]. In linguistics, a lexical ambiguity occurs when a word in a phrase or a sentence exhibits different syntactic representations in different cases. The wrong POS tagging by the Stanford POS tagger becomes more serious as the Stanford parser generates wrong parse trees and wrong dependencies.

A complex example of such cases is shown in Figure 4.5 where a token 'books' is wrongly tagged as 'NNS' by the Stanford POS tagger while, the token 'books' is a verb and should be tagged as 'VBZ'. The effect of wrong POS tagging is also shown in Figure 4.5 where the Stanford parser generates a wrong parse tree as there is no verb phrase in the tree. Similarly, the typed dependencies (collapsed) generated by the Stanford parser are also wrong as `det(books-3, A-1)` should be `det(customer-2, A-1)`, `nn(books-3, customer-2)` should be `nsubj(books-3, customer-2)`, and `dep(books-3, items-5)` should be `nobj(books-3, item-5)`. As these dependencies are directly translated to a logical representation in semantic analysis, it is very important to handle such issues.

English: A customer books two items.

POS Tagging: [A/DT] [customer /NN] [books/NNS] [two/CD] [items/NNS] [./.]

Parse Tree:
```
(ROOT
  (NP
    (NP (DT A) (NN customer) (NNS books))
    (NP (CD two) (NNS items))
   (. .)))
```

Typed Dependencies:
```
det(books-3, A-1)
nn(books-3, customer-2)
num(items-5, two-4)
dep(books-3, items-5)
```

Figure 4.5: Wrong POS tagging by the Stanford POS tagger

The researcher explained some other examples of lexical ambiguity in NL constraints in [Bajwa, 2012a]. One more example of lexical ambiguity is "A customer can bank on manager". In this example, word 'bank' is wrongly POS tagged 'NN' but the correct POS tag is 'VB'. A similar example is "The manager made him type on typewriter." In this example word 'type' is wrongly tagged as 'NN', while the correct tag is 'VB'. Cases of lexical ambiguity are quite common in the natural language sentences. Moreover, the incorrect POS tagging of such cases result in incorrect parse trees generated by the Stanford parser.

The researcher has used the Stanford parser for parsing the NL constraint and his semantic analyser totally relies on the performance of the Stanford parser, if POS tags go wrong, the parse tree is wrong and eventually the semantic analysis goes wrong resulting in wrong SBVR and OCL. To address cases of incorrect POS tagging due to ambiguity, contextual information is needed. As UML class model is target of NL constraints, the information of UML class models is used to decide the correct tags. As a solution, the POS tags identified by the Stanford POS tagger are also confirmed by mapping all POS tags with the UML class model. For NL to UML mapping, the researcher has used the set of mapping rules in Table 4.1.Here, a user is expected to use the vocabulary that is part of the target UML class model.

Table 4.1: Mapping of English elements to UML class model elements

UML class model elements		English language elements
Class names	→	Common Nouns
Object names	→	Proper Nouns
Attribute names	→	Generative Nouns, Adjectives
Method names	→	Action Verbs
Associations	→	Action Verbs

The mappings shown in Table 4.1 work as follows: if a token matches an operation or a relationship name, then that token should be classified as a verb. A token matches to a classor an attribute, then the token is classified as a common noun or proper noun.

Figure 4.6 shows a UML class model, in which it is shown that 'books' is an association in two classes 'Customer' and 'Item'. By using the set of mappings given in Table 4.1, it can be identified that the token 'books' cannot be a noun in the context of UML class model. However, the token 'books' should be classified as a verb and the correct POS tag of token 'books' should be 'VBZ' as the token 'books' comes after a model verb (MD) 'can' in the NL constraint. We have written a small rule-based module that corrects output of the Stanford POS tagger and the Stanford parser.

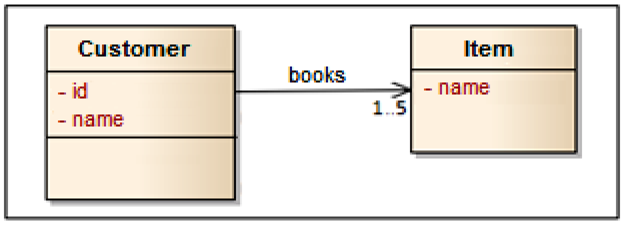


Figure 4.6: A UML Class model involving scenario of Customer booking an item

Once the POS tags of a NL constraint are corrected by using the mappings (given in Table 4.1) and the information given in the UML class model (shown in Figure 4.6), the parse tree and set of dependencies for the example (given in Figure 4.5) can be corrected. The corrected parse tree and the dependencies for the above discussed example are as shown in the Figure 4.7.There is a possible case that the UML model has multiple representations of 'books'. In that case, the user is given a message that he should manually select the correct meanings.

English: **A customer books two items.**

Tagging: [A/DT] [customer /NN] [books/VBZ] [two/CD] [items/NNS] [./.]

Parse:
```
(ROOT
          (S
            (NP (DT A) (NN customer))
            (VP (VBZ books)
              (NP (CD two) (NNS items)))
            (. .)))
```

Typed Dependencies:
```
det(customer-2, A-1)
nsubj(books-3, customer-2)
num(items-5, two-4)
dobj(books-3, items-5)
```

Figure 4.7: Corrected Parts-of-Speech tag, parse tree and dependencies

4.2.2 Generating Syntax Tree and Dependencies

A parse tree represents the syntactic structure of a NL constraint. Phrase structure rules are a common way to describe a given language's syntax. Such rules help in breaking down a NL sentence into chunks (phrasal categories) such as Noun Phrase (NP), Verb phrase (VP), Preposition Phrase (PP), and Quantificational Phrase (QP). A parse tree is also the basis of dependencies [Marneffe, 2006].Among various syntactic structures, the dependencies are the target of our syntax analysis. The syntactic dependencies help us in identifyingthe possible relationships among various syntactic structures of a NL constraint.

English: **An increase is awarded to any worker with injury.**

Parse Tree:
```
(ROOT
  (S
    (NP (DT An) (NN increase))
    (VP (VBZ is)
      (VP (VBN awarded)
        (PP (TO to)
          (NP
            (NP (DT any) (NN worker))
            (PP (IN with)
              (NP (NN injury)))))))
    (. .)))
```

Typed Dependencies:
```
det(increase-2, An-1)
nsubjpass(awarded-4, increase-2)
auxpass(awarded-4, is-3)
root(ROOT-0, awarded-4)
det(worker-7, any-6)
prep_to(awarded-4, worker-7)
prep_with(worker-7, injury-9)
```

Figure 4.8: Syntactic Tree generated using the Stanford Parser

The researcher has used the Stanford parser for generating a parse tree and the dependencies. The Stanford parser is 84.1% [Cer, 2010] accurate in generation and its dependencies. The Stanford parser provides two outputs: a parse tree and a set of dependencies. The Stanford parser generates two sets of dependencies: (1) dependencies and (2) typed dependencies. Here, typed dependencies are a compact version of simple dependencies. In the researcher's approach, typed dependencies are involved in representing the grammatical relations in a NL constraint. An example of a parse tree, for the above discussed example of NL constraint, generated by the Stanford parser is shown in Figure 4.8.

English: **The pay is given to all employees with bonus.**

Tagging: [The/DT] [pay/NN] [is/VBZ] [given/VBN] [to/TO] [all/DT] [employees/NNS] [with/IN] [bonus/NN] [./.]

Parse Tree:
```
(ROOT
  (S
    (NP (DT The) (NN pay))
    (VP (VBZ is)
      (VP (VBN given)
        (PP (TO to)
          (NP
            (NP (DT all) (NNS employees))
            (PP (IN with)
              (NP (NN bonus)))))))
    (. .)))
```

Typed Dependency:
```
det(pay-2, The-1)
nsubjpass(given-4, pay-2)
auxpass(given-4, is-3)
det(employees-7, all-6)
prep_to(given-4, employees-7)
prep_with(employees-7, bonus-9)
```

Figure 4.9: Typed dependency (collapsed) generated using the Stanford Parser

The typed dependencies generated by the Stanford parser are quite helpful in establishing relationships in various parts of a sentence. The Stanford parser is fairly efficient in processing complex sentences. However, the researcher has identified a few cases where the Stanford parser generates a correct parse tree, but wrong dependencies on account of attachment ambiguity. Attachment ambiguity is a type of syntactic ambiguity where a prepositional phrase or a relative clause in sentence can be lawfully attached to one of two parts of that sentence [Kiyavitskaya, 2008]. An example of such cases is shown in Figure 4.9. In this example, it is shown that the typed dependencies generated by the Stanford parser are wrong such as `prep_with(employees-7, bonus-9)`. However, the correct typed dependency for this example should be `prep_with(pay-2, bonus-9)`to represent the actual meaning of the example, i.e., the pay with bonus is given to all the employees. As the researcher explained earlier, the output of the Stanford parser is input of our semantic analyser, the wrong typed decencies lead to wrong semantic role labelling and wrong logical representation. For correct NL to SBVR an OCL transformation, we need to resolve such cases.

It is a common knowledge that involvement of context is the major reason of attachment ambiguity. Contextual information, such as a UML class model, can be used to resolve such issues. Figure 4.10 shows a UML class model that can help us to identify the correct dependencies of the example discussed in Figure 4.9.

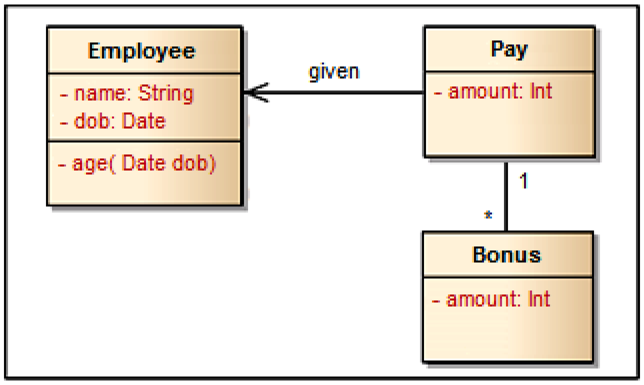


Figure 4.10: A UML class model involving employee, pay, and bonus

The relationships in UML class model, such as associations (directed and un-directed), aggregations and generalizations, can help us to deal with such cases of attachment ambiguities in English. To correctly identify the attachment of the noun ‘pay’ with other nouns ‘bonus’ instead of noun ‘employee’, the researcher maps the (three) candidate English elements in the NL constraint (such as nouns) to the classes in the UML class model.

The used mapping for attachment ambiguity resolution is slightly different from the mapping used in Section 4.2.1 to resolve lexical ambiguity. To resolve such cases, the researcher has written a simple algorithm as below:

1. Each noun is mapped to a class name in the input UML class model.
2. If all nouns are mapped to respective classes in the UML class model, the associations between those classes are analysed.
3. If there is a direct association between two candidate classes, they are attached to each other. Otherwise they are not attached to each other.

Algorithm 4.1: An algorithm to handle attachment ambiguity

The case of attachment ambiguity given in Figure 4.9 involves three nouns ‘pay’, ‘employees’, and ‘bonus’. All these three nouns are mapped to classes (such as ‘Pay’, ‘Employee’, and ‘Bonus’) in the UML class model shown in Figure 4.10. After this mapping, the associations in all three classes are analysed. The Stanford parser wrongly identifies that noun ‘bonus’ is attached to the noun ‘employees’. However, the UML class model shows that there is no direct relationship in classes 'Bonus' and 'Employee'. While, there is a direct relationship in class 'Pay' and class 'Bonus'. By using this information, we can correct the wrong dependencies by associating 'Bonus' to 'Pay' instead of 'Employee'. The corrected parse tree and dependencies are shown in Figure 4.11.

English: The pay is given to all employees with bonus.

Tagging: [The/DT] [pay/NN] [is/VBZ] [given/VBN] [to/TO] [all/DT] [employees/NNS] [with/IN] [bonus/NN] [./.]

Parse Tree:
```
(ROOT
  (S
                (NP
                  (NP (DT The) (NN pay))
                  (PP (IN with)
                    (NP (NN bonus))))
                (VP (VBZ is)
                  (VP (VBN given)
                    (PP (TO to)
                      (NP
                        (NP (DT all) (NNS employees))))))
                (. .)))
```

Typed Dependency:
```
det(pay-2, The-1)
nsubjpass(given-4, pay-2)
auxpass(given-4, is-3)
det(employees-7, all-6)
prep_to(given-4, employees-7)
prep_with(pay-2, bonus-9)
```

Figure 4.11: Corrected typed dependencies (collapsed)

As we cannot change code of the Stanford parser, we correct the output of the Stanford parser to reflect the correct relationships in dependencies. The researcher has written a rule-based module that can perform the steps given in Algorithm 4.1that is used to correct the dependencies, if they are wrong. After correction the corrected dependencies are shown to the user for his approval. If the user is satisfied, rest of the processing is performed. Otherwise, the user is allowed to correct the dependencies. Here the output is shown to the user by a message and the user can manually classify if it is not correctly classified.

The researcher has generalized the used approach so that all the variations of the discussed type of attachment ambiguity can be handled. For this purpose, the analysis of the relationships in classes of a UML class model such as associations (directed and un-directed), aggregations and generalizations can play a key role.

4.2.3 Voice Classification

In voice classification phase, a sentence is classified into the active or passive voice category. It is important to classify voice in NL constraints, as passive-voice sentences are treated differently as compared to active-voice sentences in the NL2OCL approach, due to different grammatical structure of both types of sentences. Typically, the passive voice implies focus on the grammatical patient (thematic object or beneficiary of the action) in place of the agent (actor of the action) of the sentences. Various grammatical features manifest passive-voice representation such as the use of past participle tense with main verbs can be used for the identification of a passive-voice sentence (see Figure 4.12).

[The] [order][was] [$_{\text{Past_participle_Tense}}$placed][.]

Figure 4.12: Identifying passive voice sentences

The use of 'by' preposition in the object part is also another sign of a passive-voice sentence. However, the use of by is optional in passive-voice sentences (see Figure 4.13). Using this information, a set of rules was defined to classify the voice of a sentence. The examples in Figure 4.13 show the use of past participle tense and 'by' preposition in passive voice sentences.

[The] [order][was] [$_{\text{Past_participle_Tense}}$placed] [$_{\text{by_preposition}}$by] [the] [customer][.]

Figure 4.13: Identifying passive voice sentences with 'by' preposition

After voice classification, various parts of a sentence are classified into a subject, verb or object. In case of a NP relation, there can be more than one subject or object relating to a verb. Similarly, in a VP relation, more than one verb can relate to a subject. This process is also called shallow syntactic parsing in which a sentence is analysed to identify various constituents such as subject, verb, object, etc. A set of heuristic rules were used to read the parse tree and classify the text into subject, object or nothing. Each sentence is divided into three sections: first section called subject starts from the first word of the sentences and ends before the start of the helping verb or main verb. Second section is verb that is combination of auxiliary verb and/or main verb. The third section object starts from the first word after verb section and ends at the last word of the sentence. To handle passive-voice sentences, we swap subject with the object.

4.2.4 Processing Conjunction and Disjunction

In the NL2OCL approach, the processing of logical operators is an important phase in analysis of NL constraints. Logical operators such as conjunction and disjunction can be analysed using syntactic information. It is important to identify the role of conjunctions and disjunctions in a NL constraint, as the conjunctions and disjunctions are reflected in logical representation and ultimately become part of the SBVR and OCL representation.

The following sub-sections explain the way conjunctions and disjunction are handled.

A. Resolving Conjunction

The researcher has used the parse tree information to identify conjunction ($p \land q$) in English sentences. Typically, conjunction is represented using a few words such as “and”, “but”, “yet”, “so” “moreover”, “however”, “although”, “even though”, etc. Conjunction can be used to join two nouns or two verbs. The ‘and’ conjunction used with two nouns is easy to interpret e.g. “A student and teacher can borrow a book”. However, the use of “and” conjunction with two verbs can be ambiguous e.g. “John opened the door and went out”. In this example, ‘and’ is exposing a sequence. We aim to handle such implicatures in the pragmatic analysis that is the part of the future work.

B. Resolving Disjunction

In natural language text, disjunction can be inclusive or exclusive. Typically, inclusive disjunction ($p \lor q$) means either p is true or q is true or both. In English, inclusive disjunction represented using “or” word. Similar to “and”, the use of “or” is also ambiguous in English as sometimes it disjoins nouns/adjectives and sometimes disjoins two propositions. The researcher has identified this difference and defined simple rules to classify the different use of “or” in all three possible situations. For example “A student can borrow a book or a CD.” Other possible representation of inclusive disjunctions in English can be the use of “unless”, “and/or”.

Exclusive disjunction (p ⊕q) (XOR) is also used in English, e.g., "Do you want milk or sugar in your coffee?" doesn't give a constraint "milk" XOR "sugar". In another context, the example, "do you want milk and sugar in your coffee?" suggests "milk" AND "sugar". Both are examples of inclusive OR. The researcher is aware of such subtle aspects. However, we have not handled this type of relations until mentioned explicitly. In natural languages, there are very few cases with XOR relations. Hence, the exclusive disjunction does not seriously affect our approach. However, the researcher aims to address the possible cases of exclusive disjunction in future.

4.2.5 Generating an Intermediate Logical Representation

In this phase, an intermediate logical representation is generated for the further semantic analysis performed in the next phase. At this stage, a tabular representation is generated containing the various syntactic chunks and their associated representations such as syntax type (e.g. subject, verb or object), various quantifications, logical operator if used in the NL constraint, and preposition associated to various nouns.

An example of an intermediate logical representation for a natural language constraint is shown in Table 4.2.

Table 4.2: An intermediary logical representation of a NL constraint

#	Chunk	Syntax	Quantification	Logical Operator	Preposition	EOS
1	customer	Subject	1			
2	can	H.Verb		Not		
3	place	M.Verb				
4	order	Object	more than 1			True

A major feature of this intermediary representation is that the active-voice and passive-voice are mapped to same representation such as subject of a passive-voice sentence is represented as object and object of a passive-voice sentence is represented as subject.

4.3 Semantic Analysis

A typical semantic analysis yields a logical form of a sentence. The logical form is used to capture semantic meaning and depict this meaning independent of a particular context. The goal of semantic analysis is to understand the exact meaning of the input text and identify various chunks of a sentence, such as Object Types, Verb Concepts, etc. For a complete semantic analysis of domain specific text, we have to analyse the text in respect of a particular domain such as a UML class model. Domain specific text analysis demands knowledge from the application domain to be mapped with the input English. In this research, UML class model is an application domain of the input NL specification of constraints.

The researcher's semantic analyser performs the following three steps to identify semantic relations in various parts of a NL constraint:

1. Shallow Semantic Parsing
2. Deep Semantic Parsing
3. Semantic Interpretation

All these steps of semantic analysis are explained below.

4.3.1 Shallow Semantic Parsing

In shallow semantic parsing the semantic or thematic roles are typically assigned to easy syntactic structure in a NL sentence. This process is also called *Semantic Role Labeling*. Typically used semantic roles are agent, patient, beneficiary, etc., whereas the researcher introduces SBVR vocabulary base semantic roles such as Object_Type, Fact_Type, etc. The researcher proposes the use of SBVR vocabulary based semantic roles in NL2OCL approach as the researcher aims at generating SBVR rule representation from NL constraint. If the researcher had used the typical semantic roles in this approach, he had to map the typical roles to SBVR vocabulary that was an overhead and could complicate the process semantic roles labelling.

The used SBVR based semantic roles in shallow semantic parsing are shown in Table 4.3:

Table 4.3: SBVR based semantic role labels used in SRL

English language elements		SBVR based role labels
Common nouns	→	Object_Type/ Characteristic
Proper nouns	→	Individual_Concept
Main verb	→	Verb_Phrase/ Fact_Type

Generative Phrases, Adjectives	→	Characteristic

A sequence of steps was performed for labelling SBVR based semantic roles to respective semantic predicates and their arguments given in a NL constraint. Following are the three main steps involved in the phase of semantic role labelling.

A. Identifying Predicates

In semantic role labelling, the primary step is the identification of the terms in a sentence that can be semantic predicates or predicate arguments. The semantic predicates and their respective predicate arguments are the basis of a logical representation as output of the semantic analysis in the NL2OCL approach. Once the semantic predicates and their respective predicate arguments are identified, each semantic predicate and predicate argument is annotated with a suitable semantic role. In the identification of predicates, the information extracted in the syntactic analysis phase plays a key role. Besides the syntactic information the researcher further needs to extract semantic features which can be helpful in the identification of a semantic predicate, a predicate argument and relations between predicate and arguments.

Following are the description of the approach used to extract predicates and their arguments.

i. Extracting Semantic Predicates: In this phase, we extract the possible semantic predicates. This module relies mainly on external resources, thus the elements in the target UML Class models (class names, attribute names, method names) are likely to be semantic predicates and predicate arguments. The chunks not matching the elements of the target UML Class model are not considered as semantic predicates or predicate arguments. For extracting semantic predicates we check for a simple verb, a phrasal verb or a verbal collocation and tag the verb phrase as a Verb Concept. A Verb Concept is a SBVR vocabulary and we map SBVRVerb Concepts to semantic predicates. An example of the extraction of a Verb Concept is shown in Figure 4.14.

[A] [customer] [cannot][$_{\text{Verb_Concept}}$place] [more][than][one][order][.]

Figure 4.14: Identifying Verb phrases

In English sentences, Verb Concepts are typically represented in a combination of auxiliary verb and main verb (possibly following participle). However sometimes, there are only auxiliary verbs and no main verbs.

ii. Extracting Predicate Arguments: A few statistical methods (based on FrameNet and PropBank) are available for the extraction of predicate argument [Giuglea, 2006]. However, statistical methods are typically less accurate in the extraction of predicate-argument structures on account of the data sparsity problem. However, the researcher proposes the use of decision tree as theyachieve high accuracy [Surdeanu, 2003] as compared to statistical methods. The researcher has used a simple decision tree that identifies predicate arguments on the basis of the use of pre-modifiers and post-modifiers in a sentence. Additionally the type of phrases also helps in identification of predicate arguments.

The use of pre-modifiers and post-modifiers is very common in English sentences. In a typical English sentence, the noun concepts are represented with a pre-modifier and/or a post-modifier. An example of such cases is shown in Figure 4.15:

[$_{\text{Pre-Modifier}}$The] [customer] [$_{\text{Post_Modifier}}$on the Chair].......[.]

Figure 4.15: English sentence with a prepositional phrase as a post modifier

In Figure 4.15 it is shown that an article (a determiner) can be a possible pre-modifier. A post-modifier such as prepositional phrases (see Figure 4.15), relative (finite and non-finite) clauses (see Figure 4.16 and Figure 4.17), and adjective phrases (see Figure 4.17) can also be used in a English sentence. Another example of noun concepts can have a pre-modifier such as adjective phrase and a post modifier such as relative finite clause as shown in the Figure 4.16.

[The] [$_{\text{Pre-Modifier}}$gold] [customer] [$_{\text{Post_Modifier}}$who applied for the account].......[.]

Figure 4.16: English sentence with an adjective phrase and a relative finite clause

[$_{\text{Pre-Modifier}}$The] [customer] [$_{\text{Post_Modifier}}$applying for an account].......[.]

Figure 4.17: English sentence with a relative infinite clause as a post modifier

By excluding the pre-modifiers and post modifiers, we can extract the noun concepts. For further classification of Object Types and Individual Concepts the POS type of the noun concept is checked. If the POS type is common noun, it is categorized as an *Object Type* and if the POS type is proper noun, it is categorized as an *Individual Concept*.

a. Processing Phrases: Once the noun concepts are extracted, the next phase is to process phrases to generate a semantic representation. We have identified three types of phrases in typical constraints as following:

Processing Phrases: Typical phrases are a combination of two or more words. In SBVR, both Object Types and Individual Concepts are represented in the form of phrases. The following two examples show how phrases are processed to a semantic representation:

English: credit customer

FOL: $\exists x$ isa (x, customer) ∧ Object_Type(x, credit)

English: gold credit customer

FOL: $\exists x$ isa (x, customer) ∧ Object_Type (x, credit) ∧ Object_Type(x, gold)

Generative Noun Phrases: The generative noun phrases are also very common in constraints. Especially the SBVR Characteristics are described by using generative noun phrases e.g. customer's age, customer's salary, etc. The following examples show the way the researcher has processed generative noun phrases to a semantic representation.

English: customer's account

FOL: $\exists x$ isa (x, account) ∧ Object_type(x, customer)

English: account of customer

FOL: $\exists x$ isa (x, account) ∧ Object_Type(x, customer)

Adjective Phrases: Adjective phrases are not common in constraints but the researcher has processed the adjective phrases as they can be a possible case. Following are the examples showing the processing of adjective phrases:

English: The customer is happy.

FOL: $\exists x$ isa (x, customer) ∧ Characteristic(x, happy)

English: This is a gold customer.

FOL: ∃x isa (x, customer) ʌ Characteristic(x, happy)

B. Sense Recognition

The researcher has identified various cases where a predicate or a predicate argument can be associated with more than one semantic role. In Table 4.3, the researcher has shown that a common noun can be mapped to an Object Type or as well as a Characteristic. For example, in Figure 4.18, it is shown that there are two common nouns (or predicate arguments): customer and name. However, one noun '*customer*' is an Object Type and other noun '*name*' is a Characteristic. By using the mappings given in Table 4.3, it is not possible to correctly identify the semantic roles for all common nouns.

[A] [$_{\text{NN}}$customer][enters] [his][$_{\text{NN}}$name] [.]

Figure 4.18: English sentence mapped with a UML class model

Similarly, it is also shown in Table 4.3 that a verb (predicate) can be mapped to a Verb Concept or a Fact Type, as in English a verb can be in the form of a simple verb, a phrasal verb or a verbal collocation. This case is very important to resolve, because if a verb is labelled as a *Verb Concept* then it will be mapped to navigation expression in OCL or else it is ignored. For example, in Figure 4.19, the token 'place' can be a *Verb Concept* or part of a *Fact Type*.

[A] [$_{\text{Class_name}}$customer][cannot] [$_{\text{Association_name}}$place][more than one][$_{\text{Class_name}}$order] [.]

Figure 4.19: English sentence mapped with a UML class model

These multiple mappings are due to semantic ambiguity. Typically, semantic ambiguities are due to the absence of context. Hence, to resolve the above discussed semantic ambiguities, the exact sense of the predicates and predicate arguments needs to be recognized so that accurate semantic roles may be assigned. The researcher proposes the use of information given in the target UML class model to identify the actual sense of a predicate or a predicate argument.

Table 4.4: SBVR based semantic role labels used in SRL

English Elements	UML Elements	SBVR based Semantic Roles
Common Noun	Class	Object Type
	Attribute	Characteristic
Proper Noun	Class	Individual Concept
Generative Noun, Adjective	Attribute	Characteristic
Verb	Method	Verb Concept
	Association	Fact Type

In Table 4.4, it is shows that each common noun is mapped to the target UML class model and if a common noun maps to a class, then it is represented as an Object Type or if a common noun maps to an attribute of a class, it is represented as a Characteristic. To solve the ambiguity of the NL constraint given in Figure 4.18, we use the UML class model given in Figure 4.20 where it is given that 'customer' is a class hence, 'customer' is tagged as `Object_Type` and `name` is an attribute if class `Customer` hence name is tagged as a `Characteristic`.

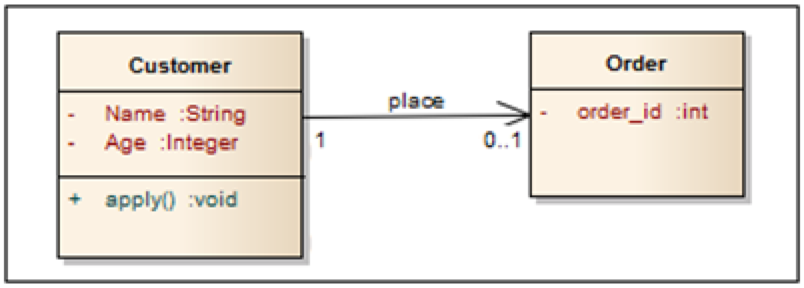


Figure 4.20: A UML class model involving a customer and an order class.

Table 4.4 also represents that if a verb maps to a method in the target UML class model, the verb should be tagged as `Verb_phrase`, else if a verb maps to an association in the target UML class model, the verb should be tagged as a `Fact_Type`. Similarly, there are two additional benefits of mapping a verb with an association as below:

i. A unary association or a binary association in a UML class model helps to identify that a fact type is a unary fact type or a binary fact type.
ii. Direction of the association helps in identifying the active and passive elements in a fact type.

For example, after mapping the NL constraint in Figure 4.19, we find that predicate arguments 'Customer' and 'Order' are mapped to classes `customer` and `order` respectively and there is a directed association between these two classes. The directed association shows that the 'Customer' is an agent or an actor and 'Order' is a patient or a thematic object. In the light of this information it is simple to identify that the predicate arguments should be like `place(customer, order)`. Another benefit of such mapping is that if English sentence in passive voice the same predicate will be generated e.g. `place(customer, order)`.

C. Role Classification

After sense recognition, the exact semantic label or semantic role is assigned to each substring in a sentence. The substrings are labelled with a semantic role. The used approach for semantic role labelling works as the syntactic tree representation of a sentence is mapped into a set of syntactic constituents. Finally, each syntactic constituent is classified into one of semantic roles. An example of classification of semantic roles is shown in Figure 4.21.

[A] [$_{\text{Object_Type}}$customer][cannot] [$_{\text{Fact_Type}}$place][more than one][$_{\text{Object_Type}}$order] [.]

Figure 4.21: English sentence mapped with a UML class model

The classification is performed on the basis of the sentence structural features or the linguistic context of the target constituent. Role classification is performed as the syntactic information (part of speech and syntactic dependencies) with predicate and predicate role set are given input and the output of this phase is semantic predicates and predicate arguments (see Figure 4.21) labelled with its corresponding roles.

4.3.2 Deep Semantic Analysis

In computational semantics, the key is understanding the complete meaning of a natural language sentence instead of focusing on text portions only. For the sake of computational semantics, we perform deep semantic analysis of the input text. Typically, deep semantic analysis involves generation of a fine-grained semantic representation from the NL text. Traditionally, various aspects are involved in deep semantics analysis. However, we are interested in a most commons aspect such as identifying quantifications, quantification scope resolution, and processing semantic logical operators. Following is the description of all these three steps.

A. Identifying Quantifications

Quantification is a very common part of a natural language sentence. In NL sentences, quantifications are typically expressed with noun phrases (NPs). Similarly, in First-Order Logic (FOL), the variables are quantified at the start of the logical expressions. Generally, the natural language quantifiers are much more vague and varied. This vagueness makes translation of NL to FOL complex. However, the researcher has done two things to handle quantifiers variable scoping.

With respect to the researcher's target representation (SBVR rules), he has identified the following four types of quantifications that he needs to handle, as SBVR 1.0 also support these four types of quantifications. The first two quantifications such as universal quantification and existential quantification are commonly used. However, these two types do not cover all possible types in detail. The researcher covers all possible types of quantifications in natural languages. Besides, the researcher has used two other types such as uniqueness and solution quantification. Hence, it will be simple to map these NL quantifications to SBVR quantifications.

***i. Universal Quantification* (∀X):** The universal quantifier is represented using all sign "∀" and means all the objects X in the universe. The universal quantification is mapped to Universal Quantification in SBVR. The NL quantification structures 'each', 'all', and 'every' are mapped to universal quantificational structures. Similarly, the determiners 'a' and 'an' used with the subject part of the sentence can be treated as universal quantification due to the fact that the researcher is processing constraints and generally constraints are mentioned for all the possible X in a universe (see Figure 4.22). However, the researcher has addressed the role of determiners as quantifiers in next section of quantifier scope resolution.

***ii. Existential Quantification* (∃X):** The existential quantifier is represented using *exists* sign "∃" and means at least one object X exists in the universe. The existential quantification is mapped to *Existential Quantification* in SBVR. The keywords like many, little, bit, a bit, few, a few, several, lot, many, much, more, some, etc. are mapped to existential quantification.

***iii. Uniqueness Quantification* ($\exists_{=1}$ X):** The uniqueness quantifier is represented using "$\exists_{=1}$" or "$\exists_{!}$" means exactly one object X in the universe. The uniqueness quantification is mapped to *Exactly-One* Quantification in SBVR. The determiners 'a' and 'an' used with object part of the sentence are treated as uniqueness quantification. However, we have addressed the role of determiners as quantifiers in next section of quantifier scope resolution.

***iv. Solution Quantification* (§X):** The solution quantifier (Hehner, 2004) is represented using section "§" sign and means *n* object in the universe. The solution quantification is mapped to *Exactly-n* Quantification in SBVR. If the keywords like more than or greater than are used with *n* then solution quantifier is mapped to *At-most Quantification* (see figure 8). Similarly, if the keywords like less than or smaller than are used with *n* then solution quantifier is mapped to *At-least Quantification*.

[$_{\text{Universal_Quantification}}$A] [$_{\text{Object_Type}}$customer][cannot] [$_{\text{Fact_Type}}$place]
[$_{\text{at_least_n_Quantification}}$more than one][$_{\text{Object_Type}}$order] [.]

Figure 4.22: Semantic roles assigned to input English sentence.

Two other types of quantifications are also available such as Plaucal quantification ($\exists^{\text{many}}$X) and mutal Quantification ($\exists^{\text{few}}$X). However, we are not using these both quantifications as both of them are not supported by SBVR and UML and ultimately can't be translated to OCL.

B. Quantifier Scope Resolution

In quantification resolution, the second issue is quantifier scope resolution. For quantification variable scoping, the researcher has treated syntactic structures as logical entities. However, in a few cases quantifier scope resolution is difficult due to ambiguities. Context plays an important role in resolution of such ambiguities in scope of quantifiers [Villalta, 2007]. As UML class models are the scope of a constraint, the information given in a UML class model such as multiplicity of associations is used. For this purpose, we have used the following algorithm

1. Each noun after a determiner is mapped to the class names in the input UML class model.
2. If a noun is found in the UML class model, the associations in this set of classes are analysed and checked for the associated multiplicity with the noun.

3. If the multiplicity is found for the noun, map the cardinality to quantification as

   a. If multiplicity is *, then quantification should be universal ($\forall X$)

   b. If multiplicity is 1, then quantification should be uniqueness ($\exists_{=1}X$)

   c. If multiplicity is 1..*n*, then it should be existential quantification ($\exists X$)

   d. If multiplicity is 0..*n* or *n*, then it should be solution quantification (§X)

Algorithm 4.2: An algorithm to handle attachment ambiguity

To address the determiner 'a' used with noun 'customer' in above example (see Figure 4.21) can be solved by using Algorithm 4.2 and the information available in the UML class model shown in Figure 4.20. As the associated multiplicity with 'customer' class in the given UML class model is '1' and algorithm's step 4.b says, it should be expressed as uniqueness ($\exists_{=1}X$) quantification. In a case if UML model disagrees with NL, user is given a message that he should recheck the given NL input.

[$_{\text{Solution_Quantification}}$A] [$_{\text{Object_Type}}$customer][cannot] [$_{\text{Fact_Type}}$place] [$_{\text{at_least_n_Quantification}}$more than one][$_{\text{Object_Type}}$order] [.]

Figure 4.23: Semantic roles assigned to input English sentence.

Figure 4.23 shows the NL constraint after handling the scope of quantifier represented by a determiner.

C. Processing Negation and Implication

We described the processing of conjunctions and disjunctions in Section 4.2.3. However, there are some other important logical operators typically involved in natural language constraints are negation and implication. These two types of logical operators cannot be processed just using the syntactic information. For processing negation and implication, semantic information is also required.

i. Negation: Negation is an important construct that is used to negate a structure by using keywords *no* and *not* e.g. "A customer cannot apply for more than one account." Here, negation has been used to restrict customers to a single account. We have also worked out the double negation as a positive sentence. Hence, $\neg(\neg p) = p$. Another possible way of representing a negation in a natural language is negative adjective, e.g., unhappy, bad, etc. Since, adjectives are not common part of constraints, currently, our approach does not support negative adjectives. However, if the user exclusively mentioned no/not in a NL sentence, our approach is able deal with such cases.

ii. Implication: In English, a few expression are used to represent implications such as "if p, then q", "if p, q", "q if p", "p only if q", "p implies q", "p entails q", "p hence q", "q provided p", "q follows from p". For example "If a student is adult, he can get a pass." The researcher has also identified that some expressions such as "q since p", "since p, q", "because p, q", "q because p", "p therefore q" are not true cases of implications. A set of rules were devised to handle possible types of implications in NL constraints.

4.3.3 Semantic Interpretation

In lexical semantics, the frame is also considered a useful tool in text semantics and the semantics of grammar. The interpreter of a text invokes a frame when assigning an interpretation to a piece of text by placing its contents in a pattern known independently of the text. A text evokes a frame when a linguistic form or pattern is conventionally associated with that particular frame. A simplified template that is used to generate a logical representation for a NL constraint is shown in Figure 4.24:

Template:
```
(Predicate_Name
    (Semantic_Role = (Quantification ~(Predicate_Arg ? Var))) ...)
```

Figure 4.24: Logical Representation of a NL constraint

The semantic representation shown in Figure 4.25 is enriched from of first-order logic. Besides Predicate name and variables, we have added some extra information such as predicate type (such as Object Type or Individual Concept), extra quantification (such as Solution quantification, Uniqueness quantification). By adding such extra information in typical first-order logic, we can prepare the logical representation more useful. Figure 4.24 shows a template used to create the logical representation. The template has 5 elements that constitute a complete logical representation such as `Predicate_Name` (that is Verb Concept), `Semantic_Role` (that is Object Type, Individual Concept or Characteristic),`Quantification`(Universal, Existential, Solution or uniqueness quantification), `Predicate_Arg` (that is predicate argument), and `var` (that is a variable name).

The researcher has written a rule based module that uses the generalized template shown in Figure 4.24 that generates a logical representation based on SBVR vocabulary. The output of this module is a SBVR based logical representation (is shown in Figure 4.25).

Semantic Interpretation:
```
(place
                    (Object_Type = (∃=1X ~ (customer ? X)))
                    (Object_Type = (§Y ~ (order ? Y))))
```

Figure 4.25: Logical Representation of a NL constraint

4.4 Generating SBVR Rule Representation

Once the logical representation is extracted from a NL constraint, the next phase is to generate a SBVR rule representation from the SBVR vocabulary and the logical representation extracted in previous phases. Here, two types of SBVR rules can be generated: structural rule or behavioural rule on the basis of type of information represented in the NL constraint. A typical SBVR rule is generated in three phases as below.

4.4.1 Constructing SBVR Rules

To generate a SBVR rule from NL constraint, it is primarily analysed that it is a structural rule or a behavioural rule. Following are two types of SBVR rules those can be classified on the basis of the various features of a NL constraint:

A. Generating Structural Rules

The use of auxiliary verbs such as ‘is’, ‘has’, etc. is identified to classify a NL constraint as a SBVR structural requirement. A sentence representing a state of being, e.g., “Robby is a robot” or a state of possession, e.g., “robot*has*two arms” can be categorized as structural requirement.

B. Generating Behavioural Rules

The use of auxiliary verbs such as ‘should’, ‘must’ are identified to classify requirement as a behavioural rule. In case of “should have” or “must have”, our parser looks for “should” only, additional "have" will have no impact. Moreover, the use of action verb can be categorized as a behavioural rule, e.g., "robot *picks up* parts". The example discussed in Figure 4.23 is mapped to a SBVR Fact Type (see Figure 4.26) that becomes base of a SBVR rule generated in next phase.

customer *cannot place* order.

Figure 4.26: SBVR Fact Type generated from NL constraint.

4.4.2 Applying Semantic Formulation

Once a raw representation of a SBVR rules is generated, a set of semantic applications are applied to comply with SBVR standard. A set of semantic formulations are applied to each fact type to construct a SBVR rule:

A. Apply Modal Formulation

Modal formulation (OMG, 2008) specifies seriousness of a constraint. Modal verbs e.g. '*can*' and '*may*' are mapped to possibility formulation while the modal verbs '*should*', '*must*' or Verb Concept "*have to*" are mapped to obligation formulation. Description of various types of modal formulations is given in Chapter 2, Section 2.1.2-C. Figure 4.27 shows that a modal formulation is applied on the SBVR rule generated in Figure 4.26 by adding a phrase 'It is possibility' and this phrase is concatenated to SBVR rule using a word 'that'.

It is possibility that customer *cannot place* order.

Figure 4.27: Applying modal formulation to core SBVR rule.

B. Apply Logical Formulation

A SBVR rule can have multiple fact types using logical operators [OMG, 2008] e.g. AND, OR, NOT, implies, etc. In logical formulation, the tokens '*not*' or '*no*' are mapped to negation($\neg$ a). Similarly, the tokens '*that*' and '*and*' are mapped to conjunction (a $\wedge$ b) and token '*or*' is mapped to disjunction (a $\vee$ b) and the tokens '*imply*', '*if*, '*infer*' are mapped to implication (a $\Longrightarrow$ b).

C. Apply Quantification

Quantification [OMG, 2008] is used to specify the scope of a concept. Quantifications are applied by mapping tokes like "*more than*" or "*greater than*" to at least n quantification; token "less than" is mapped to at most n quantification and token "*equal to*" or a positive statement is mapped to exactly n quantification. Figure 4.28 shows that quantification is applied on the core SBVR rule (generated in Figure 4.26) by adding two quantifications: quantification is added to the 'customer' Object Type that is 'exactly one' and second quantification is added to the 'order' Object Type.

It is possibility that exactly one customer cannot place at most one order.

Figure 4.28: Applying quantification to complement SBVR rule.

4.4.3 Applying Structured English Notation

The last step in generation of a SBVR rule is application of the Structured English notation. To apply Structured English notation, the Object Types are printed in bold frame and underlined e.g. **customer**; the Verb Concepts are italicized e.g. *cannot place*; the SBVR keywords are printed in bold frame e.g. **at most**; the Individual Concepts are underlined e.g. Patron. Similarly, the Characteristics are also italicized and underlined with a different colour: e.g. *name*. An example of a SBVR rule with Structured English notation is shown in Figure 4.29.

It is possibility that **exactly one** **customer** *cannot place* **at most one** **order**.

Figure 4.29: Semantic roles assigned to input English sentence.

4.5 Summary

In this chapter, the researcher has explained various steps involved in generation of SBVR rules representation from NL constraints. To achieve this goal, the NL constraints were processed using typical phases of NLP phases. However, the researcher came across various cases with NL constructs involving syntax ambiguities(such as lexical ambiguity, and attachment, ambiguity) and semantic ambiguities. The researcher also developed a novel approach to handle the identified types of ambiguities in NL constraints. The used approach to address syntactic and semantic ambiguities involves the information from the target UML class model that is the context of the information given in the NL constraints. An additional benefit of mapping information (given in NL constraints) with UML class model is that the finally generated SBVR and OCL will also be semantically consistent with UML class model. Here, semantic consistency means that the generated SBVR rules or OCL constraints will not have any information that is not the part of the target UML class model.

There are two contributions of this chapter: (1) Resolution of syntactic and semantic ambiguities (2) SBVR Intermediate representation. The approach used to resolve syntactic ambiguities was also presented in [Bajwa, 2011a]. Similarly, The approach used to resolve semantic ambiguities was also presented in [Bajwa, 2011c]. By resolving syntactic and semantic ambiguities, more accurate OCL constraints can be generated. Other major contribution is SBVR based intermediate representation that can not only be simply transformed to OCL but also other important formal languages such as Alloy, B, BPMN, BPML, etc.

CHAPTER 5
MODEL TRANSFORMATION FROM SBVR TO OCL

In the previous chapter, the researcher explained the generation of a logical representation and a SBVR rule representation from NL constraints. In this chapter, the transformation from SBVR to OCL is explained. The SBVR based logical representation and SBVR rule generated in the previous chapters are mapped to OCL representations using model transformation technology. For a model transformation of SBVR to OCL, the researcher needs to do two things to generate OCL constraints, as explained below:

i. Choose an appropriate OCL template (such as invariant, pre/post conditions, collections, etc.) from given set of templates.

ii. Map source elements of the logical form to the equivalent elements in the used OCL templates.

In the remaining part of this chapter, the researcher explains the transformation from SBVR to OCL using a set of transformation rules and templates.

5.1 OCL Templates

The researcher has designed generic templates to generate traditional OCL expressions: OCL invariant, OCL pre-condition, and OCL post-condition. One of the given three templates is chosen automatically on the basis of the type of SBVR rule. For a SBVR structured rule, the template for OCL invariant is chosen, while templates for OCL pre/post conditions are chosen if the SBVR rule is a behavioural rule. Once a template is chosen for one of the constraints, the missing elements in the template are extracted from the logical representation of English constraint. Figure 5.1 shows the template used to generate an OCL invariant:

```
package [UML-Package]
context [UML-Class]
inv: [Body]
```

Figure 5.1: Template for OCL invariant

To generate an OCL invariant, we need three things: a *UML-Package* that is a `package` of the invariant, a *UML-Class* that is `context` of the invariant and the *Body* of the invariant.

Figure 5.2 shows the template we used for OCL pre-condition. To generate an OCL pre-condition, first thing needed is *UML-Package* that is `package` of the pre-condition. Additionally, a `context` is required that is composed of a *UML-Class* (a UML class), a *Class-Op* (an operation or method of the UML class used) with *Param* (set of parameters of the related class operation) and the *Return-Type* (return type of the used class operation). Finally, to complete the pre-condition *Body* is also required.

```
package [UML-Package]
context [UML-Class::Class-Op(Param):Return-Type]
pre: [Body]
```

Figure 5.2: Template for OCL pre-condition

The template the researcher used to generate an OCL prost-condition is shown in Figure 5.3. Similar to OCL pre-condition, to generate an OCL post-condition, the first thing needed is a *UML-Package* that is `package` of the post-condition. Additionally, a `context` is also required that is composed of a *UML-Class* (a UML class), a *Class-Op* (an operation or method of the UML class used) with *Param* (set of parameters of the related class operation) and the *Return-Type* (return type of the used class operation). To complete the post-condition *Body* is also required. Finally, another element `result` is also involved. However, the `result` is optional in post-conditions.

```
package [UML-Package]
context [UML-Class::Class-Op(Param):Return-Type]
post:  [Body]
result: [Body]          -- optional
```

Figure 5.3: Template for OCL post-condition

In the all shown templates, elements written in brackets '`[ ]`' are required. The researcher gets these elements from the logical representation of English sentence. The following mappings are used to extract these elements:

i. *UML-Package* is the package name of the target UML class model.

ii. *UML-Class* is the name of the class in the target UML Class model and *UML-Class* should also be an Object Type in the subject part of the English Constraint.

iii. *Class-Op* is one of the operations of the target class (such as context) in the UML Class model and *Class-Op* should also be the Verb Concept in English constraint.

iv. *Param* is the list of input parameters of the *Class-Op* and they are retrieved from the UML class model. These parameters should be of type Characteristics in English constraint.

v. *Return-Type* is the return data type of the *Class-Op* and they are retrieved from the UML class model. The return type is the data-type of the used Characteristic in English constraint and this data type is extracted from the UML class model.

vi. *Body* can be a single expression or combination of more than one expression. The details of *Body* are given in the next section. Here, for each type of expression in the body, the researcher designed a small template.

In contrast to Wahler's approach [Wahler, 2008], where he used large templates to generate a complete OCL constraint, the researcher has small templates those generate small expressions and the researcher has integrated all those expression to form body of an OCL constraint. To integrate the generated expressions, the researcher has used the relationships given in the SBVR based logical form generated in Chapter 4, Section 4.3.3. Such small expressions are based on the relations that the researcher has explained in the form of tables, explained in the next section.

5.2 Mapping SBVR based Logical Form to OCL

The *Body* of the invariants and the pre/post-conditions is generated from the logical form generated in Section 4.3.3. A set of transformation rules [Bajwa, 2011b] were used to transform a SBVR based logical representation to OCL constraints by mapping element(s) of the SBVR metamodel to equivalent element(s) in the OCL metamodel. These rules are explained in Section 5.3.

For SBVR to OCL transformation, a model-to-model transformation is used for automated transformation of SBVR rules to OCL invariants. A typical model transformation technology is used by creating abstract syntax of source model and then converting it into the target model representation using the transformation rules. Here, SBVR metamodel, OCL metamodel and a set of transformation rules are used to perform the transformation of SBVR to OCL.The OCL metamodel is explained in Section 2.1.2 and the SBVR metamodel is explained in Section 2.2.2. For SBVR to OCL transformation, the *SiTra* library was used as shown in Figure 5.4:

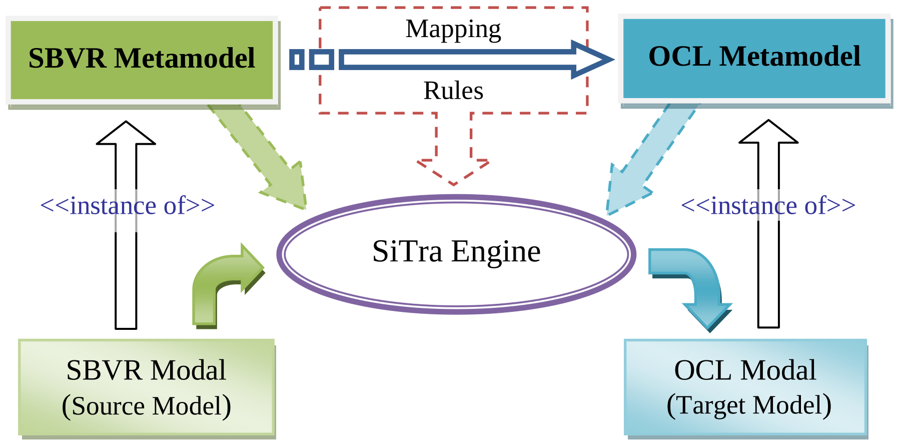


Figure 5.4: SBVR to OCL Transformation Framework

Moreover a set of mappings were used to map logical elements to OCL elements. Following is brief overview of the used mappings from logical representation to OCL:

5.2.1 Logical Expression

In OCL, two expressions are concatenated using a logical operator. Here, the SBVR representation is mapped to the equivalent OCL representation Table 5.1 shows the possible cases of logical expressions:

Table 5.1: Mapping logical expressions

SBVR Representation	OCL Representation
p or q	`p or q`
p and q, p but q, p yet q, p so q, p moreover q, p however q, p although q, p even though q	`p and q`
p then q, if p q, q if p, q only if p, p implies q, p entails q, p hence q, q provided p, q follows from p	`p implies q`

5.2.2 Relational Expressions

In OCL, two expressions can be concatenated using a relational operator. Table 5.2 shows the possible cases of relational expressions:

Table 5.2: Mapping relational expressions

SBVR Representation	OCL Representation
p is q, p is exactly q	`p = q`
p is at least q	`p > q`
p is at most q	`p < q`
p is at least or exactly q	`p >= q`
p is at most or exactly q	`p <= q`

5.2.3 Postfix Expressions

In OCL, there can be a postfix expression such as self. A possible mapping of postfix expressions is shown in Table 5.3. An example of such cases can be "name of a customer" is mapped to `self.name` if customer is a context.

Table 5.3: Mapping postfix expressions

SBVR Representation	OCL Representation
Characteristic of the Object Type	`self.`[*attribute*]

5.2.4 Navigation

Navigation expressions are the most common expressions in OCL. A possible mapping of navigation expressions is shown in Table 5.4. For example, "name of a customer" is mapped to `customer.name` and if customer is a context then it is mapped as `self.name`. Some other expressions of OCL operations are also shown in Table 5.4. We have implemented a selected set of OCL operations those are commonly used. Implementation of the remaining OCL operations is future work.

Table 5.4: Mapping navigation expressions

SBVR Representation	OCL Representation
p's q, q of P	`p.q`
p is q()	`p.q()`
size of p, number of p	`p->size()`
number of p in q	`p->count(q)`
p is empty, no p, zero p	`p->isEmpty()`
sort p, arrange p	`p->sortBy()`
q exists in p, there is q in p	`p->exists(q)`

5.2.5 Conditional Expression

In OCL, there can be a conditional expression. The conditional expressions can be of two types: if-then expressions, and if-then-else expressions. Table 5.5 shows a possible mapping of conditional expressions. Here, *Relational-Exp* are conditions of the `if` structure and these conditions are generated using the mappings given in Table 5.2.

Table 5.5: Mapping conditional expressions

SBVR Representation	OCL Representation
if p then q	`if` [*Relational-Exp*] `then`***q*** `endif`
if p then q else R	`if` [*Relational-Exp*] `then`***q*** `else`***R*** `endif`

5.3 SBVR to OCL Transformation Rules

The researcher presents an automated approach for the transformation of the SBVR based logical representation to OCL constraints. This approach not only softens the process of creating the OCL syntax but also verifies the formal semantics of the OCL expressions with respect to the target UML class model. Here, verification of formal semantics mean that the generated OCL constraints semantically complies with the target UML class model and the OCL expression will not have any extra contextual (UML class model) information. As OCL should comply with its context, it is necessary to verify an OCL expression against the target UML model. The *Si*T*ra* library [Akehurst, et al., 2007] based model transformation framework is shown in Figure 5.4 to transform SBVR rules to OCL constraints using a set of transformation rules.

The mapping of SBVR rules to OCL code is carried out by creating different fragments of OCL expressions and then concatenating these fragments to compile a complete OCL expression. Typically, OCL expressions can be of two types: OCL invariant and OCL query operation. In this thesis, the researcher has presented only the creation of OCL invariants. The creation of OCL query operation is part of the future work.

It is possibility that **exactly one** **customer** *can place* **exactly one** **order**.

It is possibility that exactly one `p` can `function_1()` exactly one `q`.

Figure 5.5: Applying quantification to complement SBVR rule

The SBVR rule and logical representation, created in the previous chapter, is further analysed to extract the business information as shown in Figure 5.5. Where, `p` is Object Type "customer". Similarly, `q` is also an Object Type "order". However p and q can also be Individual Concepts in some other examples. Additionally `function_1()` is verb phrases "place". The analysed SBVR rule is further transformed to the logical representation after omitting the SBVR keywords as following

$$p \Rightarrow q \text{ if } p.\mathit{function\_1}() \rightarrow q.\mathit{size}() = 1$$

This generalized representation is finally transformed to the OCL constraint by using the defined transformation rules. A typical transformation rule comprises of the variables, predicates, queries [Akehurst, et al., 2007], etc. A typical transformation rule consists of two parts: a left-hand side (LHS) and a right-hand side (RHS). The LHS is used to access the source model element whereas the RHS expands it to an element in the target model.

In this Chapter, the transformation rules for each part of the OCL constraints are based on the abstract syntax of SBVR and OCL that are given in the following section. Here, all the transformation rules are represented in the form of functions.

5.3.1 Generating OCL Context

The context of an OCL expression defines the scope of the given invariant or pre/post condition. To specify the context of an OCL invariant, the major `Object_Type` or `Individual_Concept` in the SBVR rule is extracted to specify the context. To specify the context of an OCL pre/post condition, the action performed by the actor in a SBVR rule is considered as the context. If the context is missing in the NL constraint, the user is given a message to include at-least one class in NL constraint that can work as a context. Rule 5.1.1 shows the OCL context for invariant expressions and Rule 5.1.2 shows the context of for the pre/post condition of an operation:

```
context-inv(Object_Type){
                        context-name= Object_Type
                        return context-name
                   }
```

Rule 5.1.1: Returns the context for an invariant

```
context-cond(Object_Type, Verb_Phrase){
                  context-name = Object_Type
                  operation-name = Verb_Phrase
                  return context-name + “::” + operation-name
                 }
```

Rule 5.1.2: Returns context for a pre/post condition

Precondition and post-condition can co-exist in a single OCL expression. However, the both precondition and post-condition can share the same context.

5.3.2 Generating OCL Constraints

Transformation rules for mapping of the SBVR specification to OCL constraints are defined in this section. As, we explained above that OCL constraints can be of three types: invariants precondition and postcondition, we have defined three templates separately for each type OCL constraints. Rule 5.2.1 shows the template used for OCL invariant. Similarly, rules for OCL precondition and postcondition are described in Rule 5.2.2 and Rule 5.2.3, respectively. Each of the rules for these constraints consists of two elements: context of the constraint and body of the constraint.

```
invariant( context-inv, inv-body ){
          return  “context” + context-inv + “\n inv:”  + inv-body
          }
```

Rule 5.2.1: Returns an invariant

```
pre-cond(context-cond, pre-cond-body){
          return “context” + context-cond + “\n pre:” + pre-cond-body
          }
```

Rule 5.2.2: Returns a precondition

```
pre-cond(context-cond, post-cond-body){
          return “context” + context-cond +
                 “\n post:” + post-cond-body
          }
```

Rule 5.2.3: Returns a postcondition

5.3.3 Generating OCL Invariants

The OCL invariant specifies a condition on a class's attribute or association. Typically, an invariant is a predicate that should be TRUE in all possible worlds in UML class model's domain. The OCL context is specified in the invariants by using the `self` keyword in place of the local variables.

```
inv-body(ocl-exp){
                 return "inv:" + ocl-exp
                 }
```

Rule 5.3.1: Returns body of an invariant

An invariant can be expressed in a single attribute or set of attributes from a class. There can be three type of expressions in a typical OCL invariant; a general expression, a collection expression or `if` expression (see Rule 5.3.2). A collection expression is based on a single or set of collection operations those are used to perform basic operations on the set of attributes (see Rule 5.3.3). Other possible expressions are if expression (see Rule 5.3.4).

```
ocl-exp(){
           Return "self." + ( Expression |collection-exp | if-exp )
         }
```

Rule 5.3.2: Returns body of an invariant with `self`-keyword

```
collection-exp(Expression){
          Return Expression →  collection-op |
      Expression →  collection-op → collection-exp | ""
        }
```

Rule 5.3.3: Returns collection expression

```
if-exp(Condition, Expression-1, Expression-2){
        Return "If" + Condition + "then" + Expression-1 +
              "else" + Expression-2  + "endif"
         }
```

Rule 5.3.4: Returns if expression

5.3.4 Generating OCL Pre/Post Conditions

Similar to the OCL invariant, the OCL preconditions and the OCL post-conditions are used specify conditions on operations of a class. Typically, a precondition is a predicate that should be TRUE before an operation starts its execution, while a post-condition is a predicate that should be TRUE after an operation completes its execution.

```
pre-cond-body (ocl-exp){
        return "pre:" + ocl-exp}
```

Rule 5.4.1: Returns body of a pre-condition

```
post-cond-body( ocl-exp, value ){
        Return "post:" + ocl-exp | "post:" + result = value}
```

Rule 5.4.2: Returns body of a pre-condition

```
value( thematic-object ){
        Return Integer-value | Double-value |
               String-value | Boolean-value
      }
```

Rule 5.4.3: Returns body of a pre-condition

A pre/post condition can be expressed in a single attribute or set of attributes from a class. Rule 5.3.2 and Rule 5.3.3 are reused here to accompany Rule 5.4.1 and Rule 5.4.2. In Rule 5.4.3 the attribute value is verified that the provided value is of accurate type e.g. integer, double, or String, etc. Here, SBVR Structural rules can be mapped to invariants and SBVR behavioural rules can be mapped to pre/post-conditions

5.3.5 Generating OCL Expressions

The OCL expressions express basic operations that can be performed on available attributes of a class. An OCL expression in the OCL invariant can be used to represent arithmetic, and logical operations. OCL arithmetic expressions are based on arithmetic operators e.g. '+', '–', '/', etc, while, logical expressions use relational operators e.g. '<', '>', '=', '<>', etc and logical operators e.g. 'AND', 'implies', etc.

```
Expression( Expression ){
        Return prefix-oper + Expression | Expression  +
               infix-oper + Expression  | Expression
      }
```

Rule 5.5.1: Returns an expression

```
infix-oper(Quantification ){
        Return + | - | * | / | = | > | < | >= | <= |
<> | OR | AND |implies
      }
```

Rule 5.5.2: Returns an in-fix expression

```
prefix-oper( Quantification ){
        Return  -|NOT
      }
```

Rule 5.5.3: Returns a pre-fix expression

5.3.6 Generating OCL Operations

The OCL collections represent a set of attributes of a class. A number of operations can be performed on the OCL collections e.g. `sum()`, `size()`, `count()`, `isEmpty()`, etc.

```
collection-op( Expression ){
      forAll(Expression) | exists(Expression) |
      select(Expression) | allInstances(Expression) |
      include(Expression) | ….
     }
```

Rule 5.6.1: Returns a collection expression

SBVR specification of a business rules shown in the example of section 2 is mapped to an OCL constraints to show the working of the defined transformation rules. First of all to derive the context of the OCL constraint, transformation Rule 5.1.1 was used as we want to create OCL invariant. For pre/post conditions transformation Rule 5.1.2 will be used.

```
   Context customer
```

Figure 5.6: Generating a context

As the input SBVR rule is based on a ‘if-condition’, transformation Rule 5.3.4 will be used to derive an equivalent structure in OCL syntax. For if condition the Rule 5.5.1 and Rule 5.5.2 are employed. Following is the obtained structure:

```
    if c.age >= 18 then
    c.bankAccount -> size()>=1
    endif
```

Figure 5.7: Generating a if-expression

The ‘then’ part of the above shown if-expression involves a set of persons and such expressions are handled by Rule 5.6.1. Here, `size()` OCL operation is used to specify the quantification of bank accounts for a customer. Finally, to complete the OCL invariant expression, again the Rule 5.6.1 is used to derive following expression.

```
    inv: self.allSubTypes()-> forAll(c|…)
```

Figure 5.8: Generating an invariant

To construct a complete expression of an OCL constraint, all the generated constituents are concatenated into a single expression as following:

```
    context customer
     inv: self.allSubTypes()-> forAll(c| if c.age >= 18 then
          c.bankAccount -> size()>=1 endif)
```

Figure 5.9: Generating an invariant

All the model transformation rules were implemented using SiTra library. For example in Rule 5.5.1, the SBVR elements of a Boolean expression (represented as `BoolExp`in Figure 5.5.6) are mapped to equivalent OCL binary expression. For the sake of implementation, we pass three elements (operand1, operator, and operand2) to `BoolExpImpl()`.

```
  BoolExp bexpl = new
  BoolExpImpl(source.getOperand1(),source.getOperator(),source.getOper
  and2());
```

Figure 5.10: Implementation of Rule 5.5.1

The implementation of Rule 5.5.1 shown in Figure 5.10 involves two classes BoolExpImpl that implements interface BoolExpand BinaryExpressionImpl that implements interface BinaryExpression.

```
class BoolExpImpl implements BoolExp
class BinaryExpressionImpl implements BinaryExpression
```

Figure 5.11: Involved classes in implementation of Rule 5.5.1

Figure 5.11 shows that the implementation of Rule 5.5.1 is added into a vector bexp like other transformation rules in OclBinExp() method. The OclBinExp() method returns the OCL representation of the input SBVR elements.

```
private String OclBinExp(String operand1, String operator, String operand2){
      Vector<BinaryExpression> bexp = new Vector<BinaryExpression>();
      bexp.add(binaryExpression(operand1, operator, operand2));
      List<? extends BoolExp> binexp =
trans.transformAll(BinExpression2BoolExp.class,bexp);
      return  binexp.toString().substring(1, binexp.toString().length()-1);
}
```

Figure 5.12: Interface of Rule 5.5.1

All other transformation rules are implemented in the same manner as it is shown in above discussed example.

5.4 Limitations of Transformation

The researcher has mapped only those items of SBVR's meanings metamodel that had equivalent elements in OCL metamodel. For example, in SBVR's meanings metamodel. The Representation and its sub-elements Designation and Text are not mapped to OCL as in OCL metamodel, there is no equivalent of them. The reason is that Representation and its sub-elements deal with the placement of various SBVR vocabulary items in a SBVR business rules and it also provides structural representation to a SBVR rule. Similarly, there are five logical formulations given in SBVR 1.0 document and the researcher has mapped only those formulations that are involved in OCL invariants syntax such as Logical Formulation, and Quantification. The other three formulations such as atomic formulation, instantiation formulation, and modal formulation are not mapped as these formulations are related to the structure of a SBVR business rule, but do not carry any information that can be used in process of generating OCL invariants.

5.5 Summary

In this chapter, the framework used for transformation of SBVR to OCL is explained. The researcher explained the used templates for generating OCL invariants, OCL pre-conditions, and OCL post-conditions. The *S*i*T*ra based implementation of the SBVR to OCL transformation rules were explained. The use of transformation rules is also explained with the help of couple of examples. The presented SBVR to OCL is fully automated.

CHAPTER 6
TOOL SUPPORT

The NL2OCL*via*SBVR tool is the implementation of NL2OCL approach used to transform NL constraints to OCL via SBVR. In this chapter, the researcher presents the architectural and implementation details of the NL2OCL*via*SBVR tool. He also provides an overview of the used off-the-shelf components used in the implementation.

6.1 Architecture of the NL2OCL*via*SBVR

The NL2OCL*via*SBVR is a Java based implementation of the NL2OCL approach. The tool is available as an Eclipse plugin implemented using Eclipse Modelling Framework (EMF) [Steinberg, 2008]. In the previous chapters, the researcher has explained that the NL2OCL approach takes two inputs: a NL constraint and a UML class model as shown in Figure 6.1. A phase called NL processing was involved to analyse NL constraints syntactically and semantically and to extract SBVR vocabulary. Output of the NL processing phase is a SBVR vocabulary based logical representation that can be mapped to other formal languages. Such SBVR based logical representation is further processed to generate the SBVR rules. Finally, the SBVR rules are model transformed to OCL constraints using *Si*T*r*a [Akehurst, 2006] transformation engine.

Figure 6.2 illustrates the implementation of the NL2OCL approach. It is depicted in Figure 6.1 that there are two inputs: a NL constraint and a UML class model. The researcher has developed parsers for both inputs: a NL parser that parses NL constraint and an Ecore parser that parses the Ecore representation of the UML class model.

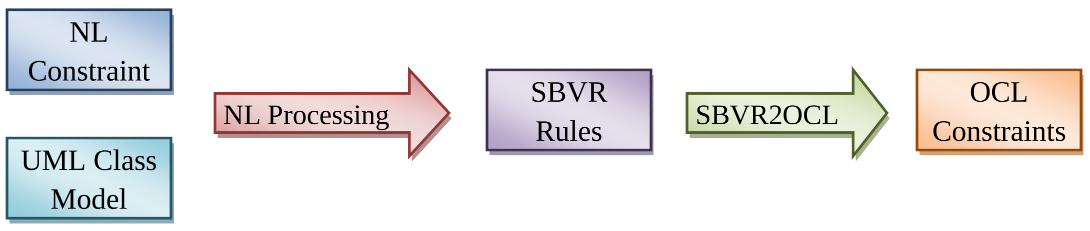


Figure 6.1: Overview of NL2OCL Approach

It is shown in Figure 6.2 that if there is ambiguity in NL text, an error message is given to the user to handle the situation manually, if not handled by the tool. Once the outputs of both parsers are received, both the output is mapped to each other to ensure that the information represented in the NL constraint is also the part of the UML class model. If the NL to UML mapping is successful, SBVR vocabulary is generated. In this process, if there is some inconsistency in NL constraint with respect to the UML class model, user is prompted about it. Once the SBVR vocabulary is available, a module *SBVR Generator* generates a SBVR rule by using the SBVR metamodel. Finally, a SBVR2OCL module generates OCL constraints by using OCL metamodel. Similarly, during OCL is generation, if NL constraints is incomplete or there is discrepancy, user is again given a message so that he may revise the NL statement to handle the case.

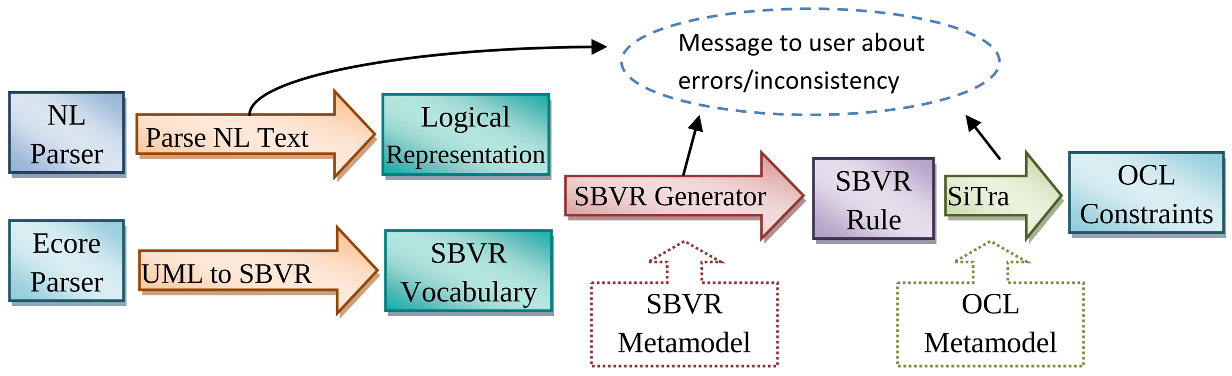


Figure 6.2: Overview of NL2OCL*via*SBVRImplementation

For implementation of the SBVR to OCL transformation, a set of transformation rules are defined (explained in Section 5.3). The Simple Transformer (*Si*T*r*a) transformation engine is used in SBVR to OCL transformation, as the researcher found it simple to use and implement with his approach. The SBVR and OCL metamodels are implemented in the Java programming language. Figure 6.3 shows a high level view of all the major components involved in the

NL2OCL*via*SBVR tool. There are eight major components involved in NL to OCL transformation. Out of eight components, there are two metamodels: OCL metamodel and SBVR metamodel. Then, there are four components which process NL constraints: Java Tokenizer, Java Sentence splitter, the Stanford POS Tagger, and the Stanford parser. One component involved is the Ecore parser that is typically involved in parsing the EMF Ecore format of a UML class model. Various Eclipse packages such as `org.eclipse.emf.ecore` are used in Ecore parser. Finally, the eighth component is the *Si*T*r*a transformation engine [Akehurst, et al, 2012] that maps the SBVR to OCL transformation rules. All these eight components work in combination to support NL to OCL transformation.

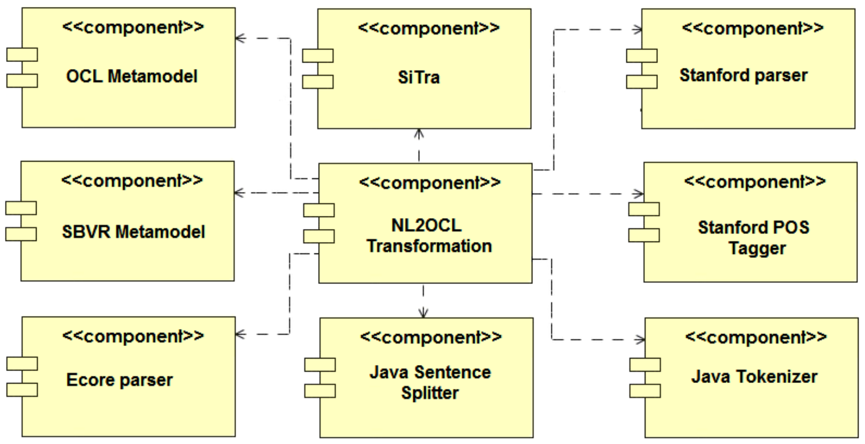


Figure 6.3: Libraries used by NL2OCL*via*SBVR

The rest of the chapter describes the implementation of the NL2OCL approach in more detail. The researcher has divided the implementation details into two phases for the sake of simplicity: implementation of NL2SBVR phase and implementation of SBVR2OCL phase. Detailed explanation of each phase is given below:

6.2 Implementing NL2SBVR

The NL2SBVR phase implements processing of NL constraints and generation of SBVR rules. The theory of this approach is given in chapter 4 and also published in [Bajwa, et al., 2011a]. As it is depicted in Figure 6.4, the implementation of the NL2SBVR approach consists of five sub-modules: pre-processor, syntax analyser, semantic analyser, semantic analyser, NL to UML mapping, and SBVR rule generator. An overview of these five modules is given below:

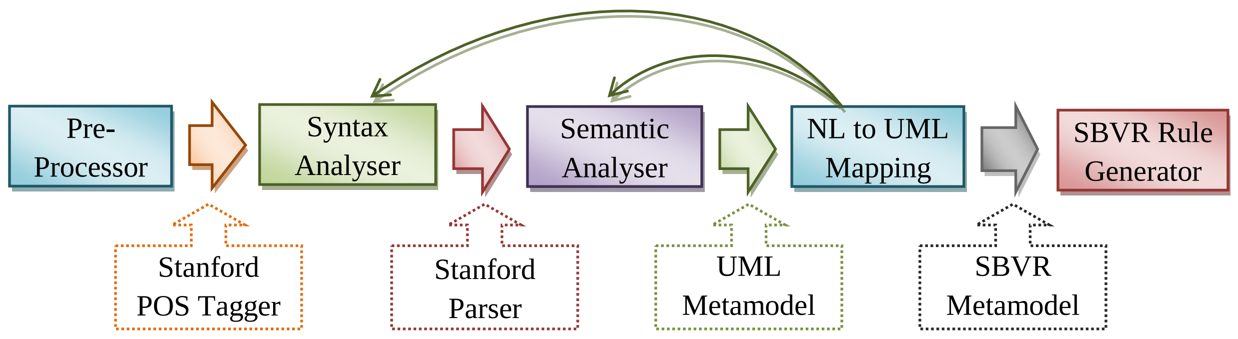


Figure 6.4: Overview of NL2OCL*via*SBVRImplementation

6.2.1 Pre-Processor

The Pre-processor is the primary module of the NL2SBVR phase. The pre-processor performs three basic steps: sentence splitting, tokenization and lemmatization. For sentence splitting, the researcher has used Java sentence splitter library to identify boundaries of each sentences in the given NL specification of constraints. Afterwards, Java tokenizer is used to identify tokens in each sentence. For lemmatization of each token, he has written a small rule based module to extract lemma of each token. Here, a pre-processor preserves both copies of a token (such as before lemmatization and after lemmatization) for detailed processing in later stages.

6.2.2 Syntax Analyser

The syntax analyser involves four sub modules: POS tagger, parse tree and dependencies generator, voice classifier, and logical operator handler. For POS tagging, the Stanford POS tagger is used. Similarly, to generate a parse tree and syntactic dependencies for the input NL constraint, the Stanford parser is used. The `jar` files of the Stanford POS tagger and the Stanford parser are integrated with the researcher's syntax analyser. However, a piece of code verifies that the output of the Stanford POS tagger and the Stanford parser are consistent with the input UML class model. This process helps in addressing lexical and attachment ambiguity in NL constraints. The detail of resolution of various types of syntactic ambiguities is given in Section 4.2.4 and Section 4.2.5 of this thesis and also discussed in [Bajwa, et al., 2012a].

To identify the voice of each sentence the researcher has written a rule-based classifier that classifies each sentence into active-voice or passive-voice category. Once the voice of sentence is identified, it is processed accordingly. The fourth and final sub-module of syntax analyser processes the logical operators (conjunction and disjunction) in the NL constraint. This module identifies right-hand side and let-hand side of a logical operator.

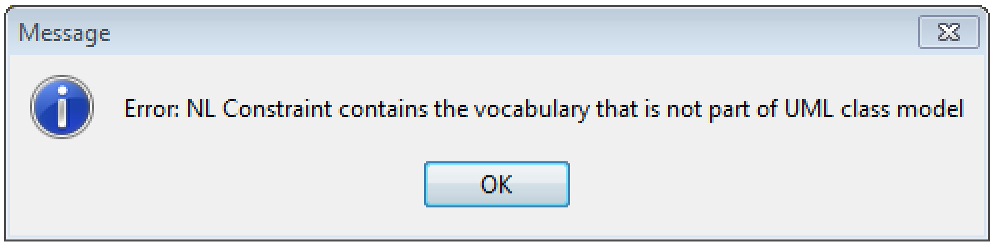
6.2.3 Semantic Analyser

The semantic analyser contains three sub-modules: semantic role labeller, quantification handler and logical representation generator. Here, each sub-module is implemented in a separate Java file. The semantic role labeller is a rule based module implemented in Java and identifies the SBVR based semantic roles (explained in Chapter 4, Section 4.3) for various parts of a NL constraint. The second module processes used NL quantifiers in the NL constraint and also resolves the quantifier scope. Here, the NL information is sent to the UML module to verify NL quantifications. The third module is also a rule-based component that generates the SBVR based logical representation. The third module uses the template for a logical representation (explained in Chapter 4), fills it, and generates a logical representation.

All these three modules are implemented as Java classes and are sequentially connected to each other as the output of one module is input of the next module.

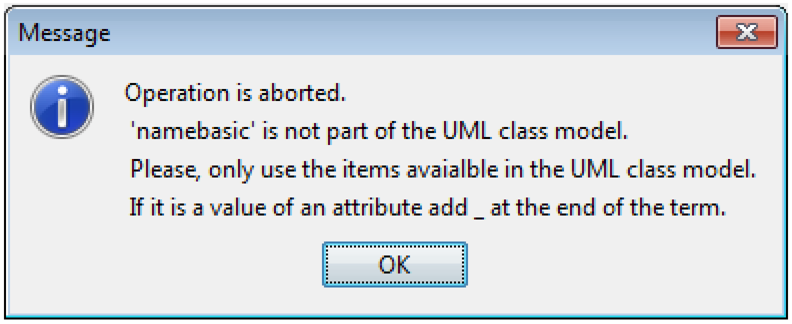
6.2.4 NL to UML Mapping

This module is a standalone component but it works in parallel with the other two modules: syntax analyser and semantic analyser. This module is based on an Ecore parser that can read an EMF Ecore file. The Ecore parser is a Java implementation that extracts metadata of a UML class model. We use the metadata of a UML class model to validate the output of the syntactic and semantic analyser. However, the output of the semantic analyser is mapped to the UML class model to validate that all information given in the NL constraint should also be part of the UML class model. Any piece of information that does not map with the target UML class model is omitted and does not become part of the SBVR rule representation generated by the next module. Moreover, the user is also a given an error message (see Figure 6.7 and Figure 6.8) so that he may revise NL statement. Figure 6.5 shows the error message for Constraint 7.2.13 and Figure 6.6 shows an error message for Constraint 7.2.15.


Message

Error: NL Constraint contains the vocabulary that is not part of UML class model

Figure 6.5: Error message shown to user in case of inconsistency


Message

Operation is aborted.
'namebasic' is not part of the UML class model.
Please, only use the items avaialble in the UML class model.
If it is a value of an attribute add _ at the end of the term.

Figure 6.6:Error message shown to user in case of extra vocabulary

6.2.5 SBVR Rule Generator

The fifth and final module generates a SBVR rules for each NL constraint. The SBVR rule generator module consists of two Java classes. First Java class takes the SBVR based logical representation and applies various semantic formulations such as quantification, modal formulation and logical formulations. A rule based algorithm identifies that what type of particular formulations should be applied on the basis of the nature of the sentence. Second Java class applies the structured English notation. The second Java class consists of a set of rule that applies particular formatting on the basis of the type of the SBVR vocabulary item.

6.3 Extending SiTra for SBVR to OCL Transformation

With the intention of implementation of SBVR to OCL transformation, a set of transformation rules were defined. SBVR to OCL transformation rules are implemented by extending the *SiT*ra transformation engine. The Simple Transformer (*SiT*ra) has been developed by Akehurst et al. [2008] and is a simple and lightweight implementation of an extensible transformation engine.

The*SiT*ra framework involves two interfaces typically used in *SiT*ra transformation framework: the `Transformer` interface and the `Rule` interface (explained in Chapter 2, Section 2.1.4).

The `Transformer` interface provides the skeleton of the methods to achieve the transformation. The `Transformer` interface consists of two key methods: the `transform()` method and the `transformAll()` method. On the other hand, the `Rule` interface is a set of mapping rules which the researcher has implemented according to the SBVR to OCL transformation rules. We have defined such transformation rules in Chapter 5, Section 5.3. However, the use of the *Si*T*ra* library is very simple as modeller needs to implement the `Rule` interface by using defined set of transformation rules. The `Rule` interface consists of three methods as explained Section 2.1.3. The first method is `check()` that is involved in the rule interface. The second method `build()` method is executed to generate the target model element. The third method `setProperties()` is involved in setting the attributes and links of the newly created target element.

6.4 Off-the-shelf components used in NL2OCL*via*SBVR

The NL2OCL*via*SBVR is composed of fifteen small modules as shown in Table 6.1. Each module performs a distinct functionality. To perform a few functionalities the researcher has used off-the-shelf components which help to generate accurate OCL constraints from NL specification.

Table 6.1: Overview of the components used in the NL2OCL*via*SBVR

	Component Functionality		Component Type
1	Text Tokenization	→	Java Tokenizer
2	Sentence Splitting	→	Java Sentence Splitter
3	Part-Of-Speech (POS) Tagging	→	Stanford POS Tagger
4	Resolving Lexical Ambiguity	→	Self-Developed
5	Generating Parse Tree & Dependencies	→	Stanford Parser
6	Resolving Attachment Ambiguity	→	Self-Developed
7	Voice Classification	→	Self-Developed
8	Processing Logical operators	→	Self-Developed
9	Parsing UML Class model	→	Self-Developed
10	SBVR based Semantic Role Labeling	→	Self-Developed
11	Quantifier Scope Resolution	→	Self-Developed

12	Generating Logical Representation	→	Self-Developed
13	Generating SBVR Rules	→	Self-Developed
14	Applying Structured English notation	→	Self-Developed
15	SBVR to OCL Transformation	→	Extended version of SiTra Library

Table 6.1, all fifteen modules are represented including text tokenization, sentence splitting, Part-Of-Speech (POS) tagging, generation of parse tree, the (typed) dependencies, and transformation of SBVR to OCL. In the first four modules, off-the-shelf components are involved. However, the last module that performs the SBVR to OCL transformation uses the extended version of *Si*T*ra* framework. The Stanford POS Tagger and the Stanford Parser are explained in Section 2.1.3 and the *Si*T*ra* framework is explained in Section 2.1.4 of this thesis. Hence, out of fifteen modules, five modules are based on the off-the-shelf components. We have developed the remaining ten modules by ourselves and integrated these ten modules with the five off-the-shore based modules.

6.5 Architecture of the NL2OCL*via*SBVR

The researcher has discussed the list of all the components involved in the NL2OCLviaSBVR tool. A sequence of functionalities is performed here and to perform each function, one or more than one components are involved. Figure 6.8 shows architecture of the NL2OCL*via*SBVR tool. In Figure 6.8, mainly, the researcher has used three types of boxes: dotted-line arrow boxes, dotted-line boxes, and solid-line boxes. Here, dotted-line arrow boxes represent inputs or outputs with the help of various colours while the dotted-line boxes represent off-the-shelf components involved in the transformation. Similarly, the solid-line boxes represent the components, the researcher has developed himself.

It is shown in Figure 6.8, that there is GUI layer, which helps a user to communicate with the system and receive errors/problems/inconsistencies messages. A user can give two inputs to the systems (such as NL constraint and UML class model) and can get output from the system using GUI. Once a user gives both the inputs, NL constraint is processed by NL components with the help of Java libraries and the Stanford parser while the UML parser extracts metadata from the UML class model. Afterwards, the SBVR generator and the OCL generator modules process input with the help of SBVR and OCL metamodel respectively. In this processing output of

UML parser and EMF platform also plays key role. Once the processing is complete, two outputs are returned back to the user.

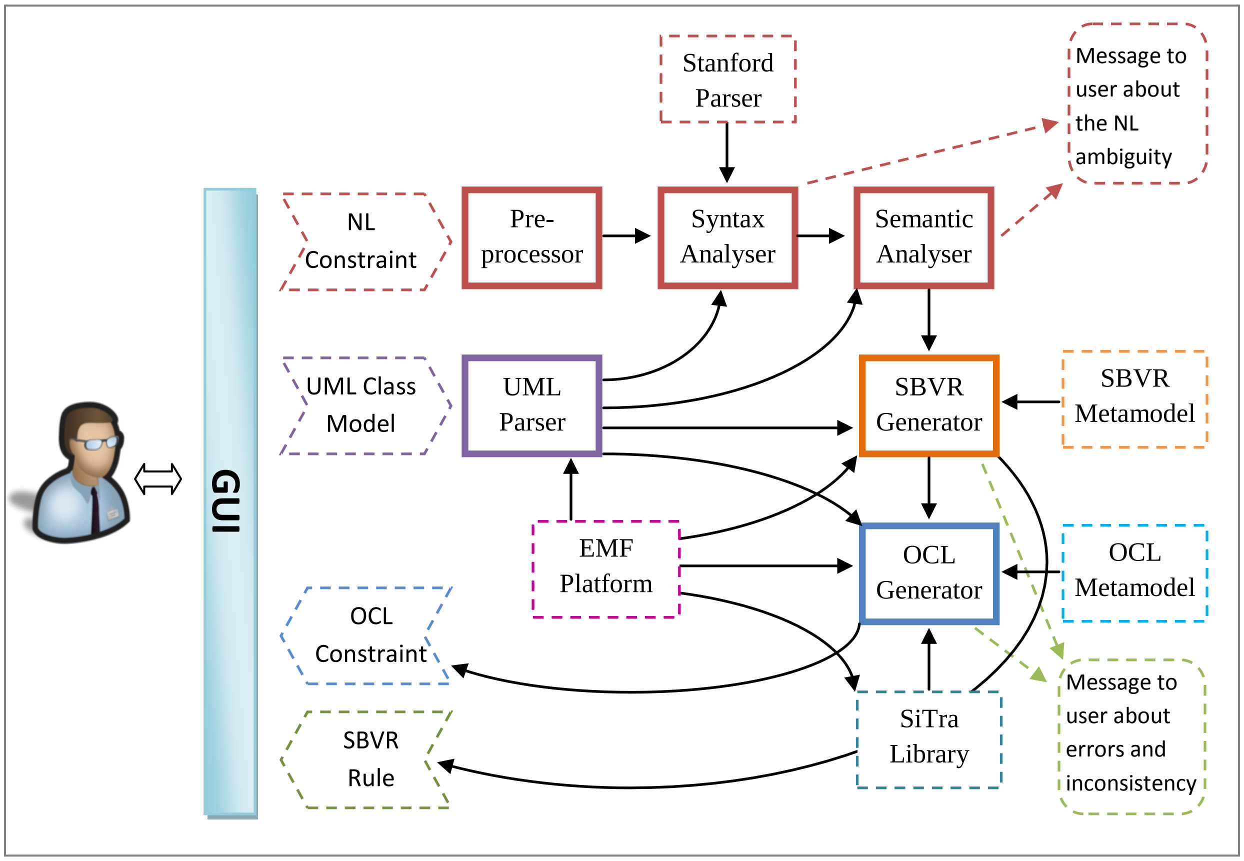


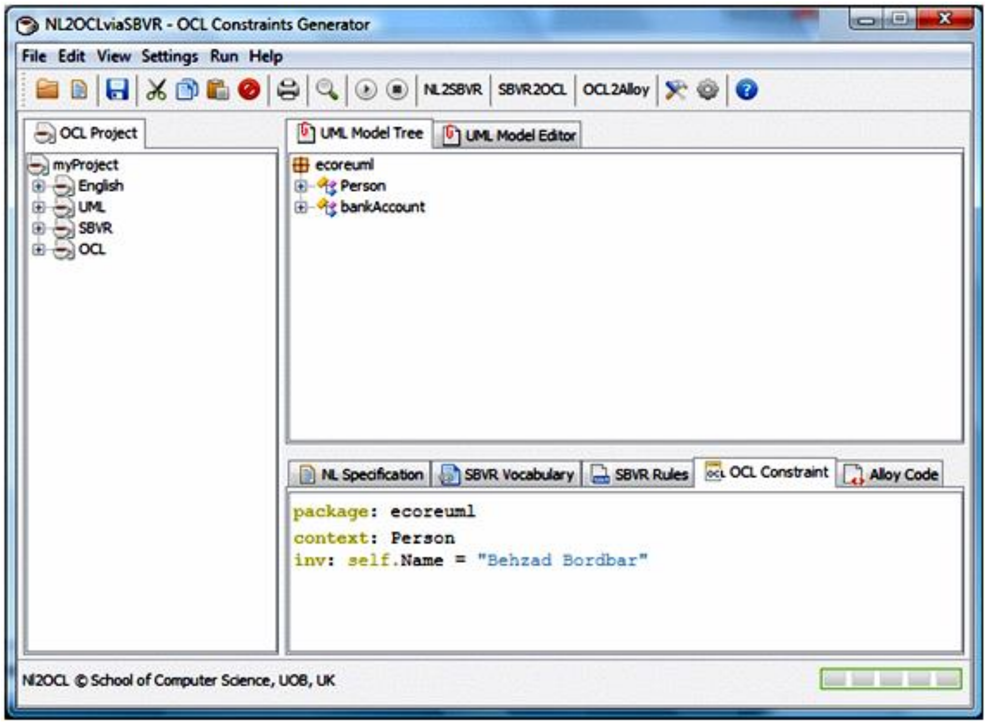
Figure 6.8: The NL2OCL*via*SBVR architecture

6.6User Interface Screenshots

The tool provides a graphical user interface, so that a user may easily provide inputs to the tool and get the output from tool. Tool's GUI is implemented using Java GUI libraries and is based on Windows look and feel by using call shown in Figure 6.9.

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFee
l");
```

Figure6.9: Windows look & feel for the tool



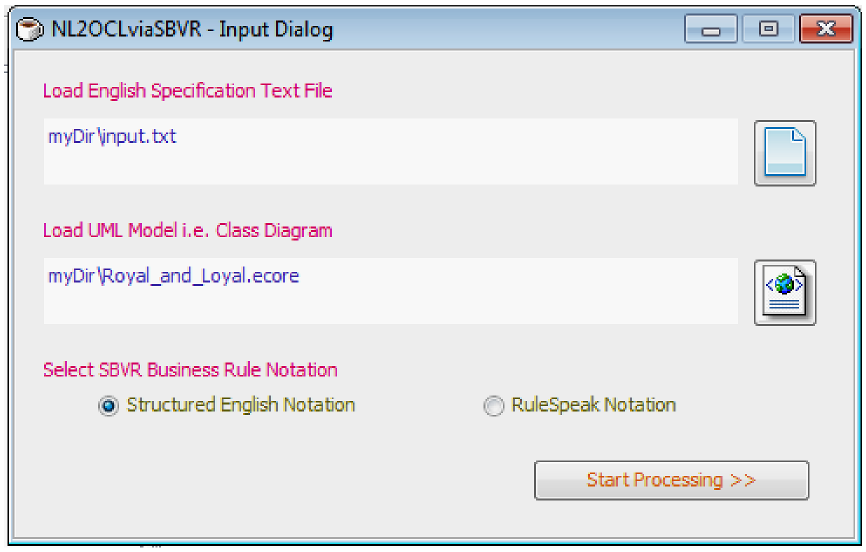
```
package: ecoreuml
context: Person
inv: self.Name = "Behzad Bordbar"
```

Figure 6.10: Screen shot of NL2OCL*via*SBVR

Figure 6.10 shows a screenshot of the NL2OCL*via*SBVR tool. The interface of the tool has three main sections. The section on the left shows the list of input files involved in the transformation. The section at the upper-right shows the metadata of the UML class model. The section at the lower-right shows multiple tabs. The third section consists of five tabs: NL specification tab, SBVR vocabulary tab, SBVR rule tab, OCL constraints tab, and Alloy code tab (see Figure 6.10). The NL specification tab is further divided into sub-tabs; those represent input text, output of lexical analysis, syntax analysis and semantic analysis. An extension of this work is generation of Alloy code from NL constraints [Bajwa, 2011e] via SBVR/OCL. This work shows that even there are limitations of the tool; still the tool has been used successfully for generating SBVR, OCL and Alloy for bench-mark case studies.

6.7 Tool in Use

Following are the steps performed to generate OCL from NL constraints. Figure 6.11 shows an input dialogue that gets two inputs: (1) Text file containing NL constraint (2) Ecore file containing UML class model. Figure 6.12 shows the UML Model Tree window that shows details of the input UML class model.


NL2OCLviaSBVR - Input Dialog

Load English Specification Text File

myDir\input.txt

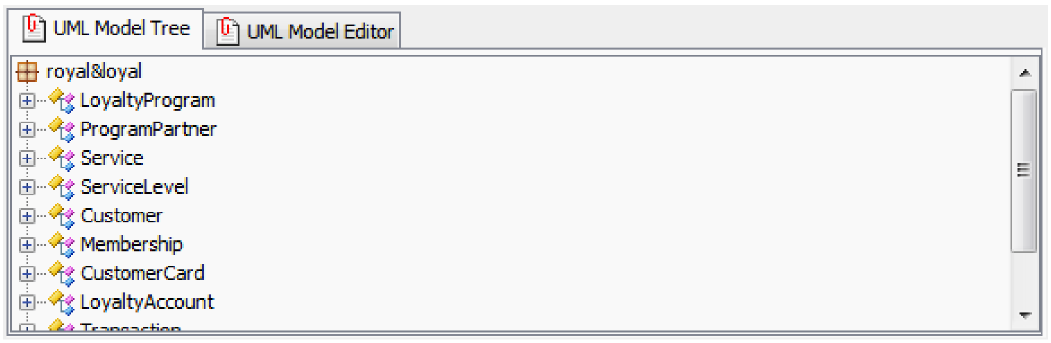
Load UML Model i.e. Class Diagram

myDir\Royal_and_Loyal.ecore

Select SBVR Business Rule Notation

[✓ Structured English Notation] RuleSpeak Notation

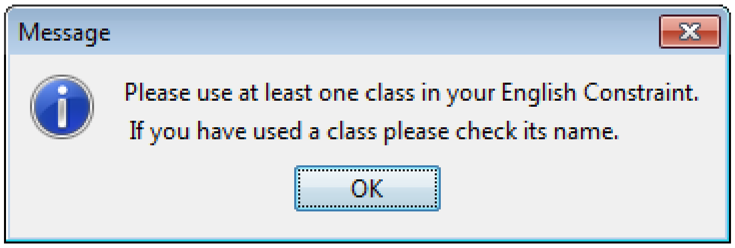
Figure 6.11: Input dialogue of the NL2OCL*via*SBVR



royal&loyal
- LoyaltyProgram
- ProgramPartner
- Service
- ServiceLevel
- Customer
- Membership
- CustomerCard
- LoyaltyAccount
- Transaction

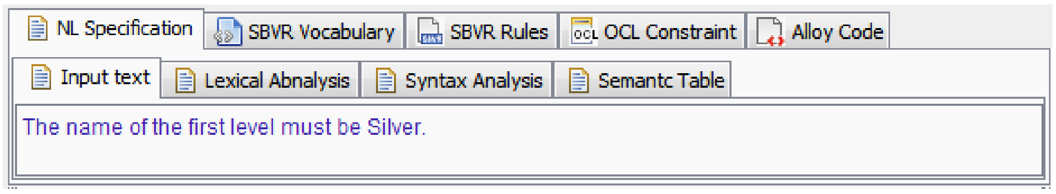
Figure 6.12: Input dialogue of the NL2OCL*via*SBVR

Figure 6.13shows an example of the message given for Constraint 7.2.14, where context is missing and user is indicated to provide at least one class in the UML constraint that can be used as a context.


Message

Please use at least one class in your English Constraint.
If you have used a class please check its name.

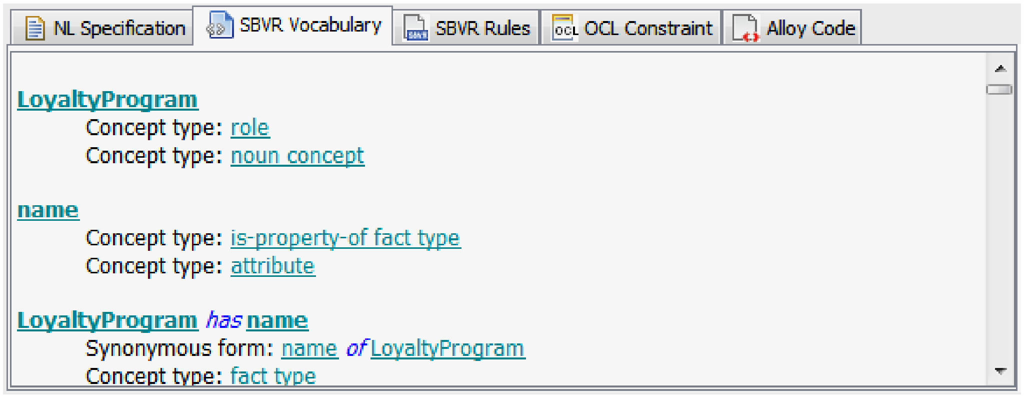
Figure 6.13:Error message shown to user in case of missing context



The name of the first level must be Silver.

Figure 6.14:Input Text dialogue showing NL constraint

Once the user corrects the Constraint 7.2.14 by introducing class `LoyaltyProgram` in the NL constraint as a context, the tool generates the output. Figure 6.15 shows the output of the UML module that generates the SBVR vocabulary from the input UML class model.



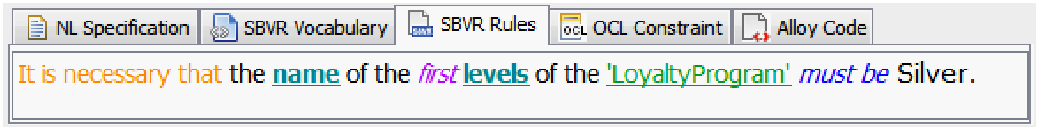
LoyaltyProgram
Concept type: role
Concept type: noun concept

name
Concept type: is-property-of fact type
Concept type: attribute

LoyaltyProgram *has* name
Synonymous form: name *of* LoyaltyProgram
Concept type: fact type

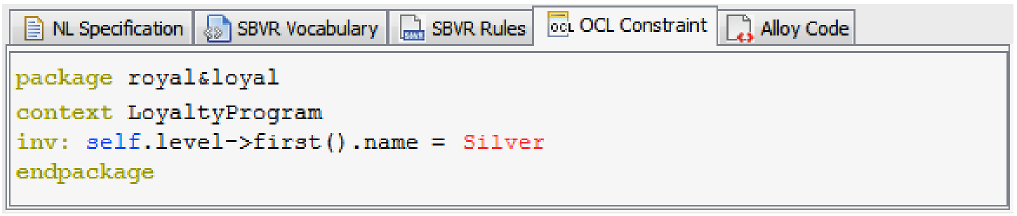
Figure 6.15:SBVR Vocabulary dialogue showing output SBVR vocabulary

Figure 6.16 shows the output of the SBVR module that generates the SBVR rule. Figure 6.17 shows the output of the OCL module that generates the OCL invariant.



It is necessary that the **name** of the *first* **levels** of the 'LoyaltyProgram' *must be* Silver.

Figure 6.16:SBVR Rules Dialogue showing output SBVR rule



```
package royal&loyal
context LoyaltyProgram
inv: self.level->first().name = Silver
endpackage
```

Figure 6.17:OCL Constraint showing output OCL invariant

6.8 Summary

In this chapter, the implementation framework of the NL2OCL*via*SBVR tool has been presented. The implementation uses a set of Java libraries and a few readymade components such as the Stanford parser, SiTra library, etc. However, to perform the rest of the functionalities no appropriate components were available. Hence, the researcher had to implement the rest of the components at his own. He has implemented all the components in Java. A few components are interconnected sequentially, while some others work in parallel with other related components. A windows look & feel based GUI was also presented. By using the tool the researcher has done three case studies, presented in the next chapter.

CHAPTER 7
EVALUATION

As the researcher has asserted in the previous chapters that the NL2OCL approach can generate OCL constraints from natural language (NL) specifications of constraints, in this chapter he presents three case studies to validate the NL2OCL approach and their results. He also presents evaluation criteria used to evaluate the performance of the presented approach. A similar criterion is used by Wahler [2008] in his PhD thesis to validate the performance of pattern based approach in automatic generation of OCL constraints. Since the researcher aims at comparing the NL approach with the pattern based approach for automated generation of OCL constraints, he has developed similar evaluation criteria with a few changes. He has not used 'analysis performance' and 'elicitation coverage' because in this thesis the analysis of models and requirements elicitation is not discussed. However, the researcher introduced a few additional criteria such as 'throughput measure', 'syntactic correctness', and 'transformation correctness' because in this thesis, aim of the research is to improve usability and ease adaption of OCL.

In the following section, the researcher presents parameters of quantitative and qualitative evaluation criteria used in this thesis to validate his claims.

7.1 Evaluation Criteria

The researcher has used the following criteria for evaluation of the NL2OCL approach. The evaluation criteria are divided into two categories: quantitative and qualitative.

7.1.1 Quantitative Criteria

There are two aspects of quantitative criteria used in this thesis, as below.

A. Specification Coverage

To know the effectiveness of the NL2OCL approach, the researcher needed to find out what proportion of NL and OCL specifications can be covered by the NL2OCL approach. There are two aspects of this criterion: coverage of OCL syntax and coverage of NL specification. Hence, it is measured what portion of OCL syntax can be covered by the NL2OCL approach. Similarly, the researcher also needed to find out that what percentage of a given set of NL constraints can be mapped to OCL constraints using the NL2OCL approach. This criterion will help the researcher to measure the completeness of the presented approach.

B. Throughput Measure

As the key focus of the presented research is to improve the usability of OCL, the researcher needed to measure up to what extent the NL2OCL approach has made it easier and time saving to generate OCL. To find out the role of the NL2OCL approach in the improvement of OCL usability, a throughput measure is calculated to measure the time and effort involved in generation of a set of OCL constraints from NL specification of constraints. Here, he has also compared this amount of time and effort that involved in manual generation of OCL.

7.1.2 Qualitative Criteria

There are three aspects of qualitative criteria used in this thesis, as below.

A. Syntactic Correctness

It is pertinent to find out if the OCL produced by the NL2OCL approach is syntactically correct. The measurement of syntactic correctness helps in establishing to what extent the NL2OCL approach can be involved in real time software modelling. The syntactic correctness is measured by compiling the OCL constraints with an OCL compiler, such as USE [Gogolla, et al., 2007].

B. Transformation Correctness

To find out if the OCL generated from the NL specification of constraint is equivalent to the original, this criterion will measured by generating component diagrams for both OCL constraint generated by NL2OCL*via*SBVR tool and OCL constraints generated by human expert. If the component diagrams generated by both OCL are same, then we can say that a NL specification is correctly transformed to an OCL specification.

C. Limitations

There are a few limitations of the NL2OCL approach. We need to discuss such limitations that help us to find out that what type of NL statements can be transformed to OCL. We also need to find out the way we can deal with such limitations in future.

7.1.3 Selection of Case Studies

By considering the nature of the approach and tool, the researcher looked for the case studies that had both NL and expert-written OCL. There were some case studies that had only NL constraints and no OCL constraints, e.g., legal-text case studies. On the other hand, some case studies had only OCL but no NL constraints, e.g., Mondex scenario [Kuhlmann, 2008]. It is not possible to come up with large number of case studies. However, we have done the available case studies and these case studies are bench mark in their respective domains. For example, the Royal and Loyal model is a part of book [Warmer J. & Kleppe A., 2003] and a PhD thesis [Wahler, 2008]. Similarly, the QUDV model has been worked by NASA and ESA and is also an ISO standard. The WBM case study is from IBM and its OCL is also written by IBM. In this PhD thesis, my method outperforms these case studies.

In the rest of the chapter, we present three Case studies with their results evaluated using the above criteria.

7.2 Case Study: Royal & Loyal

The first case study we have solved using the NL2OCL*via*SBVR tool is based on the "Royal & Loyal" model. The Royal & Loyal model was originally presented by Warmer and Kleppe [2003] in their book. The Royal & Loyal case study was used in various publications such as by Tedjasukmana [2006] to evaluate translation of OCL to SQL and by Wahler [2008] to evaluate the automated generation of OCL using the pattern based approach. The following section presents the overview of the case study and provides the details of constraints given in the Royal & Loyal model and their automated generation of OCL using the NL2OCL*via*SBVR tool.

7.2.1 The Royal & Loyal Model

The Royal & Loyal model is a good example of typical MDE approaches. The model is a computer system of a company that handles loyalty programs for its various customers. In this model, the central class isLoyaltyProgram(see Figure 7.1). Other classes such as class Customer and classProgramPartnerare connected through the central classLoyaltyProgram.

There is another class Membershipthat connects Customer with available Services in the loyalty program and also to each customer's respective account represented using classLoyaltyAccount. Each customer has aCustomerCardfor each membership in a loyalty program. Each customer can perform various types of Transactions using his card. In the Royal & Loyal model, the ProgramPartnersuse various services and each membership is associated with exactly one ServiceLevel.

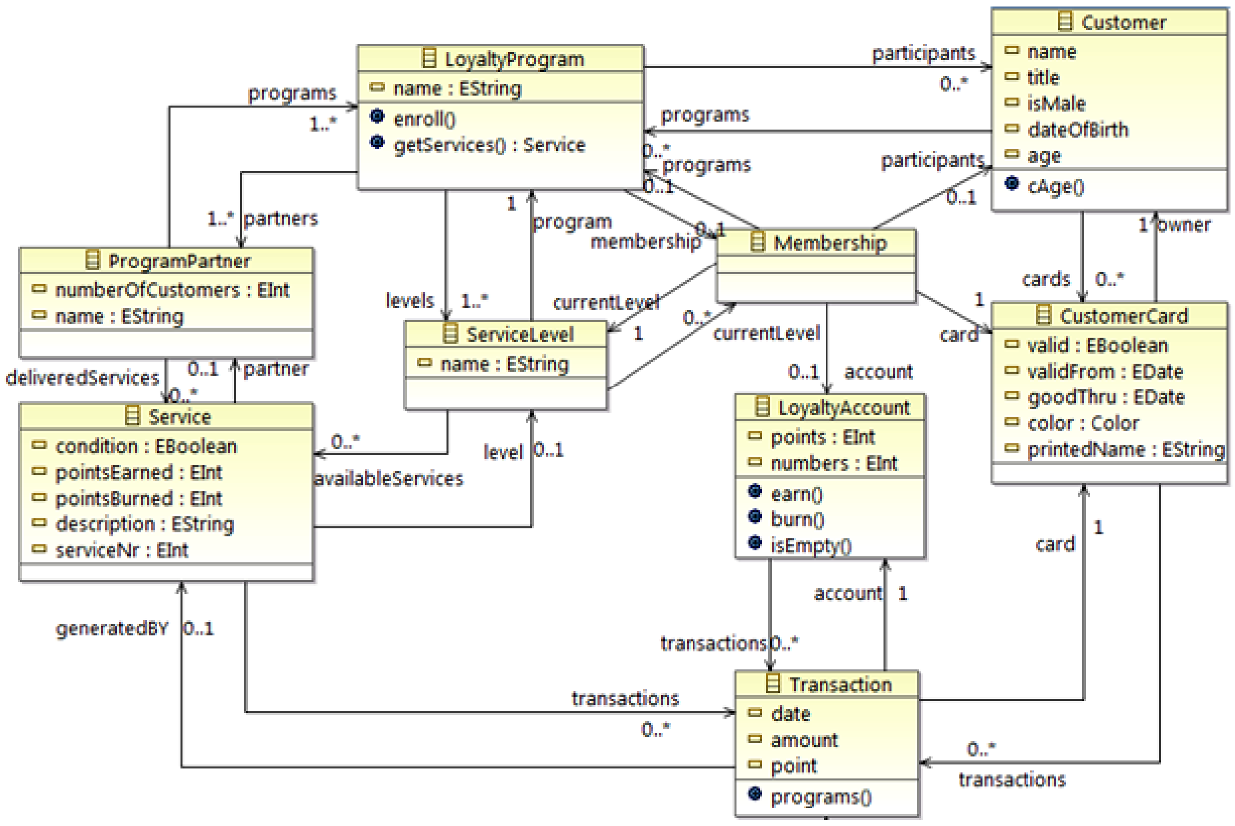


Figure 7.1: The Royal & Loyal model

There are two enumerations, `Date` and`Color`, in the model as well. However, the current implementation of NL2OCL approach does not support the enumerations and implementation of enumerations is a future work. Therefore, we have not represented enumerations in the used model shown in Figure 7.1. An overview of the components of the Royal & Loyal model is shown in Table 7.1.

Table 7.1: Overview of Royal & Loyal Model

	Type of Components		Number of Components
1	Classes	→	9
2	Attributes	→	24
3	Methods	→	7
4	Associations	→	25

7.2.2 Constraints for the Royal & Loyal Model

In the following text, we present the constraints given in [Kleppe and Warmer, 2003].We also present the SBVR and the OCL for each constraint generated by our tool the NL2OCL*via*SBVR. The OCL of each constraint given in [ibid] is also represented under title 'OtherOCL'.

Constraint 7.2.1

English: *Every customer who enters a loyalty program must be of legal age.*

SBVR: It is necessary that every **customer** who *enters* a **loyalty Program** *must be* of legal *age*.

OCL:
```
package royal_and_loyal
 context Customer
 invself.age >=
Endpackage
```

In the Constraint 7.2.1, SBVR was simple to generate as there are two Object Types '**customer**' and '**loyalty Program**'. However, there is another possible candidate of Object Type 'legal' but it is not available in the UML class model (see Figure 7.1).That is why it is not represented as Object Type. In generation of OCL, the term *legal* causes a similar problem as it is not part of the Royal and Loyal class model. The NL2OCL*via*SBVR tool is not able to recognize the items those are not part of the target UML class model. Therefore, the NL statement is transformed to "`self.age >= `", that is incomplete. Here, the user is given a message that *legal* is neither part of the input UML class model nor it is a valid integer value and the user should reconsider the NL constraint.

OCL:
```
package royal_and_loyal
context Customer
invself.age >= 18
   Endpackage
```

OtherOCL:
```
contextCustomer
invlegalAge: age >= 18
```

However, if the user changes NL constraint from "must be of legal age" to "must be of minimum age 18" or "minimum age must be 18", the NL constraint is mapped to OCL invariant "`self.age >= 18`".

Constraint 7.2.2

English: *Male customers must be approached using the title Mr..*

SBVR: It is necessary that *male* **customers** *must be approached using* the *title* 'Mr.'.

OCL:
```
package royal_and_loyal
context Customer
invself.isMaleimpliesself.title= Mr.
   Endpackage
```

OtherOCL:
```
contextCustomer
invmaleTitle: isMale implies title = 'Mr.'
```

In the Constraint 7.2.2, SBVR was easy to generate as there is one Object Type '**customer**' and there are two Characteristics '*male*' and '*title*'. However, the '*male*' Characteristic was difficult to identify as Customer class has 'isMale' attribute instead of 'male'. The NL2OCL approach handles such cases by checking the data-type of such Characteristics as if the data-type is Boolean, a prefix 'is' is concatenated with such Characteristics. For example, the attribute in class model is "isMale" while in NL constraint only "Male" has been mentioned. However, rest of the OCL mapping was straightforward.

Constraint 7.2.3

English: *The number of valid cards for every customer must be equal to the number of programs in which the customer participates.*

SBVR: It is obligatory that each the number of *valid* **cards** for each **customer** *must be* equal to the number of **programs** in which the **customer** *participates*.

OCL:
```
package royal_and_loyal
context Customer
inv self.cards->select(valid=true)->size()=self.programs->size()
    endpackage
```

OtherOCL:
```
context Customer
inv sizesAgree:programs->size()=cards->select(valid=true)->size()
```

In Constraint 7.2.3, it can be seen that a longer English sentence involving four Object Types '**cards**', '**customer**', '**programs**', and '**customer**' and one Characteristic '*valid*' is transformed to SBVR and OCL automatically. In this constraint, the term 'number of ' is mapped to OCL function `size()`.

Constraint 7.2.4

English: *The* validFrom *date of customer cards should be earlier than* goodThru.

SBVR: It is obligatory that the 'validfrom' date of **customercard** *should be* earlier than 'goodthru'.

OCL:
```
package royal_and_loyal
context CustomerCard
inv self.validFrom <self.goodThru
   Endpackage
```

Other OCL:
```
context CustomerCard
inv checkDates: validFrom.isBefore(goodThru)
```

In the Constraint 7.2.4, the term 'earlier' is used and this term is mapped to parameterized function '`isBefore`()' in OtherOCL. However, the current version of the NL2OCL*via*SBVR does not support the parameterized function calls, the 'earlier' keyword is mapped to simple relational operator '<' as an alternate. Here, the user is given a message about this mapping. The support of the parameterized function calls is a future piece of work.

Constraint 7.2.5

English: *The birth date of the owner of a customer card must not be in the future.*

SBVR: It is necessary that the 'dateofbirth' of the **owner** of a **customercard** *must* not *be* in the future.

OCL:
```
package royal_and_loyal
contextCustomerCard
   invself.owner.dateOfBirth <> future
endpackage
```

OtherOCL:
```
contextCustomerCard
invbirthDate: self.owner.dateOfBirth.isBefore(Date::now)
```

The Constraint 7.2.5 involves two Object Types '**customercard**','**owner**' and a Characteristic 'dateofbirth'. However, identification of 'dateofbirth' Characteristic was complex as in NL specification the used term is 'birth date' while `CustomerCard` class has an attribute `dateOfBirth`. To process such attributes we have provided support to map various combinations of date of birth to `dateOfBirth`. In OCL generation, similar to Constraint 7.2.4, the term 'not be in future' is used and this term is mapped to parameterized function '`isBefore`()' in OtherOCL. Here, we have again mapped the term 'not in future' to the relational operator '<>' due to the non-support of parameterized function calls in current implementation of the NL2OCL approach.

Constraint 7.2.6

English: *The owner of a customer card must participate in at least one loyalty program.*

SBVR: It is necessary that the *owner* of a **customercard** *must participate* in at least one **loyaltyprogram**.

OCL:
```
package royal_and_loyal
contextCustomerCard
invself.owner.programs -> Size()>= 1
endpackage
```

OtherOCL:
```
contextCustomerCard
invprogramParticipation: self.owner.programs ->size() > 0
```

In the Constraint 7.2.6, the transformation from NL to SBVR and OCL is fully automated. In this constraint the term 'at least' is mapped to Boolean operator '>=' to represent the meanings "greater than or equal to". Similarly, if the term 'at most' is used in NL specification that is mapped to Boolean operator '<=' to represent the meanings "less than or equal to".

Moreover, in the Constraint 7.2.6, to reach from `CustomerCard` to `LoyaltyProgram`, we need to navigate through two associations '`owner`' and '`programs`' to reach `LoyaltyProgram` (see Figure 7.2). Current implementation of the NL2OCL approach is able to handle such navigations that involve up to four classes and three associations. Actually, we have used Array-list to implement this module and for higher number of associations, strong data structure is required. This is sufficient to handle the Royal and Loyal case study. However, we aim to enhance the capability of the tool so that it may handle any number of associations in future.

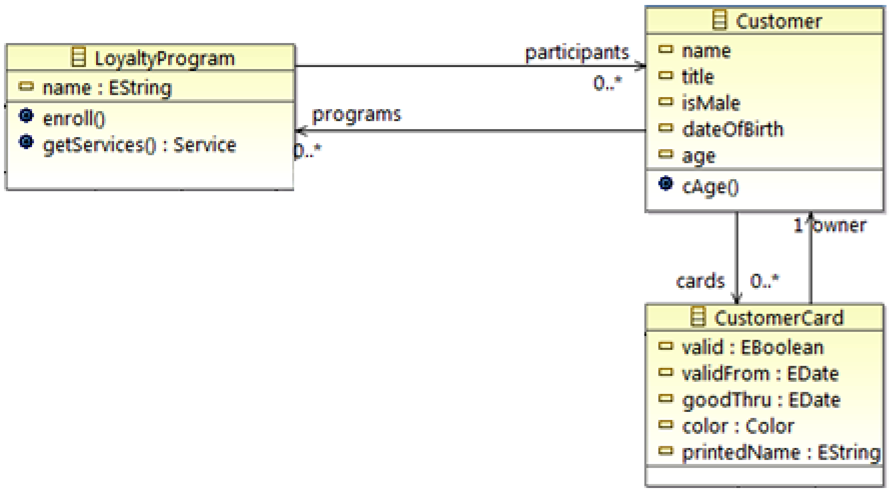


Figure 7.2: A subset of the Royal & Loyal model.

Constraint 7.2.7

English: *There must be at least one transaction for a customer card with at least 100 points.*

SBVR: It is necessary that there *must be* at least one **transaction** for a **customercard** with at least 100*points*.

OCL:
```
package royal_and_loyal
contextCustomerCard
invself.transaction->select(point >= 100)->Size()>= 1
endpackage
```

OtherOCL:
```
contextCustomerCard
    invtransactionPoints : self.transactions->
    select(points>100) ->notEmpty()
```

In the Constraint 7.2.7, the term 'at least' has been used twice. Moreover, in this constraint, two Object Types such as '**transaction**' and '**customercard**' and one Characteristic '*points*' are involved. Since, "at least one" is particularly mentioned in NL constraint, in OCL we map it to "`size() >= 1`". Rest of the transformation of Constraint 7.2.7 to SBVR and OCL is performed simply.

Constraint 7.2.8

English: *The service level of each membership must be a service level known to the loyalty program.*

SBVR: It is necessary that servicelevel of each membership *must be* a servicelevel *known* to loyaltyprogram.

OCL:
```
package royal_and_loyal
contextMembership
invself.currentLevel ->includes(self.program.levels)
  endpackage
```

OtherOCL:
```
contextMembership
invknownServiceLevel: programs.levels->includes(currentLevel)
```

The Constraint 7.2.8 involves three classes `Membership`, `LoyaltyProgram` and `ServiceLevel`. The transformation to SBVR is simple for this constraint but to generate OCL we need to navigate multiple associations such as to navigate from `Membership` to `ServiceLevel` via `LoyaltyProgram`.In this constraint, two associations are involved such as `programs` and `levels`(see Figure 7.3).

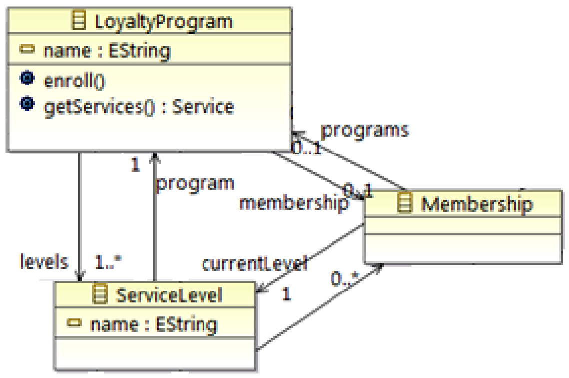


Figure 7.3: A subset of the Royal & Loyal model.

As the current version of the implementation can handle up to multiple associations, the NL specification was also transformed to OCL successfully. In this constraint, another OCL operation '`includes()`' is involved. Our tool uses maps the input NL to the '`includes()`' operations if there are is a cycle in the used associations.

Constraint 7.2.9

English: *The participants of a membership must have the correct card belonging to this membership.*

SBVR: It is obligatory that each **participants** of a **membership** *must have* the correct **card** *belonging* to this **membership**.

OCL:
```
package royal_and_loyal
contextMembership
invself.participants.cards ->includes(self.card)
    endpackage
```

OtherOCL:
```
contextMembership
    invcorrectCard: participants.cards->includes(self.card)
```

The Constraint 7.2.9 is similar to the Constraint 7.2.8 as this constraint also involves three classes `Membership`, `Customer` and `CustomerCard` and two associations; `cards` and `participants` as shown in Figure 7.4. The mapping to OCL was also forthright.

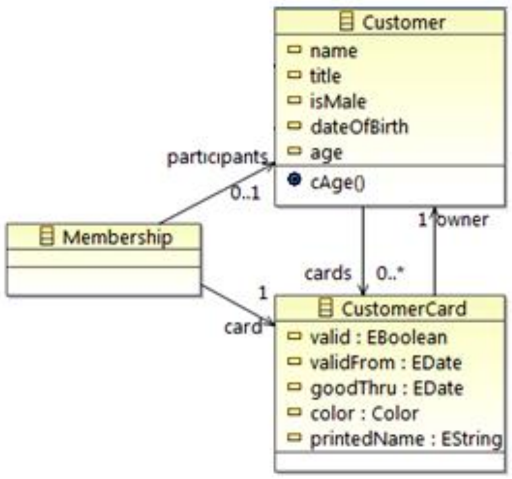


Figure 7.4: A subset of the Royal & Loyal model

Constraint 7.2.10

English: *The color of a membership's card must match the service level of the membership.*

SBVR: It is obligatory thatthe **color** of a **membership**'s**card***mustmatch* the **service level** of the **membership**.

OCL:
```
package royal_and_loyal
contextMembership
invself.card.color = self.currentLevel.name
    endpackage
```

OtherOCL:
```
contextMembership
        invlevelAndColor:
        currentLevel.name = 'Silver' implies card.color = Color:: silver
        and
        currentLevel.name = 'Gold' implies card.color = Color::gold
```

In the Constraint 7.2.10, the OtherOCL involves two enumeration values 'Silver' and 'Gold'. Since, these terms are not specifically mentioned in the NL constraint and moreover, the current implementation of the NL2OCL approach does not support enumeration values, the NL2OCL approach generates a simplified version of OCL. However, the generated OCL is syntactically different from the OtherOCL. Support for enumerations is a future work and to support enumerations, an extra module is required to be added that is sufficiently intelligent in choosing among the available enumeration values. After providing the support for enumeration values, such cases will be easy to translate.

Constraint 7.2.11

English: *Memberships must not have associated accounts.*

SBVR: It is obligatory that each **memberships** *must* not *have* associated **accounts**.

OCL:
```
package royal_and_loyal
contextMembership
invself.account -> isEmpty()
   endpackage
```

OtherOCL:
```
context Membership
        invnoAccount: account->isEmpty()
```

The Constraint 7.2.11 is simple to transform to SBVR and OCL as it involves only two Object Types: '**memberships**' and '**accounts**'. However, to generate OCL for this constraint, support for '`isEmpty()`' operation was required. We provide this support by mapping the NL term 'no' to OCL operation '`isEmpty()`' as it is used with an Object Type `account`(or a class), otherwise the term 'no' is mapped to the logical operator '`Not`', if it is used with a Characteristic (or a class attribute).

Constraint 7.2.12

English: *Loyalty programs must offer at least one service to their customers.*

SBVR: It is necessary that Loyaltyprogram *must offer* at least one service to customer.

OCL:
```
package royal_and_loyal
contextLoyaltyProgram
invself.partners.deliveredServices->size() >= 1
endpackage
```

OtherOCL:
```
contextMembership
   invminServices: partners.deliveredServices->size() >= 1
```

The OtherOCL for the Constraint7.2.12 involves two associations such as `deliveredSrvices` and `partners`, while `Membership` is used as a context. However the Constraint7.2.12 cannot be transformed to OCL due to a logical contradiction in the NL constraint. Here the logical contradiction is that the OtherOCL is semantically different from the NL constraint because 1) the relationship mentioned by the NL constraint is not available in the UML class model 2) class `customer` is not involved in the OtherOCL but it is part of the NL constraint, 3) for the given OtheOCL, `Membership` cannot be the context(see in Figure 7.5), due to the fact that `customer` is not directly involved in `deliveredServices`. However, the `partners` are involved in the `deliveredServices`. Hence the correct context should be `LoyaltyProgram`. Our tool automatically analyses the correct context.

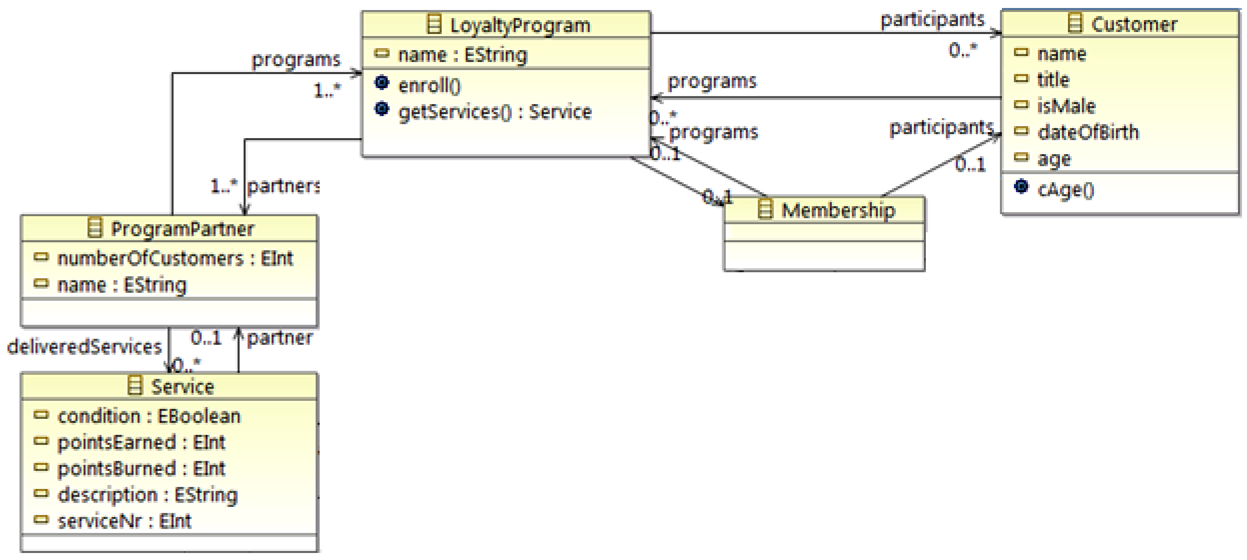


Figure 7.5: A subset of the Royal & Loyal model

Our tool is able to identify such logical contradictions due to the fact that each NL statement is mapped to a UML class model before it is transformed to OCL.

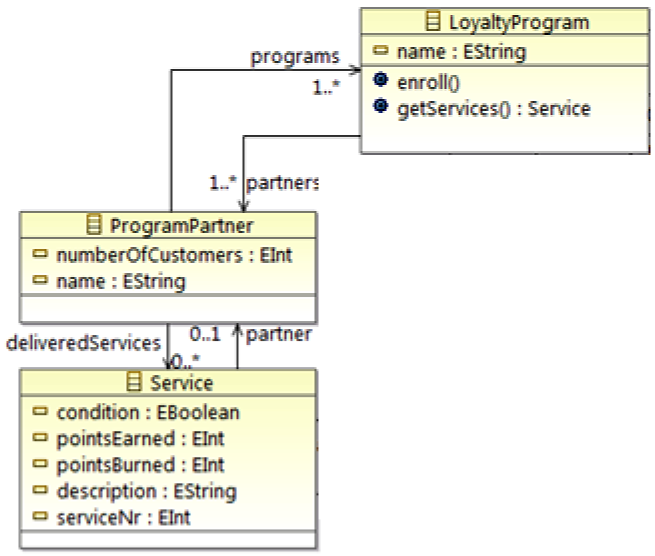


Figure 7.6: A subset of the Royal & Loyal model

To solve the Constraint7.2.12, the logical contradiction can be removed by either correcting the NL constraint or correcting the OCL constraint. However, to solve this constraint, we have changed the NL constraint by replacing `customers` with `partners`, to make NL statement is consistent with the class model. Here, `partners` become the logical replacement (see Figure 7.6). After removing the logical contradiction, the NL constraint was successfully transformed to OCL by our tool.

Constraint 7.2.13

English: *If none of the services offered in a loyalty program credits or debits the loyalty accounts, then these instances are useless and should not be present.*

SBVR: If none of the services *offered* in a loyalty program *credits* or *debits* the loyalty accounts, then it is permitted that these instances *are useless* and *should* not be *present*.

OCL:
```
package royal_and_loyal
contextLoyaltyProgram
inv
endpackage
```

OtherOCL:
```
contextLoyaltyProgram
invnoAccounts: partners.deliveredServices->forAll(
            pointsEarned = 0 and pointsBurned = 0 )
            implies Membership.account->isEmpty()
```

The Constraint7.2.13 is a larger NL constraint involving three Object Types and five Verb Concepts.

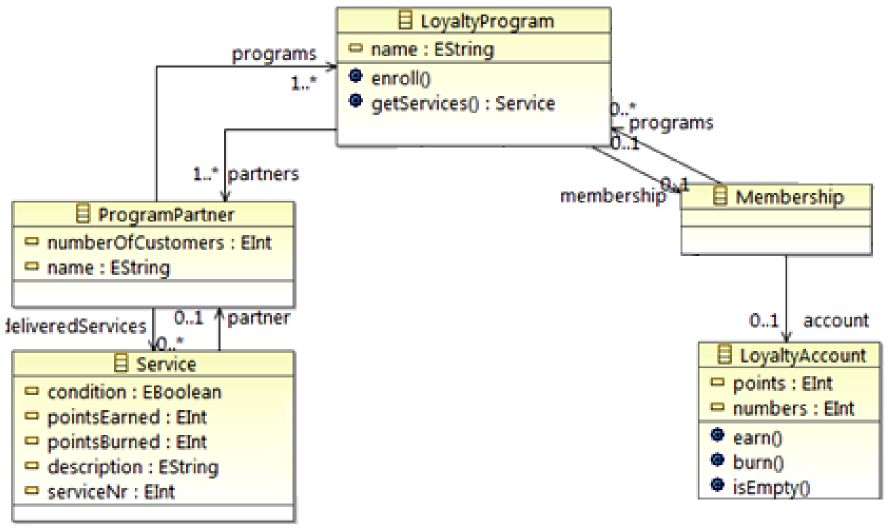


Figure 7.7: A subset of the Royal & Loyal model

The constraint was successfully mapped to SBVR. However, this constraint cannot be transformed to OCL due to the use of terms like 'credit' and 'debit' which are not part of the UML class model (see Figure 7.7). Moreover, there is a discrepancy between the English constraint and the OtherOCL that is in the NL constraint, the `LoyalAccount`class is associated with the `LoyaltyProgram`class, while in OCL the `LoyaltyProgram` class is mapped to the `Service` class by using its attributes, such as '`pointsEarned`' and '`pointsBurned`'. Here, the user is given a message that NL constraint is not consistent and the user should re-consider the NL constraint.

Constraint 7.2.14

English: *The name of the first level must be Silver.*

SBVR: It is obligatory that the *name* of the first **level** *must be* Silver.

OCL:
```
packageroyal&loyal
contextLoyaltyProgram
invself.levels->first().name = Silver
Endpackage
```

OtherOCL:
```
contextLoyaltyProgram
invfirstLevel : levels->first (). name = 'Silver'
```

The Constraint7.2.14 is transformed to SBVR but cannot be transformed to OCL, as it is a limitation of the approach that at least one class should be mentioned in the NL constraint that will become the context of the OCL. Here, the tool gives an error message to the user that there should be at least one class mentioned in the constraint. To generate the OCL similar to the OtherOCL, we need to introduce a class name such as the`LoyaltyProgram` class in the NL constraint. After the change the NL constraint will look like this: "*The name of the first level of the LoyaltyProgram must be* Silver". Afterwards, the transformation of Constraint 7.2.14 is simple.

Constraint 7.2.15

English: *There must exist at least one service level with the name basic.*

SBVR: It is necessary that there *must exist* at least one **ServiceLevel** with the 'name' basic.

OCL:
```
package royal_and_loyal
  contextServiceLevel
  invself.name = basic->exists( ()
  endpackage
```

Our tools correctly generate SBVR for the Constraint7.2.15 but the wrong OCL is generated. The wrong OCL is generated due to the fact the NL constraint for the Constraint7.2.15 is logically incomplete as "service level" mentioned in the NL constraint points to the "level" association and there is no class that will be the possible context of the OCL constraints. Here, the user is given a message that to generate a correct OCL, the user needs to introduce at least one class as a possible context. For the Constraint 7.2.15, the `LoyaltgProgram`class should be introduced in the NL constraints to provide a complete relationship. After the NL constraint is changed, it will look like this: "*There must exist at least one service level for a Loyalty Program with the name basic.*" After this change, the Constraint 7.2.15is correctly transformed to OCL.

OCL:
```
package royal_and_loyal
contextLoyaltyProgram
invself.level->exists(name = basic)
   endpackage
```

OtherOCL:
```
contextLoyaltyProgram
invbasicLevel: self.levels->exists(name = 'basic')
```

Constraint 7.2.16

English: *The number of participants in a loyalty program must be less than 10,000.*

SBVR: It is necessary that the number of *participants* in a **loyaltyprogram** *must be* at most 10,000.

OCL:
```
package royal_and_loyal
contextLoyaltyProgram
inv:self.participants->size()<10000
  endpackage
```

OtherOCL:
```
contextLoyaltyProgram
invmaxParticipants: self . participants ->size() < 10,000
```

The Constraint 7.2.16is very simple to process as it involves only one Object Type '**loyaltyprogram**' and one association '*participants*'. The transformation of Constraint 7.2.16to SBVR and OCL is very simple and straightforward.

Constraint 7.2.17

English: *The number of the loyalty account must be unique within a loyalty program.*

SBVR: It is necessary that the 'number' of the **loyaltyaccount** *must be* unique within a **loyaltyprogram**.

OCL:
```
package royal_and_loyal
contextLoyaltyProgram
invself.Membership.account->isUnique(acc|acc.numbers)
endpackage
```

OtherOCL:
```
contextLoyaltyProgram
invuniqueAccount: self.Membership.account->
isUnique(acc | acc.number)
```

In the Constraint 7.2.17, twoObject Types '**loyaltyprogram**'and '**loyaltyaccount**'is used with a Characteristic 'numbers'. However a new OCL operation '`isUnique()`' is involved in this constraint. The use of term 'unique' in NL statement hints the use of '`isUnique()`' operation. The transformation of Constraint 7.2.16to SBVR and OCL is successfully performed.

Constraint 7.2.18

English: *The names of all customers of a loyalty program must be different.*

SBVR: It is necessary that the 'name' of all **customers** of a **loyalty program** *mustbe* different.

OCL:
```
package royal_and_loyal
contextLoyaltyProgram
invself.participants.name->forAll(c1,c2|c1.name<>c2.name)
   endpackage
```

OtherOCL:
```
contextLoyaltyProgram
   invuniqueNames: self.participants->
                     forAll(c1,c2| c1<>c2 implies c1.name <> c2.name)
```

The Constraint 7.2.18involves two Object Types '**loyaltyprogram**' and '**customers**'. To transform this constraint to OCL, we need to cope with two new things: the term 'all' and the term 'different'. The functionality to map 'all' to OCL operation 'forAll' is already presented. However, the term 'different' is mapped to comparison of two terms with '<>' operator in OCL. The tool also provide functionality to compare two values, e.g., '`c1.name<>c2.name`'.

Constraint 7.2.19

English: *The maximum age of participants in loyalty programs is 70.*

SBVR: It is necessary that the at most 'age' of *participants* in **loyaltyprogram** *is* 70.

OCL:
```
package royal_and_loyal
contextLoyaltyProgram
inv:self.participants -> forAll( age<= 70)
   endpackage
```

OtherOCL:
```
contextLoyaltyProgram
   invmaxAge: participants->forAll(age()<= 70)
```

The Constraint 7.2.19, involves one Object Type '**loyaltyprogram**' and one Characteristic '*participants*'. The Constraint 7.2.16was also correctly transformed to OCL. However, the function `forAll` used in OtherOCL was not mentioned by our tool. If a user wants to introduce `forAll` operation in OCL, he/she will have to mention the term 'all' in the NL constraint.

Constraint 7.2.20

English: *There may be only one loyalty account that has a number lower than 10,000.*

SBVR: It is possibility thatthere*maybe* only one**loyaltyaccount** that*has* a'number' lower than 10,000.

OCL:
```
package royal_and_loyal
contextLoyaltyProgram
invself.Membership.account->one(number < 10,000)
   endpackage
```

OtherOCL:
```
contextLoyaltyProgram
invoneAccount: self.Membership.account->one(number < 10,000)
```

In the Constraint 7.2.20, only Object Type '**loyaltyprogram**' is used. Constraint 7.2.20was also correctly transformed to OCL. In this constraint, another OCL operation '`one()`' is also incorporated.

Constraint 7.2.21

English: *The attribute* numberOfCustomers *of class* ProgramPartner *must be equal to the number of customers who participate in one or more loyalty programs offered by this program partner.*

SBVR: It is obligatory that the attribute numberOfCustomers of class **ProgramPartner** *must be* equal to the number of **customers** who *participate* in at least one **loyaltyPrograms** *offered* by this **programPartner**.

OCL:
```
contextProgramPartner
invself.numberOfCustomers=programs.participants->asSet()->size()
   endpackage
```

OtherOCL:
```
contextProgramPartner
invnrOfParticipants: numberOfCustomers =
programs.participants->asSet()->size()
```

The Constraint 7.2.21is the lengthiest constraint of the Royal and Loyal model and it involves three classes '**ProgramPartner**', '**customers**', and '**loyaltyprograms**' and one attribute 'numberOfCustomers'. In this constraint, another OCL operation '`asSet()`' is also involved. Constraint 7.2.21 is correctly transformed to OCL.

Constraint 7.2.22

English: *A maximum of 10,000 points may be earned using services of one partner.*

SBVR: It is possibility that a maximum of 10,000 *points may be* earned using **services** of one **partner**.

OCL:

```
package royal_and_loyal
contextProgramPartner
invself.deliveredServices.pointsEarned<=10,000
endpackage
```

OtherOCL:

```
context ProgramPartner
inv totalPoints :
DeliveredServices.transactions->
select(oclIsTypeOf(Earning)).points->sum() < 10,000
```

In the Constraint 7.2.22, there are two Object Types such as '**Partner**' and '**Service**'. However the use of Object Type '**Partner**' is quite ambiguous as `partner` is an association in the Royal and Loyal model (see Figure 7.9) and if we handle `partner` as an association, no OCL can be generated because 1) partner association is directed from the `Service` class to the `ProgramPartner` class and it is not mentioned in the NL constraint 2) we can't reach from an association to a class's attribute `pointsEarned` because of opposite direction of the association.

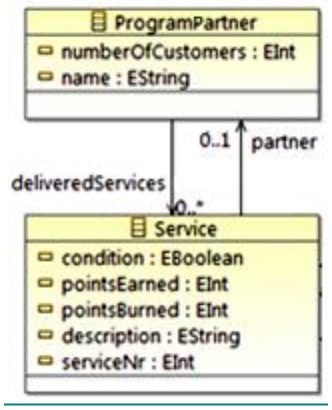


Figure 7.9: A subset of the Royal & Loyal model

Here, we need a class such as `ProgramPartner` to reach to reach `pointsEarned` of the class `Service`. Here, the user is given a message that '`partner`' is not a valid context and the user needs to introduce at least one class that is a valid context for the given NL constraint. To process this constraint, the user needs to replace the term '`partner`' with the term `ProgramPartner`. After this change, the NL constraint will look like "*A maximum of 10,000 points may be earned using services of one program partner.*" and it is simply transformed to OCL.

Constraint 7.2.23

English: *All cards that generate transactions on the loyalty account must have the same owner.*

SBVR: It is necessary that each all *card* that *generate* **transaction** on the **loyaltyaccount** *must have* the same *owner*.

OCL:
```
package: royal_and_loyal
contextLoyaltyAccount
invself.transactions.cards.owner->asSet()->size() = 1
endpackage
```

OtherOCL:
```
contextLoyaltyAccount
invself.transactions.card.owner->asSet()->size() = 1
```

The Constraint 7.2.23is another complex constraint of the Royal and Loyal model. It involves two classes '**transaction**' and '**loyaltyaccount**' two associations '*card*' and '*owner*'. Here, two other classes '**CustomerCard**' and '**Customer**' are indirectly involved as we need to access owner of the card. In this constraint, another OCL operation '`asSet()`' is also used. The Constraint 7.2.23is correctly transformed to OCL.

Constraint 7.2.24

English: *If the points earned in a loyalty account is greater than zero, there exists a transaction with more than zero points.*

SBVR: If the *points earned* in a **loyalty account** is at least zero, it is necessary that there *exists* a **transaction** with at least zero *points*.

OCL:
```
package royal_and_loyal
contextLoyaltyAccount
 invif (self.points > 0) then
            transaction -> exists( t| t.points>0)
endif
 endpackage
```

OtherOCL:
```
contextLoyaltyAccount
invpositivePoints : points > 0 implies transactions->
exists(t | t .points > 0)
```

In the Constraint 7.2.24an `if` statement is involved. This constraint was simple to generate as the current version of the tool has an ability to generate `if-else` expressions. To generate `if` expression in this constraints we need to extract two parts: (1) `if` part with condition (2)`then` part with the body. However, there is no `else` part as it is not mentioned in the NL constraint.

Constraint 7.2.25

English: *There must be one transaction with exactly 500 points.*

SBVR: It is necessary that there *must be* one **transaction** with exactly 500 'point'.

OCL:
```
package: royal_and_loyal
contextTransaction
invself.transaction->select(point = 500)->Size()=1
endpackage
```

OtherOCL:
```
contextLoyaltyAccount
inv500points: transaction.points->exists(p : Integer| p = 500)
```

The Constraint 7.2.25 involves one Object Type '**transaction**' and one Characteristic 'point'. In this constraint, the '`select()`' operation is also involved.

Constraint 7.2.26

English: *The available services for a service level must be offered by a partner of the loyalty program to which the service level belongs.*

SBVR: It is obligatory that the **available services** for a **service level***mustbe* offered by a **partner** of the **loyalty program** to which the **service level** belongs.

OCL:
```
package: royal_and_loyal
contextServiceLevel
invself.program.partners->
includesAll(self.availableServices.partner)
endpackage
```

OtherOCL:
```
contextServiceLevel
 invservicePartner: program.partners->includesAll
 (self.availableServices.partner)
```

The Constraint 7.2.26 is one of the complex constraints of the Royal and Loyal model as it involves two Object Type '**service level**' and '**loyalty program**' and two associations '**available services**' and '**partner**'. This constraint incorporates the '`includesAll()`' operation. However, the Constraint 7.2.26 is also successfully transformed to SBVR and OCL by our tool.

7.2.3 Quantitative Evaluation.

In this subsection, we perform a quantitative evaluation of the NL2OCL approach for the Royal & Loyal model. We use the criteria for quantitative evaluation as defined in Section 7.1.1: specification coverage and throughput measure.

A. Specification Coverage

The NL2OCL approach was designed to automatically extract various OCL syntactic elements so that they can be integrated to generate a complete OCL expression. The part of OCL syntax covered by the NL2OCL approach is shown in Table 7.2.

Table 7.2: OCL elements covered by the NL2OCL approach

OCL Elements	Supported by the NL2OCLviaSBVR
Context	Yes
Logical Expressions	Yes
Relational Expressions	Yes
Navigation	Yes
if-then-else	Yes
Collections	Selected ones are supported

Table 7.2 shows that most of the OCL syntax was covered in the implementation to translate English constraints of the Royal & Loyal model as it was required. Table 7.3 shows the details of the OCL elements implemented in the NL2OCL*via*SBVRtool in comparison with the OCL elements implemented in the Copacabana tool [Wahler, 2008].

Table 7.3: OCL Generation: NL2OCLviaSBVR vs. Copacabana

OCL Elements	NL2OCL*via*SBVR	Copacabana	Occurrences
Context	Yes	Yes	26
Navigation via association classes	Yes	No	19
Logical Expressions	Yes	Yes	1
Relational Expressions	Yes	Yes	20
Cardinality of Sets	Yes	No	2

Parameterized Function Calls	No	No	2
if-then-else	Yes	Yes	2
Enumerations	No	Yes	2
size()	Yes	Yes	9
isEmpty()	Yes	Yes	2
forAll()	Yes	Yes	3
exists()	Yes	Yes	2
includesAll()	Yes	No	1
select()	Yes	No	3
asSet()	Yes	Yes	2
isUnique()	Yes	No	1
oclIsTypeOf()	No	No	1

Our approach can generate 22 OCL constraints from total 26 English constraints of the Royal & Loyal model that is approximately 85% (see Table 7.4). The remaining 4 constraints cannot be translated due to the violation of one of the limitations of the NL approach discussed in section 7.4. However, if we slightly tune the inputs with respect to the given limitations of the NL2OCL*via*SBVR, further 3 constraints can be translated. This improves the ratio of results to 96.15%. Only the Constraint 7.2.13 cannot be translated to OCL due to logical contradiction in the statement and this constraint needs complete revision.

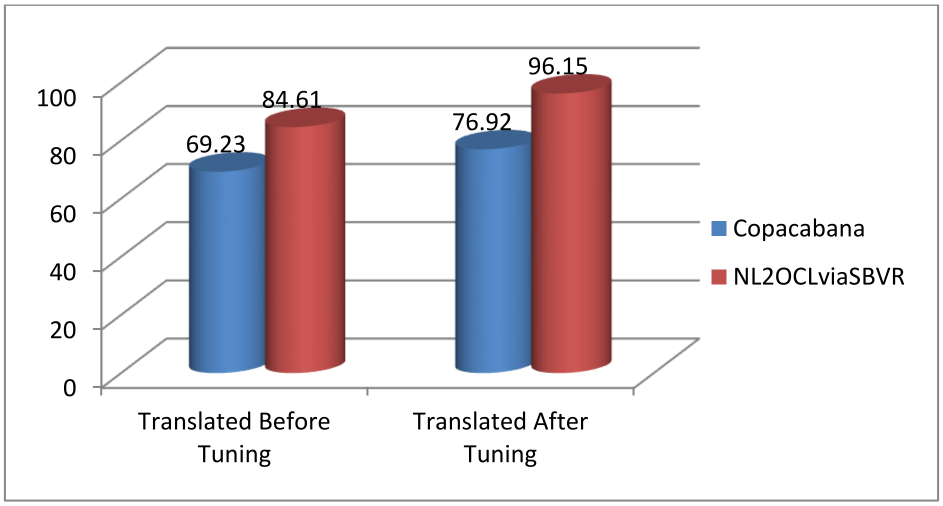


Figure 7.10: Specification Coverage: NL Approach vs. Pattern Approach

In comparison to NL based approach, the pattern based approach can generate 18 constraints before tuning and 20 constraints after tuning. A comparison of both approaches is shown in Figure 7.10. It is important to indicate here that the researcher is conscious of the fact that the NL2OCL approach used to generate OCL from NL constraints cannot be 100% correct. Furthermore, the researcher has used NL and automated generated SBVR in pair to resolve NL ambiguities and to clarify vagueness by pointing them out, this will not be a 100% solution either and the researcher is aware of it.

Table 7.4: Specification Coverage: NL Approach vs. Pattern Approach

OCL Elements	**Total Constraints**	**Translated Before Tuning**	**Translated After Tuning**
Copacabana	26	18	20
NL2OCL*via*SBVR	26	22	25

Table 7.4 shows that the NL2OCLviaSBVR tool can translate 22 constraints without tuning. Here tuning means improving the NL input statement. The ratio of translation before tuning is 84.61% that outperforms the other approaches. Since the effectiveness of un-tuned tool is good enough to be useful but the researcher has also demonstrated that further tuning can produce better accuracy that can be up to 96.15% as after tuning the tool can translate 25 constraints out of 26 constraints.

B. Throughput Measure

Throughput was measured to validate the effectiveness of the presented approach in real-time scenario, where people with various levels of knowledge and expertise need to write OCL. A small survey was conducted to measure the throughput of the NL2OCL approach.For the survey three groups were chosen as below:

- Novel : A user who is quite new to OCL
- Medium :A user who knows basics of OCL
- Expert : A user who is expert of OCL

Table 7.5: Usability Survey Results

User	Easy to Use		Time-Saving	
	Manual	By Tool	Manual	By Tool
Novel	30%	90%	25%	85%
Medium	55%	85%	40%	80%
Expert	70%	85%	60%	70%
Average	**51.66%**	**86.66%**	**41.66%**	**78.33%**

Each group was containing 10 users. Each user was given a set of 10 English constraints and then they were asked to manually write OCL for each constraint. Users of all groups were given five minutes for writing each constraint. Afterwards, all the users were asked to generate OCL for the same constraints using our tool the NL2OCL*via*SBVR. Once all the users finished their work they were given a questionnaire to fill. In the questionnaire, questions were asked regarding various aspects: simple to use, time-saving, etc. Each user was asked to give 1 to 10 score for each category. The average values calculated for different parameters are clearly showing in Table 7.12that the used approach was making an impact. The statistic in Figure 7.11 and Figure 7.12 manifest that users specifically in novel and medium category find it easy to generate OCL constraints using our tool. Figure 7.13 presents overall statistics of user's feedback.

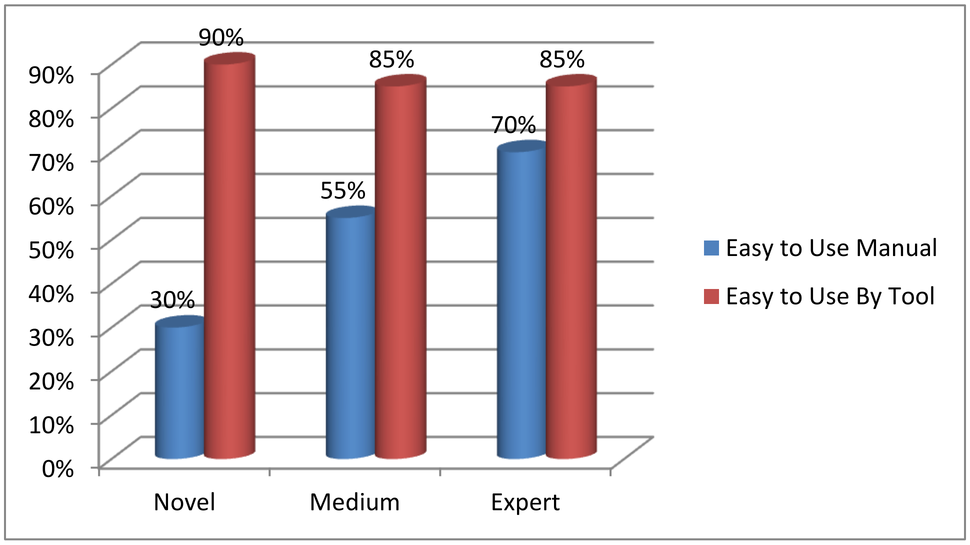


Figure 7.11: Ease to generate constraints: Manual vs. By Tool

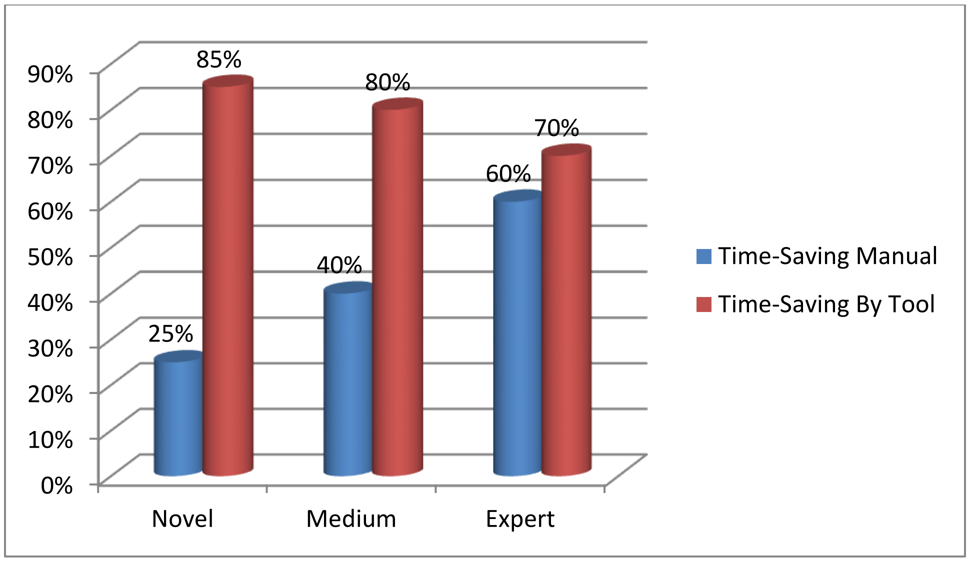


Figure 7.12: Time saving in constraints generation: Manual vs. By Tool

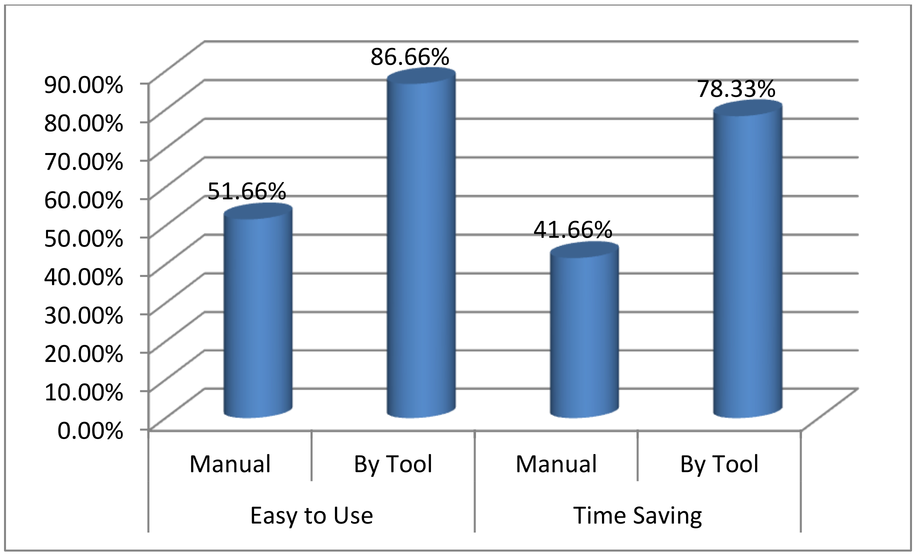


Figure 7.13: Throughput Measure: Manual vs. By Tool

7.3Case Study: QUDV

SysML is an OMG standard that is typically used for system engineering and modeling of measurement systems [OMG, 2010]. SysML is simple and smaller than UML in terms of diagram types and total constructs.

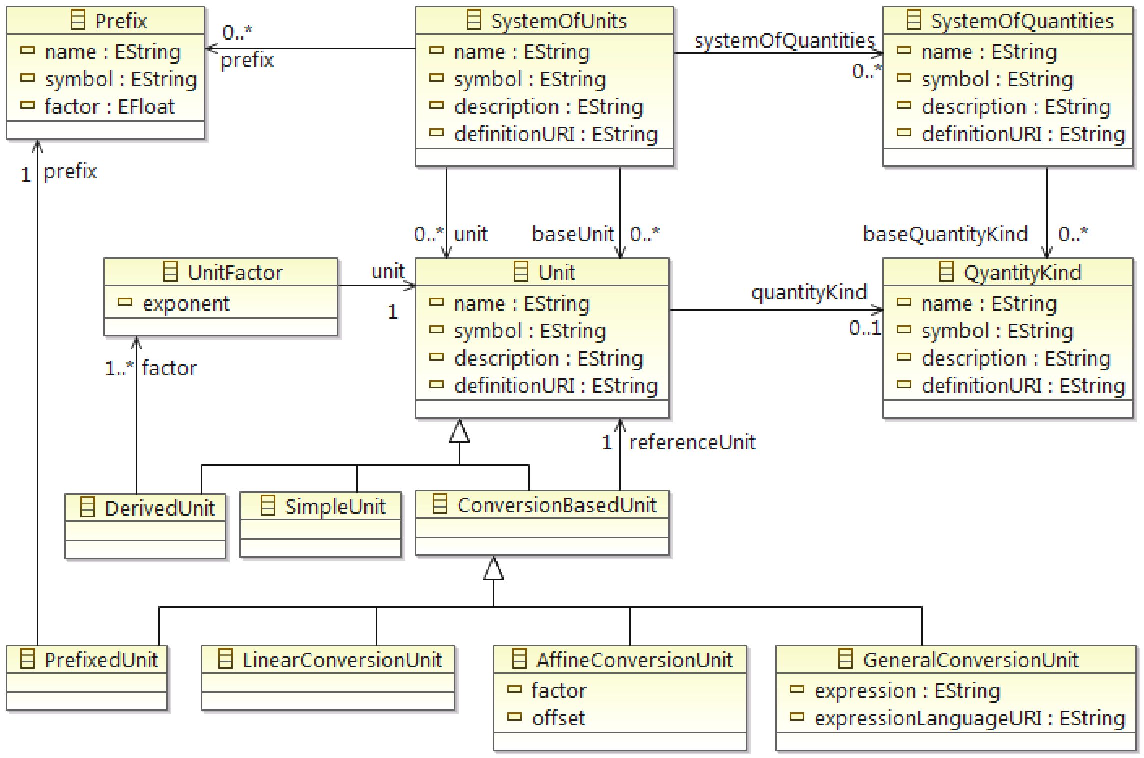


Figure 7.14: QUDV Unit diagram

7.3.1 QUDV Library Model

The presented case study is based on the well-known problem of checking the coherence of a system of units & quantities. The precise specification of this problem is given in ISO 80000. However, Koning extracted the relevant parts of this problem is given in SysML 1.2, Annex C.5, "Model Library for Quantities, Units, Dimensions, and Values (QUDV)" [Koning, 2005]. Both NASA and ESA have also worked at the problem of modeling the measurement units as a part of the system. Figure 7.14 shows the unit diagram of the QUDV model.

There are three parts of QUDV model. However, we have used only two of them: the QUDV Unit Model (see Figure 7.14) and the QUDV Concept Model (see Figure 7.15). There are four main classes in the model: `SystemOfUnits`, `SystemOfQuantities`, `Units` and `QuantityKind`.

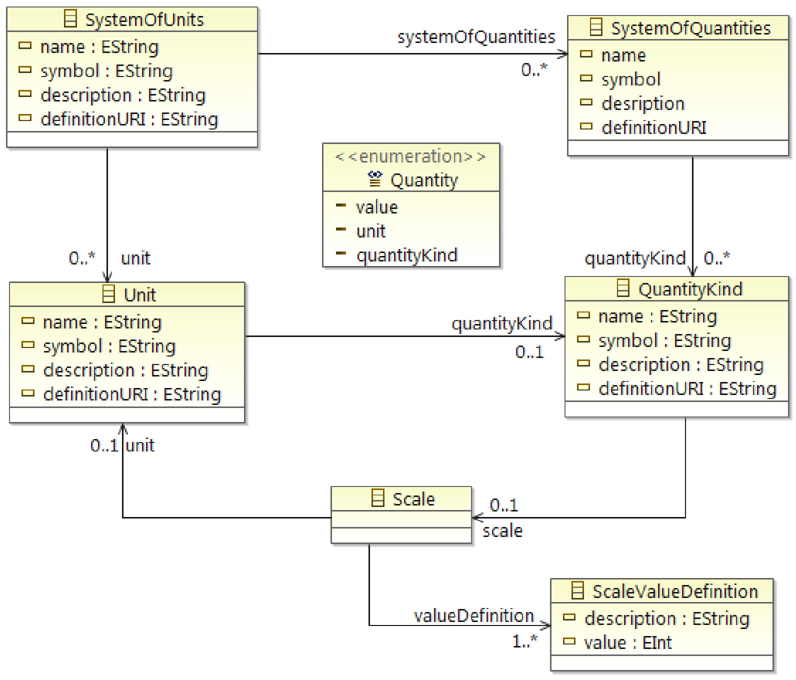


Figure 7.15: QUDV Concepts Diagram

From the QUDV concept diagram (see Figure 7.15), the English specification of a constraint on `Quantity` class is given in [ibid], Section C.5.2.11. An overview of the components of the QUDV unit and concept model is shown in Table 7.6.

Table 7.6: Overview of QUDV Model

	Type of Components		QUDV Unit Diagram	QUDV Concepts Diagram	Total
1	Classes	→	13	7	20
2	Attributes	→	22	21	43
3	Methods	→	0	0	0
4	Associations	→	17	7	24

7.3.2 QUDV Constraints

In this subsection, we present the five constraints of QUDV model library given in [OMG, 2010], Section C.5.2.10. We have generated SBVR and OCL of the QUDV English constraints using our tool NL2OCL*via*SBVR. English specification of the constraints on `PrefixedUnit` class is given in [OMG, 2010], Section C.5.2.10.

Constraint 7.3.1

This constraint specifies that it is not allowed to prefix an already prefixed measurement unit. In general the `referenceUnit` should be a `SimpleUnit`. The output of the input English specification is below:

English: *The referenceUnit shall not be a PrefixedUnit.*

SBVR: It is obligatory that each referenceUnit*shall* not*be* aPrefixedUnit.

OCL:
```
package QUDV
contextPrefixedUnit
invnot referenceUnit->oclIsTypeOf(PrefixedUnit)
endpackage
```

OtherOCL:
```
package QUDV
contextPrefixedUnit
invnot referenceUnit->oclIsTypeOf(PrefixedUnit)
endpackage
```

Two more constraint are defined on `SystemOfUnits` class and their English specification is given in [ibid], Section C.5.2.21, where `SystemOfUnits`is defined as “set of base units and derived units”.

Constraint 7.3.2

This constraint specifies that only one base unit should be specified for a base quantity. A “base unit” is a preferred unit in which base quantities of the associated `systemOfQuantities` are expressed. A base unit is a measurement unit that is adopted by convention for a base quantity. The output of the Constraint 7.3.2 is below:

English: *In a coherent system of units, there is only one base unit for each base quantity.*

SBVR: In a coherent system of units, it is obligatory that there is exactly one base unit for each base Quantity.

OCL:
```
package QUDV
contextSystemOfUnits
invself.systemOfQuantities.baseQuantityKind.size()=
self.baseUnit->size() =  1
endpackage
```

OtherOCL:
```
package QUDV
        contextSystemOfUnit
        def: isCoherent() : QUDV::Blocks::Boolean = baseUnit->size() =
        systemOfQuantities.baseQuantityKind->size() and baseUnit->
        forAll(bU| systemOfQuantities.baseQuantityKind->one(bQK|bU.
        quantityKind=bQK)) and systemOfQuantities.baseQuantityKind->
        forAll(bQK| baseUnit->one(bU|bQK=bU.quantityKind))
        endpackage
```

The OCL generated for Constraint 7.3.3 is syntactically different. However, semantically it is equivalent to the OtherOCL.

Constraint 7.3.3

The other constraint defined on the `SystemOfUnits` class is a definition of a coherent derived unit. A derived unit is a factor of a unit and the exponent of a factor is also 1. Our tool generates incomplete OCL as this constraint consists of out of domain information such as "*with no other proportionality factor than one*" and we have explained in section 7.4.3 that our current implementation cannot handle the extra domain information.

English: *A coherent derived unit is a derived unit that, for a given system of quantities and for a chosen set of base units, is a product of powers of base units with no other proportionality factor than one.*

SBVR: A coherent derived unit *is* a derived unit, it is obligatory that for a given system of quantities and for a chosen set of base units, *is* a product of powers of base units with no other proportionality factor than one.

OCL:
```
package QUDV
contextSystemOfUnits
invself.baseUnit->includesAll() ->
endpackage
```

OtherOCL:
```
package QUDV
        contextSystemOfUnit
        def: isCoherent(du : DerivedUnit) : QUDV::Blocks::Boolean=
        baseUnit->includesAll(du.factor.unit) and du.factor.exponent->
        forAll(numerator=1 and denominator=1)
        Endpackage
```

Constraint 7.3.4

The constraint specifies that from three properties (`value`, `unit`, and `quantityKind`) of Quantity, one should be specified. The output of the input English specification is below:

English: *At least one of the three properties of Quantity shall be specified.*

SBVR: It is obligatory that at least one of the three properties of Quantity*shall be specified.*

OCL:
```
package QUDV
contextQuantity
 inv self.value->isNotEmpty() orself.unit->isNotEmpty() or
 self.quantityKind->isNotEmpty()
 endpackage
```

OtherOCL:
```
package QUDV
contextSystemOfUnit
inv: quantityKind->isNotEmpty()
or unit->isNotEmpty()
or value->isNotEmpty()
endpackage
```

Constraint 7.3.5

The NL specification of the Constraint 7.3.5 is related to the `Scale` class and is given in [OMG, 2010] Section C.5.2.15. The constraint specifies that it is not allowed to prefix an already prefixed measurement unit. In general the `referenceUnit` should be a `SimpleUnit`.

English: *If a unit is specified on Scale, then it shall be the same as the unit of the associating QuantityKind.*

SBVR: If a unit *is specified* on Scale, then it is obligatory that unit *shall be* the same as the unit of the associating QuantityKind.

OCL:
```
package QUDV
contextScale
 invif (notself.unit->isEmpty()) then
self.unit.quantityKind = self.quantityKind
endif
 endpackage
```

OtherOCL:
```
package QUDV
contextScale
inv: unit->isEmpty() or unit.quantityKind = self.quantityKind
endpackage
```

In Constraint 7.3.5,`if` keyword is mentioned in the NL constraint that leads to an `if`expression in the OCL constraint. As the NL constraint says that a `Scale` unit should be equivalent to the `quantityKind` unit, the generated OCL is syntactically different from the OtheOCL however both are semantically equivalent.

7.3.3 Quantitative Evaluation

In this subsection, we perform a quantitative evaluation of the NL2OCL approach for the QUDV library model. For the QUDV Model case study, only specification coverage is discussed here. The combined results of throughput measure for all three case studies have been represented in Table 7.5.

A. Specification Coverage

The NL approach was designed to automatically extract various OCL syntactic elements so that they can be mapped to a complete OCL expression. To translate English constraints of the QUDV model, most of the OCL syntax was covered in the implementation. Table 7.7 shows the details of the OCL elements implemented in our NL2OCL approach.

Table 7.7: OCL Generation: NL Approach vs. Pattern Approach

OCL Elements	**NL2OCL*via*SBVR**	**Occurrences**
Context	Yes	05
Navigation via association classes	Yes	19
Logical Expressions	Yes	4
Relational Expressions	Yes	3
Cardinality of Sets	Yes	1
Parameterized Function Calls	No	0
if-then-else	Yes	1
Enumerations	No	0
size()	Yes	2
isEmpty()	Yes	1
isNotEmpty()	Yes	3
forAll()	Yes	0

exists()	Yes	0
includesAll()	Yes	1
select()	Yes	0
asSet()	Yes	0
isUnique()	Yes	0
oclIsTypeOf()	No	1

Our approach can generate 4 OCL constraints from the 5 English constraints of the QUDV model that is 80%. The remaining constraint (Constraint 7.3.3) cannot be translated due to the violation of one of the limitations of the NL approach discussed in Section 7.4.

7.4 Case Study: WebSphere Business Modeler

We have selected the third case study from the domain of business processes: *WebSphereBusinessModeler* (WBM) [IBM, 2007]. WBM is internally used by IBM and we aim to translate NL constraints of WBM using our tool. The NL constraints and their OCL is generated by a research team in the IBM Zurich Research Laboratory for the purpose of process merging. The basic purpose of defining these constrains was to restrict the input models those were initially described in NL and Java.

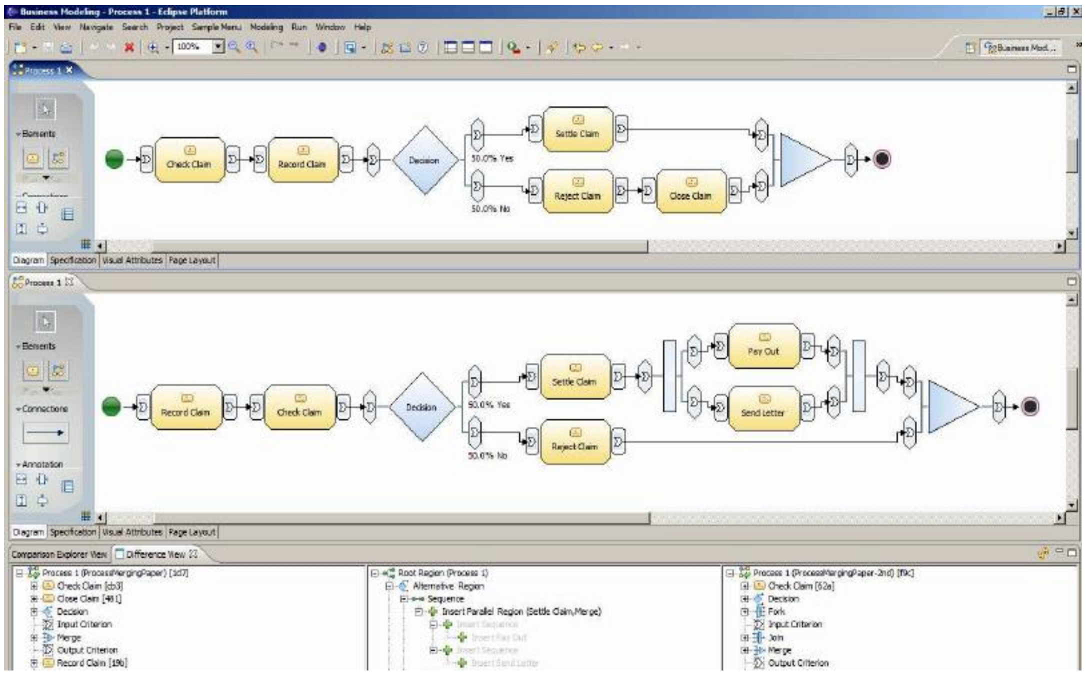


Figure 7.16: Screenshot of the process-merging prototype in WBM [Wahler, 2008].

7.4.1 The WBM Process Model

This section presents the WBM process model typically employed for specifying the business processes. The WBM process model is based on a set of UML class diagrams. There are four parts of the model. However, we have used three of them (see Figure 7.16, Figure 7.17, and Figure 7.18) with regards to the scope of constraints. Here, Figure 7.17and Figure 7.18explain the class diagrams of the metamodel of the process model. Figure 7.19provides the descriptionof various types of activity nodes in the process model.

It is shown in Figure 7.17 that activity nodes can be of two types such as `ExecutableNode` and `ControlNode`. Moreover, an executable node can be of two further types: an `Action`or a `StructuredActivityNode`.Similarly, a`ControlNode` can be either an `InitialNode` or a `FinalNode`. The`FinalNode`is further classified into two specialized types `FlowFinalNode`that ends the execution of a single branch and a`TerminationNode`that ends the whole business process.

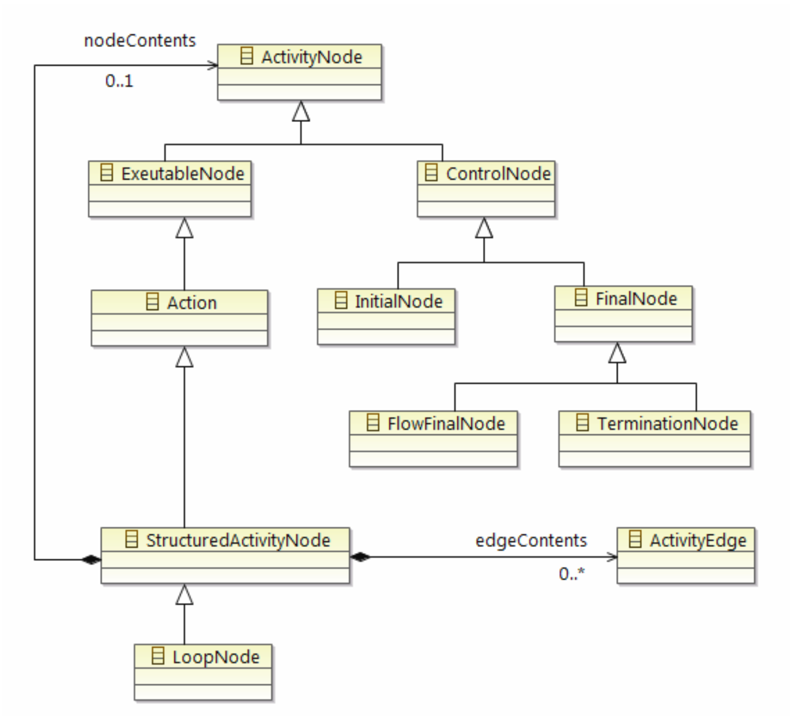


Figure 7.17: Process model: activity nodes (A)

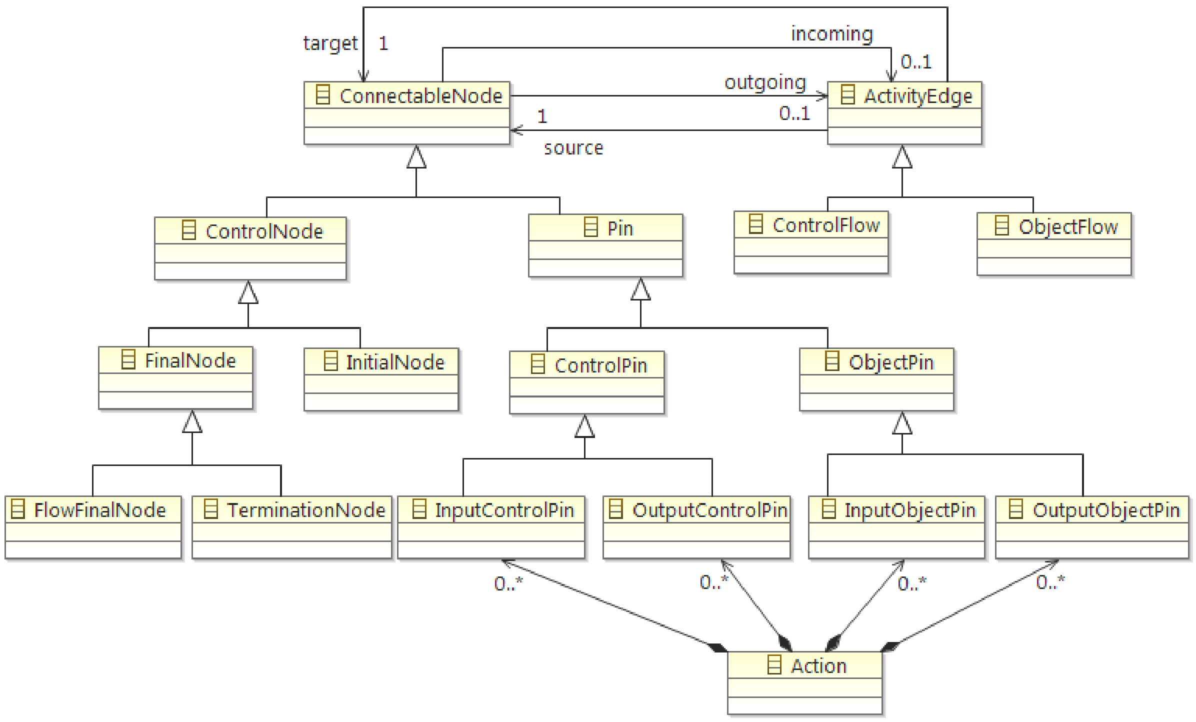


Figure 7.18: Process model: connectable nodes (B)

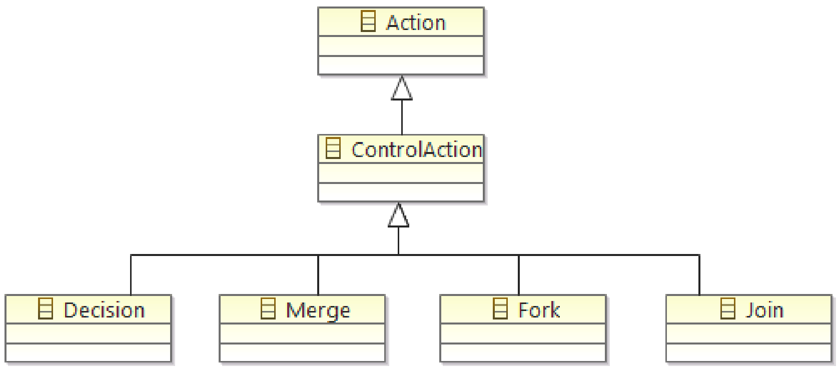


Figure 7.19: Process model: control actions (C)

Figure 7.19 shows that a `ControlAction`, is a special type of `action` that can be classified into object flow and control flow. However, a control action can be classified into four subclasses: `Decision`, `Merge`, `Fork`, and `Join`. Here, the `Decision` and `Merge` control actions are employed for modeling the optional branches in a process, while the `Join` and `Merge` control actions are involved in modeling of the parallel or analogous branches in a process model. An overview of the components of QUDV unit and concept model is shown in Table 7.8.

Table 7.8: Overview of WBM Process Model

	Type of Components		A	B	C	Total
1	Classes	→	11	17	6	34
2	Associations	→	11	22	5	38

7.4.2 Constraints for the WBM Process Model

As we have described in Section 7.3.1 that a set of constrains were also provide by IBM to restrict the process models. In this section, we specify SBVR and OCL generated by using our tool the NL2OCL*via*SBVR. Similar to previous two case studies, we also provide OtherOCL written for the NL constraints by IBM research team.

Constraint 7.4.1

English: *Only connected models are supported, i.e., every element is reachable from the start node* and *an end node is reachable from every element.*

SBVR: It is obligatory only connected models *are supported*, i.e., it is obligatory every element *is reachable* from the start node and an end node *is reachable* from every element.

OCL:
```
packageWBM
contextStructuredActivityNode
invself.getPredecessors()->exists(oclIsTypeOf(InitialNode)) and
self.getSuccessors()->exists(oclIsTypeOf(FinalNode))
```

OtherOCL:
```
contextStructuredActivityNode
inv connected models:
self.getPredecessors()->exists(n j n.oclIsTypeOf(InitialNode))
andself.getSuccessors()->exists(n j n.oclIsTypeOf(FinalNode))

context Action
def: getPredecessors() : Set(Action) =
self.inputControlPin.incoming.source.action-
>union(self.inputControlPin.incoming.source.action.getPredecessors())

context Action
def: getSuccessors() : Set(Action) =
self.outputControlPin.outgoing.target.action-
>union(self.outputControlPin.outgoing.target.action.getSuccessors())
```

In Constraint 7.4.1, our tool cannot generate two definitions given in the OtherOCL. The reason is that NL or English constraint should explicitly provide all detail for the target OCL constraint, e.g., user should use the defined keyword in the NL constraint.

Constraint 7.4.2

English: *Models with object flow are not supported.*

SBVR: It is obligatory models with object flow *are* not *supported*.

OCL:
```
package: WBM
context: Action
inv
```

OtherOCL:
```
context Action
inv self.inputObjectPin->isEmpty() and
self.outputObjectPin->isEmpty()
```

In Constraint 7.4.2, our tool cannot generate OCL as the NL description is very high level. Additionally, OCL involves the use of `inputObjectPin` and `outputObjectPin` but these two instances are not explicitly mentioned in NL constraint. Here, a message is given to the user that NL constraint is very high level and the provided information is incomplete. Moreover, the user should reconsider the NL constraint.

Constraint 7.4.3

English: *Process models with* Termination Nodes *are not supported. Use* FlowFinalNodes *instead.*

SBVR: It is obligatoryProcess models with TerminationNodes are *not supported*. *Use*FlowFinalNodes instead.

OCL:
```
package royal_and_loyal
contextProcessModel
invTerminationNode::allInstances()->isEmpty()
Endpackage
```

OtherOCL:
```
context ProcessModel
inv no termination:TerminationNode::allInstances()->isEmpty()
```

The syntax of NL description of Constraint 7.4.3 is bit similar to Constraint 7.4.2. However, in Constraint 7.4.2, required elements like `inputObjectPin` and `inputObjectPin` are not introduced but Constraint 7.4.3 includes elements like `TerminationNode`, etc. The correct OCL is generated for this NL constraint.

Constraint 7.4.4

English: *Models with implicit forks/joins/decisions/merges are not supported.*

SBVR: It is obligatory that models with implicit forks/joins/decisions/merges *are* not *supported*.

OCL:
```
package: WBM
context: StructuredActivityNode
invself.inputControlPin ->size() <= 1 andself.outputControlPin
->size() <= 1 andself.inputObjectPin ->size() <= 1 and
self.outputObjectPin ->size() <= 1
endpackage
```

OtherOCL:
```
contextStructuredActivityNode
invexplicit control flow :self.inputControlPin ->size() <= 1
andself.outputControlPin ->size() <= 1 andself.inputObjectPin -
>size() <= 1 andself.outputObjectPin ->size() <= 1
```

Similar to Constraint 7.4.2 in Constraint 7.4.4, the terms like 'forks', 'joins', 'decisions' and 'merges' are used. However the OCL involved instances like `inputControlPin`, `inputControlPin`, `inputObjectPin` and `outputObjectPin` respectively. As the NL does not provide specific information for this type of mapping such NL constraints cannot be translated to OCL. However, if we replace the terms like 'fork' with `inputControlPin`, our tool can easily generate OCL for it.

Constraint 7.4.5

English: *Loop nodes are not processed properly and cannot be merged. Therefore, loop nodes are not supported.*

SBVR: It is obligatory that loop nodes *are* not *processed* properly and *cannot be merged*. Therefore, loop nodes*are* not *supported*.

OCL:
```
package: WBM
context: ProcessModel
invLoopNode::allInstances()->isEmpty()
```

OtherOCL:
```
context ProcessModel
inv no_loops:LoopNode::allInstances()->isEmpty()
```

Similar to Constraint 7.4.2 and Constraint 7.4.3, again the similar structure "not supported" is introduced. However, similar to Constraint 7.4.3, the required element `LoopNode`is part of the NL constraint. The OCL generation was also simple for this constraint.

7.4.3 Quantitative Evaluation

In this subsection, we perform a quantitative evaluation of the NL2OCL approach for the WBM process model. For the WBM case study, only specification coverage is discussed here as the combined results of throughput measure for all three case studies have been shown in Table 7.5.

A. Specification Coverage

The NL approach was designed to automatically extract various OCL syntactic elements so that they can be mapped to a complete OCL expression. To translate English constraints of the WBM process model, most of the OCL syntax was covered in the implementation. Table 7.9 shows the details of the OCL elements implemented in our NL2OCL approach.

Table 7.9: OCL Generation: NL Approach vs. Pattern Approach

OCL Elements	NL2OCL Approach	Occurrences
Context	Yes	05
Navigation via association classes	Yes	16
Logical Expressions	Yes	5
Relational Expressions	Yes	4
Cardinality of Sets	Yes	1
Parameterized Function Calls	No	0
if-then-else	Yes	0
Enumerations	No	0
size()	Yes	4
isEmpty()	Yes	4
isNotEmpty()	Yes	0
forAll()	Yes	0
exists()	Yes	2
includesAll()	Yes	2
select()	Yes	0
asSet()	Yes	0
isUnique()	Yes	0
oclIsTypeOf()	No	2

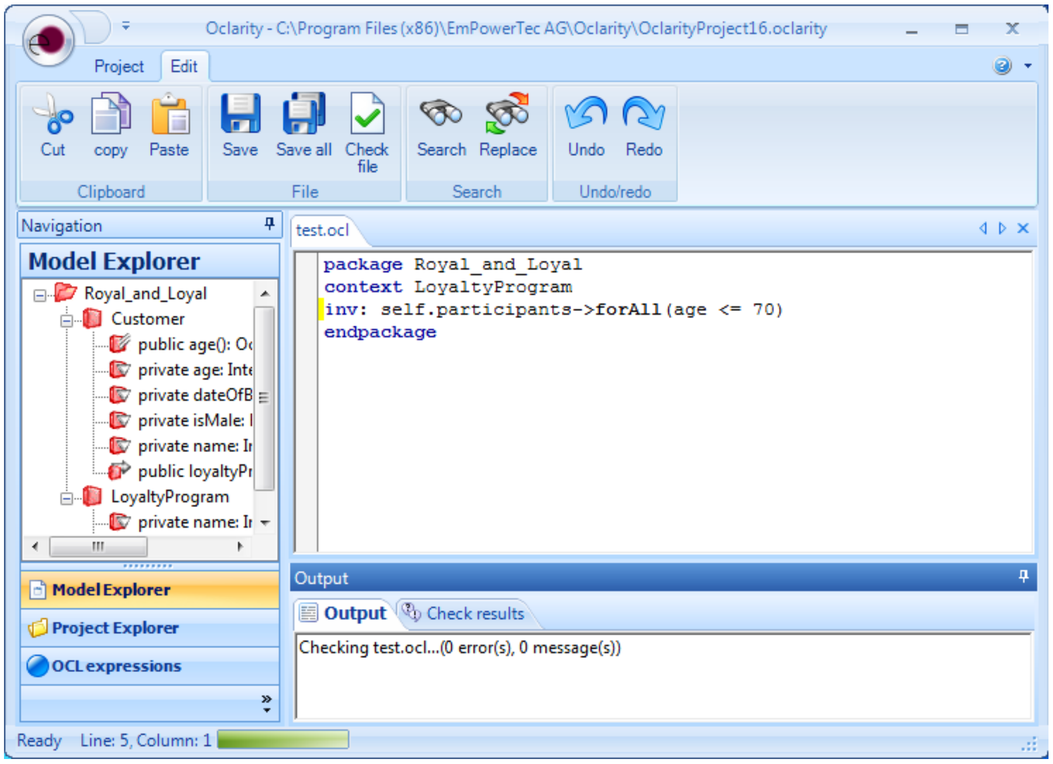
Our approach can generate three OCL constraints from the five English constraints of the WBM model that is 60%. The remaining 2 constraint, Constraint 7.4.2 andConstraint 7.4.4 cannot be translated due to the incomplete information and violation of one of the limitations of the NL approach discussed in section 7.4. The Constraint 7.4.2 is not possible to translate as it needs extra contextual information. However the Constraint 7.4.4 can be translated after tuning. Hence, after tuning we can translate four out of five constraints that results in 80% ratio.

7.5 Qualitative Evaluation

In this section, we apply the qualitative evaluation criteria defined in Section 7.1and summarize the validation of our approach.

A. Syntactic Accuracy

To check the syntactic accuracy of the OCL constraints, the OCL constraints of all three case studies (such as Royal and Loyal model, QUDV and Web Sphere modeller) are compiled with OCL compilers, OCLarity [EmPowerTec, 2010]& USE. An example of syntax checking by using OCLarity version 2.4 is shown in Figure 7.20.



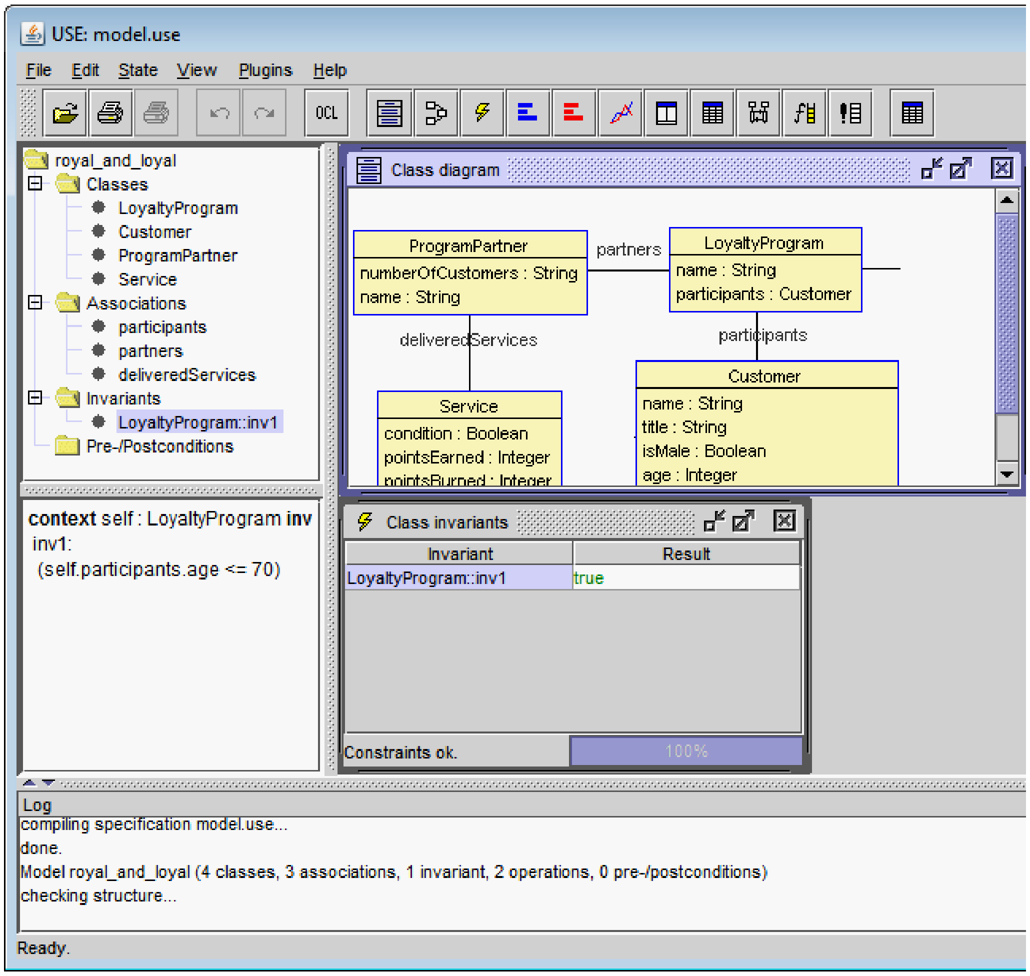
```
package Royal_and_Loyal
context LoyaltyProgram
inv: self.participants->forAll(age <= 70)
endpackage
```

```
Checking test.ocl...(0 error(s), 0 message(s))
```

Figure 7.20: OCL syntax checking

To check syntax of OCL invariants, we have given two inputs to the OCLarity tool: (1) XMI representation (.xmi) of the UML class model generated using Enterprise Architect. (2) A text file (.ocl) containing OCL invariant. If there are syntactical errors, the OCLarity tool highlights them, otherwise show `0` error(s) and `0` message(s).

An example of syntax checking by using USE version 3.0.1 is shown in Figure 7.21. There are five windows. The window on upper-left corner is project window showing the both inputs UML model and OCL. There is a Class Diagram window on upper right-corner that shows graphical representation of input Royal and Loyal model. The Class Invariant window that shows that result of input OCL invariant is true (see Figure 7.21) that means OCL syntax is correct. There is a log window at the bottom of Figure 7.21 showing the details of processing.



```
context self : LoyaltyProgram inv
 inv1:
  (self.participants.age <= 70)
```

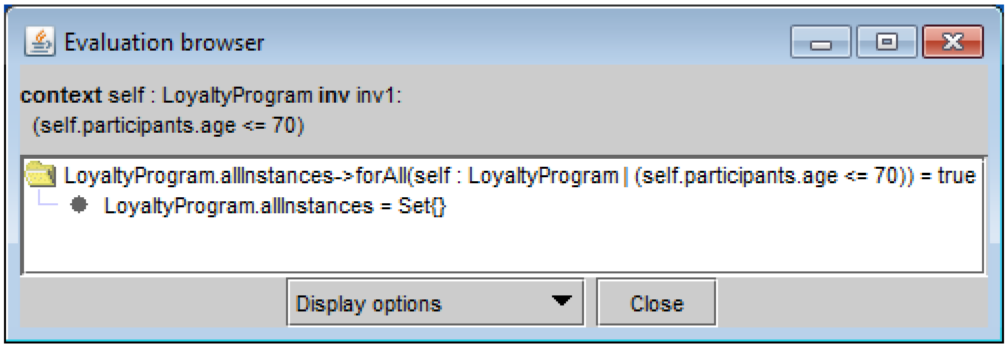
Invariant	Result
LoyaltyProgram::inv1	true

Constraints ok.

```
compiling specification model.use...
done.
Model royal_and_loyal (4 classes, 3 associations, 1 invariant, 2 operations, 0 pre-/postconditions)
checking structure...
```

Figure 7.21: OCL syntax checking

Figure 7.22 show the Evaluation browser of the USE tool that also shows the status of the OCL invariant is true.



```
context self : LoyaltyProgram inv inv1:
  (self.participants.age <= 70)
```

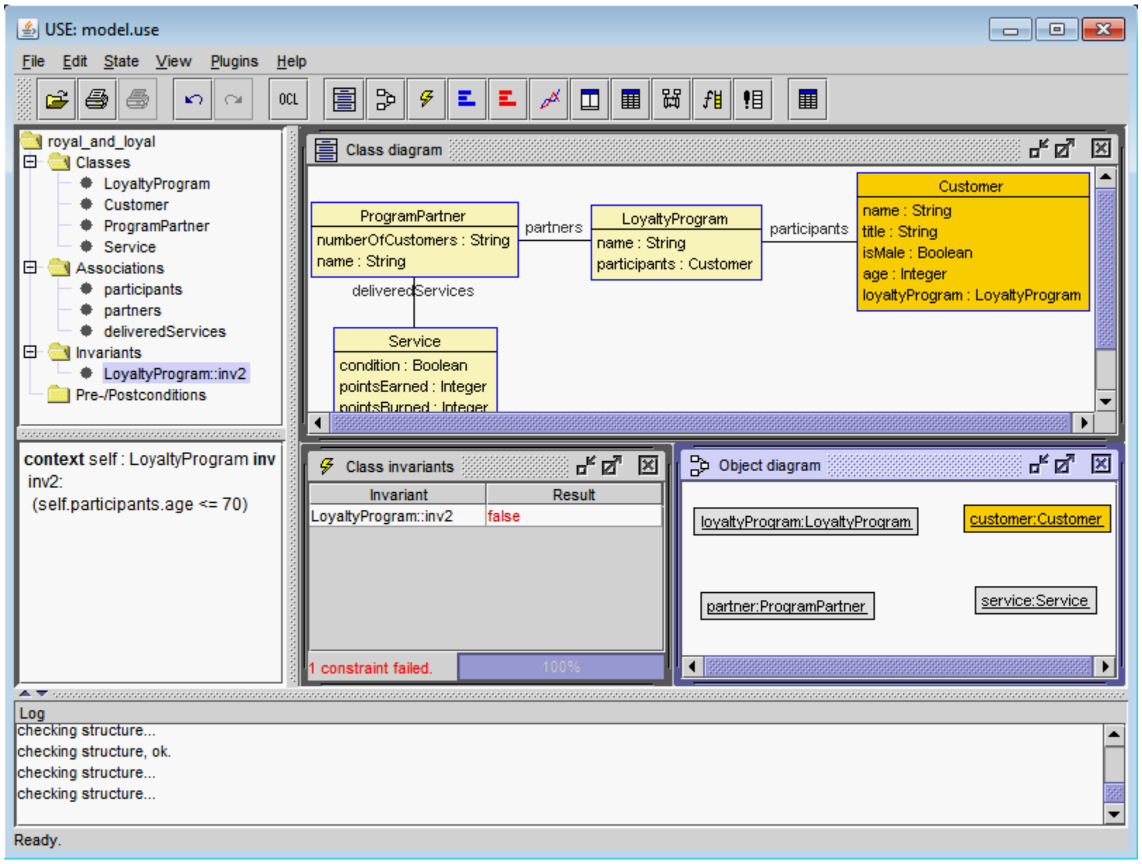
```
LoyaltyProgram.allInstances->forAll(self : LoyaltyProgram | (self.participants.age <= 70)) = true
    LoyaltyProgram.allInstances = Set{}
```

Figure 7.22: USE Evaluation Browser showing input invariant is True

B. Transformation Accuracy

Transformation correctness is measured by generating object diagrams in USE tool for both types of OCL: (1) OCL constraints generated by the NL2OCL*via*SBVR tool (2) The OtherOCL constraints generated by human expert. The procedure of the simple syntax checking of an OCL invariant in the USE tool is shown in Figure 7.21. Now, we aim to measure correctness of an OCL invariant using the USE tool.

The process of measuring correctness of transformation is very simple. A text file (.use) containing the details of input UMLclass model and OCL invariant is given to the USE tool. The USE tool reads the both inputs (class model and OCL invariant) from the (.use) file and checks OCL against the input UML class model. Here, we create an object diagram of the classes those are involved in the OCL invariant. As soon as, the objects are generated the Class Invariants window show that Result is "False" and status of OCL invariant is "1 constraint Failed" (see Figure 7.23). This status is because the values of attributes of the `customer` object and the `loyaltyProgram`object are undefined. Moreover the association between the `customer` object and the `loyaltyProgram`object is also not shown. Figure 7.24 shows that status of the constraint is `false` because it is shown that `(self.participants.age <= 70) = Undefined`. Figure 7.24 shows that the reason of failed status of the input OCL invariant is the undefined attributes of the `customer` object and the `loyaltyProgram` object.



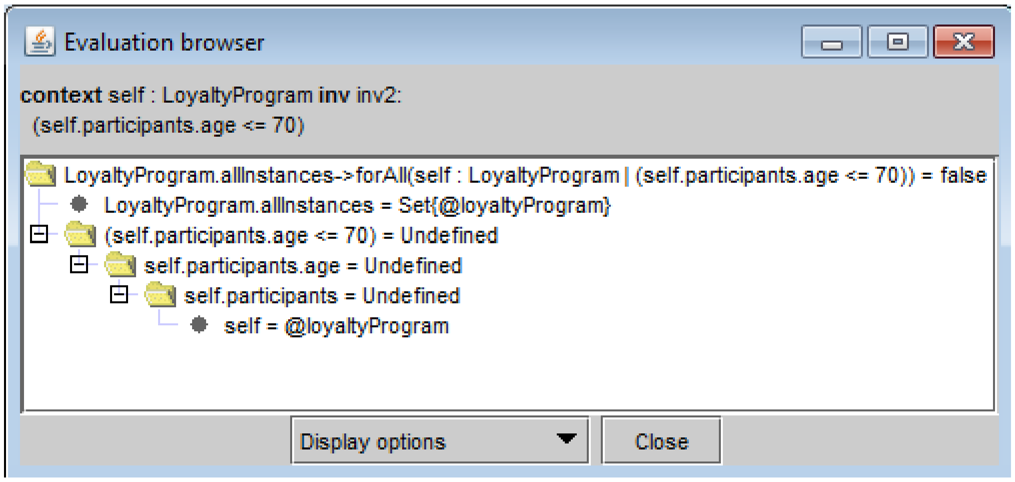
```
context self : LoyaltyProgram inv inv2:
 (self.participants.age <= 70)
```

Invariant	Result
LoyaltyProgram::inv2	false

1 constraint failed. 100%

```
Log
checking structure...
checking structure, ok.
checking structure...
checking structure...
Ready.
```

Figure 7.23: OCL verification using Object diagram in USE



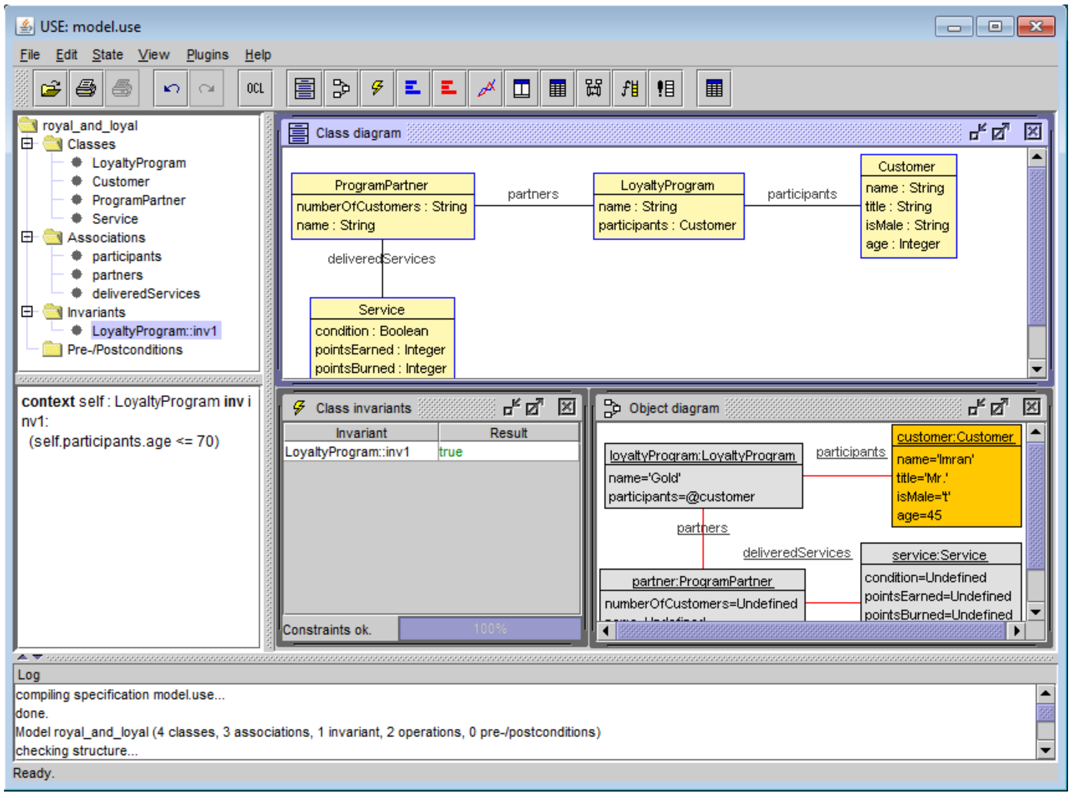
```
context self : LoyaltyProgram inv inv2:
 (self.participants.age <= 70)

LoyaltyProgram.allInstances->forAll(self : LoyaltyProgram | (self.participants.age <= 70)) = false
    LoyaltyProgram.allInstances = Set{@loyaltyProgram}
    (self.participants.age <= 70) = Undefined
        self.participants.age = Undefined
            self.participants = Undefined
                self = @loyaltyProgram
```

Figure 7.24: USE Evaluation Browser showing input Invariant is True

It is shown in Figure 7.25 that the attributes of the `customer` object are initialized with values such as `name='Imran'`, `title= 'Mr.'`, `isMale=t`, `age=45`. Similarly, the attributes of the `LoyaltyProgram` object are initialized with values such as `name='Gold'`, `particpants=`

`'@customer'`. Once the attributes of the `customer` object and the `loyaltyProgram` object are initialized, the status of the OCL Invariants window is changed to "Constraints OK" and now the Result is also again shown true. It is important to mention that the attribute age of the customer object is given value 45 that is less than 70. In case, the values is given more than 70 the status of the OCL Invariants window is again "1 constraint Failed" and now the Result is also again shown False.

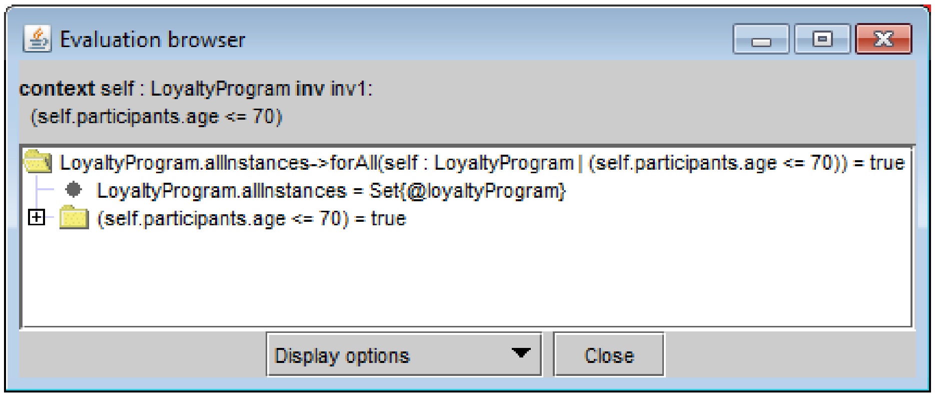


```
context self : LoyaltyProgram inv inv1:
 (self.participants.age <= 70)
```

Invariant	Result
LoyaltyProgram::inv1	true

```
compiling specification model.use...
done.
Model royal_and_loyal (4 classes, 3 associations, 1 invariant, 2 operations, 0 pre-/postconditions)
checking structure...
```

Figure 7.25: OCL verification using Object diagram in USE



```
context self : LoyaltyProgram inv inv1:
 (self.participants.age <= 70)

LoyaltyProgram.allInstances->forAll(self : LoyaltyProgram | (self.participants.age <= 70)) = true
    LoyaltyProgram.allInstances = Set{@loyaltyProgram}
    (self.participants.age <= 70) = true
```

Figure 7.26: USE Evaluation Browser showing input Invariant is True

C. Limitations of the Tool

The designed system NL2OCLviaSBVR is always capable of producing the wrong analysis but that in such circumstances the produced formal representation is correct for a particular, valid and potentially correct interpretation and can be corrected by manual intervention. In particular, we have identified a few cases where the designed system has tendency to not generate the incorrect interpretation due to the following limitations.

- The NL2OCL approach works for a restricted domain. i.e., UML Class Model. Hence, the NL constraints should not contain the vocabulary outside the UML class model.
- The vocabulary names used in the NL constraints should be consistent with the vocabulary names used in the UML class model.
- NL constraints should be complete such as a NL constraint should have at least one valid context.
- Incomplete (if one side of the relation is missing) and invalid (wrong direction of the relation) relations such as associations, aggregations are not supported
- NL constraints should not have discrepancies neither among the used elements nor between the UML class models.
- NL constraints should not involve UML enumerations.
- NL constraint should not involve parameterized function calls.
- XOR relations in NL constraints are not supported.
- The OCL operations`oclTypeof()`, `oclIsKindOf(T)`, `oclIsTypeOf(T)`, `oclAsType(T)`, `oclInState(s)`, `sortBy()`, `count()`, `collect()`, `reject()`, `and append()`are not supported in the tool.
- NL sentences should be declarative or imperative. The question based sentences are not processed.

There are some limitations of the tool due to the use of the Stanford parser as a library. Major limitations of the Stanford parser in a role of NLP plugin are below:

- A few times, the Stanford parser does not produce the right output after POS tagging due to lexical ambiguity in NL sentences. Since, the Stanford parser does handle lexical ambiguity by making a decision. However, this decision might be incorrect as it is not according to the interpretation the author intended.

- The NL2OCL approach is based on dependencies generated by the Stanford parser but the wrong typed dependencies are generated by the Stanford parser possibly due to semantic ambiguities. In such particular cases, due to wrong dependencies, wrong labeling of semantic roles can happen that result in irresolution of NL quantifiers, etc., in NL sentences.
- Since, the Stanford parser is not typically designed for the task we need, it does not generate correct output in case of some other ambiguities in NL sentences such as homonymy.

In this thesis, we have presented a novel approach to handle such ambiguities for which the Stanford parser does not produce the right output by using the information in the UML class model. However, there is a possibility that UML model does not contain the information to resolve a NL ambiguity and produce incorrect interpretation, and in such cases the user is involved to correct the output manually.

7.6 Summary

In this chapter, three case studies are presented from various dimensions such as software engineering, measuring systems, and business processes domain. The results of the case studies manifest that a natural language based approach to generate OCL constraints significantly cannot only help in improving usability of OCL but also outperforms the most closely related techniques in terms of effort and effectiveness required in generating OCL. Though, the researcher has used NL and automated generated SBVR in pair to resolve NL ambiguities and clarify vagueness by pointing them out, this will not be a 100% solution either and the researcher is aware of it.

CHAPTER 8
DISCUSSION

This chapter discusses the key contributions to knowledge by our presented approach for automated transformation of NL to OCL constraints in Section 8.1. Moreover, the researcher also presents an account of the possible future enhancements in the NL2OCL approach and its implementation in Section 8.2.

8.1 Contribution to Knowledge

In this section, the researcher discusses the contributions made in this thesis and he also explains the significance of the each contribution. These contributions have been divided in ten distinct areas and aspects.

8.1.1 Specifying Constraints using Natural Language

The researcher has designed the NL2OCL approach that can process NL specification of constraints, extracts various parts of an OCL constraint and then finally integrates those parts to generate a complete OCL constraint using model transformation technology [Bajwa, 2010]. A novelty, in the NL to OCL transformation approach is the use of SBVR as the pivotal representation. The use of SBVR facilitates the transformation from a natural language to a formal language such as OCL on account of its foundation on formal logic.

To the best of the researcher's knowledge, this is the first approach to generate automatically OCL constraints from NL specification in compliance with the target UML class model. The presented approach not only simplifies the process of OCL constraints specification and assists modellers in process of software and business modelling but also can facilitate the novice users who do not have enough expertise to write OCL constraints. Moreover, the researcher is aware that such solution cannot be 100% correct, still it can be helpful for the software designers and developers by assisting them in writing OCL constraints.

8.1.2 Resolving Syntactic Ambiguities

The researcher has identified various cases of syntactic ambiguity in the NL constraints which are not addressed by used off-the-shelf components; the Stanford POS tagger and the Stanford parser. The identified cases of syntactic ambiguity are very common in NL statements. In the identified cases the wrong POS tags identified by the Stanford POS tagger due to homonymy in NL constraints cause generation of wrong parser tree and wrong set of dependencies which result in a wrong semantic analysis and finally lead to a wrong OCL constraint [Bajwa, 2012a]. Similarly, the researcher has also identified various cases where the typed dependencies are wrongly identified by the Stanford parser due to the attachment ambiguity in NL constraints [ibid]. Further, the researcher has developed a novel approach to deal with identified cases of homonymy and attachment ambiguity. As the identified cases are due to absence of context, the used approach to solve syntactic ambiguities involves the information (classes, attributes, methods, associations, etc.) given in a UML class model as a context and successfully addresses the ambiguities.

8.1.3 Semantic Analysis of NL Constraints

The researcher has developed a novel approach for detailed semantic analysis of the NL constraints [Bajwa, 2012b]. The used approach works into two phases: (1) shallow semantic parsing to assign SBVR vocabulary based semantic roles to various parts of a NL constraint; and (2) detailed semantic parsing involving quantifier scope resolution and generation of a logical representation based on SBVR vocabulary. Besides OCL, the SBVR based logical representation can be mapped to any formal language such as Alloy, B, etc. A modeller just need to write a set of transformation rules for the required transformation.

The semantic analysis of NL constraints plays a key role in NL to OCL transformation as the semantic analysis helps to identify various SBVR constructs which are later on mapped to OCL. Moreover, during semantic analysis the researcher has also mapped the information of NL constraints with UML class model to ensure that the generated OCL constraints comply with the target UML class model. Furthermore, in semantic analysis of NL constraints, it is decided that an NL constraint is mapped to which category of OCL constraints: invariant, pre-condition or a post-condition.

8.1.4 Resolving Semantic Ambiguities

In English to OCL translation, our contribution is a semantic analyser that uses the output of the Stanford parser for shallow and deep semantic parsing. Our analysis of the output of shallow semantic parsing showed that semantic roles were miss-identified for a few English constraints due to various semantic ambiguities [Bajwa, 2012c]. Similarly, in deep semantic parsing, it is difficult to resolve scope of quantifier operators due to scope ambiguity that is another sub-type of semantic ambiguity. To resolve, identified cases of semantic ambiguities, we have used the metadata (classes, attributes, methods, associations, etc.) of the target UML class model. The resolution of semantic ambiguities is explained in detail in Section 4.3.

8.1.5 Identifying Logical Contradictions in Constraints

OCL is a side-effect free language and it compliments UML class models. Owing to this feature of OCL, an OCL constraint should always comply with the target UML class model for which the OCL constraint has been written. Similarly, to generate an OCL constraint (that conforms to the target UML class model) from a NL constraint, the specification of NL constraint should also conform to the target UML class model. However, the researcher has identified a few cases where a NL constraint cannot be mapped to an OCL constraint due to some discrepancies or logical contradiction in a NL constraint and a UML class model. Examples of such cases are Constraints 7.2.12. In NL constraints, the logical contradictions can be of various types. A most common type that the researcher has identified is the relationship in entities given in a NL constraint does not exist in the UML class model. The NL2OCL can identify such cases and intimates a user to revise the NL constraint or revise the UML class model.

8.1.6 SBVR based Logical Representation

A logical representation based on SBVR vocabulary is introduced in the researcher's approach. SBVR is a recent standard introduced by OMG to provide a formal notation to express business and software specifications. Using a standardized representation in a logical form is a novel idea.

Various automated transformation from SBVR to other standards and formal languages such as UML, BPMN have already been introduced. Hence, SBVR vocabulary based logical representation can really make it easy to transform NL specification business and software requirements to map to other standards such as UML, BPMN, Alloy, etc. In this thesis, the researcher has mapped SBVR based logical representation to SBVR rules and OCL constraints.

The use of SBVR in a logical representation is also beneficial in a way that SBVR has a defined metamodel and SBVR based representation can simply be mapped to other formal languages using model transformation technology.

8.1.7 Specifying SBVR Rules using NL Approach

The researcher has explained in Section 8.1.6 that he has used SBVR as a pivotal representation in transformation of NL constraints to OCL. SBVR plays a key role in NL to OCL transformation. However, as a by-product this approach also generates SBVR business rules from NL specifications of Business/Software requirements. Automated generation of Business Rules is itself a challenging task and an open research question. To the researcher's best of knowledge, nobody has presented any approach for automated generation of SBVR business rules from NL specification. The approach for automated generation of SBVR business rules can not only assist business modellers but also can simplify the processing of specifying business rules with a formal notation. However, even the NL representation and automated generated SBVR representation in pair can help in resolving ambiguities and clarifying vagueness by pointing them out. However, this will not be a 100% solution either.

8.1.8 SBVR to OCL Transformation Rules

From SBVR to OCL transformation, a challenging task was to map SBVR to OCL that has not been previously done. As the researcher has used model transformation technology to map SBVR to OCL, he needed to define a set of transformation rules to map each SBVR constructs to respective OCL construct [Bajwa, 2011b]. The defined set of model transformation rules play a key role in transformation a SBVR business rule to an OCL expressions and these transformation rules has been implemented using *Si**T*ra library.

8.1.9 The NL2OCL*via*SBVR Tool and Evaluation

The researcher has implemented the NL2OCL approach in Java. The Java based NL2OCLviaSBVR tool is an Eclipse plugin as a proof of concept and the tool can be used with other Eclipse based modelling tools. Evaluation criteria have been presented to evaluate the performance of NL2OCL approach in terms of qualitative and quantitative measures.

8.1.10 Case Studies and their Results

Three case studies were performed for the sake of evaluation. Besides, Royal & Loyal case study, another two case studies are performed. All three case studies from three different domains, and NL constraints each case study had different nature of NL constructs. The purpose to do three different case studies was to test the performance of the NL2OCL approach. The objective to select three different case studies was to test the performance of the approach for constraints from different domains. The selected cases studies are famous case studies in the respective domain and already under research by IBM, NASA, ESA, etc. The selected case studies have also been discussed in various PhD thesis and books and have been done with other comparable techniques. Hence, it is simple to compare the performance of our tool against some standard case studies.

8.2 Future Enhancements

To make the researcher's tool more comprehensive so that it may model constraints, he proposes a few enhancements in future. Following is a brief overview of the possible enhancements in the NL2OCL approach.

8.2.1 Multiple Sentence based NL constraints

The current version of NL2OCL approach can process single sentence-based NL constraints. However, there can be a few constraints which may have multiple sentence-based NL specifications. To process multiple sentences is a NLP research question. In future, the researcher aims to provide this ability in current approach as well.

8.2.2 Improving Semantic Analysis

The researcher has discussed in Section 8.3.2 and Section 8.3.3that there are a few NL constraints that involve contextual information which cannot be processed without involvement of its context. Moreover, there are a few terms and phrases that change its meanings with the change in domain. For example a word ‘sentence’ has different meanings in law domain but it is differently perceived in other domains. The researcher proposes the use of domain specific semantic dictionaries or ontologies.

CHAPTER 9
CONCLUSION

This chapter concludes the research work presented in the previous chapters. In this thesis, the researcher has addressed the problem of OCL usability as it is difficult to write OCL, typically for the novice users. In this thesis, the researcher also present a NLP based approach, called the NL2OCL approach, to address this problem and the researcher presents implementation of the approach, called the NL2OCL*via*SBVR, as well.

This research thesis presents a framework for dynamic generation of the OCL constraints from the NL specification provided by the user. Here, the user is supposed to write simple and grammatically correct English. The designed system can find out the noun concepts, Individual Concepts, verbs and adjectives from the NL text and generate a structural or behavioral rule according to the nature of the input text. This extracted information is further incorporated to constitute a complete SBVR rule. The SBVR rules are finally translated to OCL expressions. SBVR to OCL translation involves the extraction of OCL syntax related information, i.e., OCL context, OCL invariant, OCL collection, OCL types, etc. and then the extracted information is composed to generate a complete OCL constraint, or pre/post-condition.

As this thesis aims to address a major challenge related to usability of OCL, the researcher has presented a method of applying model transformations to create OCL statement from Natural Language expressions. The presented transformation makes use of SBVR as an intermediate step to highlight the syntactic elements of natural languages and make NL controlled and domain Specific. The use of automated model transformations ensures seamless creation of OCL statements and deemed to be non-intrusive. The presented method is implemented as prototype tool which is being extended to be integrated into the existing tools. As a next step, the researcher is hoping to investigate usability aspects of the tool directly via empirical methods involving teams of developers.

The results of the experiments indicate that a NL based solution to generate OCL can soften the process of writing constraints for UML models. Even the NL2OCL approach is accurate up to 96%, the researcher is aware of the fact that the NL2OCL approach used to generate OCL from NL constraints cannot be 100% correct. Furthermore, the researcher has used NL and automated generated SBVR in pair to resolve NL ambiguities and clarify vagueness by pointing them out, this will not be a 100% solution either and the researcher is aware of it.

References

Akehurst, D.H., Boardbar, B., Evans, M., Howells, W.G.J., McDonald-Maier, K.D. (2006). SiTra: Simple Transformations in Java, in ACM/IEEE 9TH International Conference on Model Driven Engineering Languages and Systems, LNCS, Vol. 4199, pages 351-364, 2006

Anastasakis, K., Bordbar, B., Georg, G. and Ray, I. (2007). UML2Alloy: A Challenging Model Transformation, ACM/IEEE 10TH International Conference on Model Driven Engineering Languages and Systems, LNCS, Vol. 4735, pages 436-450, 2007

Ambler, S.W (2003). Business Rules, Available at: http://www.agilemodeling.com/artifacts/ businessRule.htm, Accessed on Dec, 2010.

Aydal E. G., Paige R. F., Woodcock J. (2008). Evaluation of OCL for Large-Scale Modelling: A Different View of the Mondex Purse. LNCS - Models in Software Engineering. 5002/2008, 194-205.

Baar T., Chiorean D., etal.. (2006). LNCS - Tool Support for OCL and Related Formalisms – Needs and Trends. Satellite Events at the MoDELS 2005 Conference. 3844/2006, 1-9.

Baayen, R.H. (1991), 'De CELEX Lexicale Databank'. Forum der Letteren 32, 221-231.

Bajwa, I.S., Bordbar. B., Lee, M. (2010). OCL Constraints Generation from Natural Language Specification. in IEEE/ACM 14th International EDOC Conference 2010, Vitoria, Brazil, October 2010, pp:204-213

Bajwa, I.S., Lee, M., Bordbar. B. (2011)."SBVR Business Rules Generation from Natural Language Specification", in proceedings of AAAI Spring Symposium 2011 – Artificial Intelligence for Business Agility (AI4BA), San Francisco, USA, March 2011, pp:2-8.

Bajwa, I.S., Lee, M. (2011). "Transformation Rules for Translating Business Rules to OCL Constraints", in ECMFA 2011 - 7th European Conference on Modelling Foundations and Applications, Birmingham, UK, June 2011, pp:132-143

Bajwa, I.S., Bordbar. B., Lee, M. (2011), “SBVR vs OCL: A Comparative Analysis of Standards”, in 14th IEEE International Multi-topic Conference (INMIC 2011), Dec 2011, Karachi, Pakistan, pp:261-266

Bajwa, I.S., Lee, M., Bordbar. B. (2012). "Resolving Syntactic Ambiguities in NL Specification of Constraints using UML Class Model" in CICLING 2012 - 13th International Conference on Computational Linguistics and Intelligent Text Processing, March 2012, Delhi, India, pp:178-187

Bajwa, I.S., Lee, M., Bordbar. B.(2012). “Semantic Analysis of Software Constraints”, The 25th International FLAIRS Conference, May 2012, pp:13-18, Florida, USA

Bajwa, I.S., Lee, M., Bordbar. B.(2012). “Addressing Semantic Ambiguities in English Constraints”, The 25th International FLAIRS Conference, May 2012, Florida, USA

Bajwa, I.S., Lee, M., Bordbar. B.(2012). Translating Natural Language Constraints to OCL”, Journal of King Saud University - Computer and Information Sciences, June 2012, 24(2): Elsevier

Bajwa, I.S., Bordbar. B., Lee, M. (2012).On a Chain of Transformations for Generating Alloy from NL Constraints”, 7th IEEE ICDIM 2012, Macau [Submitted]

Bar-Hillel, Y. (1953). A quasi-arithmetical notation for syntactic description. Language, 29, 47-58.

Bel N., Marimon M., Porta, J. (1996), Etiquetado morfosintáctico de corpus en el proyecto

Burke D. A., Johannisson K. (2005). Translating Formal Software Specifications to Natural Language. LNCS - Logical Aspects of Computational Linguistics. 3492/2005, 51-66.

Cabot J., Teniente E. (2007). Transformation techniques for OCL constraints. Science of Computer Programming. 68 (3), 152-168.

Cabot J., Teniente E. (2009). Incremental Integrity Checking of UML/OCL Conceptual Schemas. Systems and Software. 82 (9), 1459-1478.

Cate B. T., Kolaitis P. G. (2009). Structural characterizations of schema-mapping languages. 12th International Conference on Database Theory. 361, 63-72.

Ceponiene L., Nemuraite L., Vedrickas G. (2009). Separation of Event and Constraint Rules in UML&OCL Models of Service Oriented Information Systems. Information Technology and Control. 38 (1), 29-37.

Cer, D., Marneffe, M.C., Jurafsky, D. and Manning, C.D. (2010). Parsing to Stanford Dependencies: Trade-offs between speed and accuracy." In Proceedings of LREC-10.

Chapin, D. (2008). SBVR: What is now Possible and Why? Business Rules Journal, Vol. 9, No. 3 (Mar. 2008), URL: http://www.brcommunity.com/a2007/b407.html

Chomsky, N. (1965). Aspects of the Theory of Syntax. MIT Press, Cambridge, Mass, 1965.

Chow, C., & Liu, C. (1968) Approximating discrete probability distributions with dependence trees. IEEE Transactions on Information Theory, 1968, IT-14(3), 462–467.

Clavel M., Egea M., and da Silva V. T. (2007). MOVA: A Tool for Modeling, Measuring and Validating UML Class Diagrams. 12th Conference on Software Engineering and Databases., 393-394.

Correa, A., Werner, C., and Barros, M. (2007). An Empirical Study of the Impact of OCL Smells and Refactorings on the Understandability of OCL Specifications. Model Driven Engineering Languages and Systems, LNCS 2007, Volume 4735/2007, pp:76-90

Delisle, S., Barker, K., Biskri, I.(1999). Object-Oriented Analysis: Getting Help from Robust Computational Linguistic Tools. The Fourth International Conference on Applications of Natural Language to Information Systems.

Demuth B., Wilke C. (2009). Model and Object Verification by Using Dresden OCL. Russian-German Workshop Innovation Information Technologies: Theory and Practice. 81-89.

EmPowerTec AG (2010), OCLarity: An OCL Authoring Environment, Version 2.4. Availableat http://www.empowertec.de/products/oclarity/. Accessed on 18 April, 2012

Engels G., Heckel R., Küster J. M. (2001). Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. LNCS - «UML» 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 2185/2001 (1), 272-286.

Fillmore, C.J., Johnson, C.R. and Petruck, M.R.L. (2003). Background to FrameNet. International Journal of Lexicography, 16(3).

Fellbaum, C. (1998) “WordNet: An Electronic Lexical Database” MIT Press, Cambridge, MA.

Giuglea, A., Moschitti, A. (2006). Shallow Semantic Parsing Based on FrameNet, VerbNet and PropBank. Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 -- September 1, 2006, Riva del Garda, Italy

Gogolla M., Büttnera F., Richtersb M. (2007). USE: A UML-based specification environment for validating UML and OCL. Science of Computer Programming. 69 (1-3), 27-34.

Guthrie D., Allison B., Liu W., Guthrie L., Wilks Y. (2006). A Closer Look at Skip-gram Modelling. Fifth International Conference on Language Resources and Evaluation (LREC--2006), Genoa, Italy, 1222-1225

Harmain H. M., Gaizauskas R. (2003). CM-Builder: A Natural Language-Based CASE Tool for Object-Oriented Analysis. Automated Software Engineering. 10 (2), 157-181.

Hart G., Johnson M., Dolbear C. (2008). Rabbit: Developing a Control Natural Language for Authoring Ontologies. 5th European Semantic Web Conference (ESWC'08). 348-360.

Hamie, A. (2004). Mapping OCL-constrained models to JML specifications. 7th world conference on integrated design and process technology (IDPT 2003), Austin, Texas

Heidenreich, F., Wende, C., Demuth, B., (2008). A Framework for Generating Query Language Code from OCL Invariants, Electronic Communications of the EASST9, 2008

IBM. (2009). Software Development. Available: http://www-01.ibm.com/software/awdtools/library/standards/. Last accessed 20 Nov 2009.

Ilieva, M.G., O. Ormandjieva. (2005). Automatic Transition of Natural Language Software Requirements Specification into Formal Presentation. LNCS - Natural Language Processing and Information Systems. 3513/2005, 392-397

Jelinek, F. and Lafferty J.D. (1991). Computation of the probability of initial sub-string generation by stochastic context-free grammars. Computational linguistics.17 (3), 315-323

Johannisson K. (2004). Disambiguating Implicit Constructions in OCL. Conference on OCL and Model Driven Engineering 2004. 30-44.

Jurafsky, D., Martin J. (2000). Speech and Language Processing. New York: Prentice Hall.

Juristo, N., Moreno,A.M. and López, M. (2000) How to use linguistic instruments for object-oriented analysis, IEEE Software June, June 200, pp:80-89

Karlsson, F. (1995). Constraint Grammar: A language Independent system for parsing Unrestricted Text. 13th International Conference of Computational Linguistics, 3. 168-173.

Karttunen, L. (1983). KIMMO: A General Morphological Processor. Texas Linguistic Forum 22, 165-186

Kiyavitskaya, N., Zeni, N., Mich, L., Berry, D. (2008). Requirements for tools for ambiguity identification and measurement in natural language requirements specifications, Requirements Engineering, Vol. 13, No. 3. (2008), pp. 207-239.

Koning, H.P. Rouquette, N., Burkhart, R., Espinoza, H., (2005). Library for Quantity Kinds and Units: schema, based on QUDV model OMG SysML(TM), Version 1.2. Available at: http://www.w3.org/2005/ Incubator/ssn/ssnx/qu/qu. Accessed on: 12 Nov, 2011.

Kleiner M., Albert P., Bézivin J. (2009). Parsing SBVR-Based Controlled Languages. LNCS - Model Driven Engineering Languages and Systems. 5795/2009 , 122-136.

Kuchmann-Beauger, N., Aufaure, M. (2011) A natural language interface for data warehouse question answering, NLDB'11 Proceedings of the 16th international conference on Natural language processing and information systems, Alicante, Spain

Kuhlmann, M. and Gogolla, M. (2008). Modeling and Validating Mondex Scenarios Described in UML and OCL with USE. *Formal Aspects of Computing*, 20(1):79-100, 2008.

Li K., Dewar R. G., Pooley R. J. (2004). Object-Oriented Analysis Using Natural Language Processing. Heriot-WattUniversity Technical Reports. www.macs.hw.ac.uk:8080/ techreps/ docs/ files/HW MACS-TR-0033.pdf

Lovins, J. B. (1968). Development of a Stemming Algorithm. Mechanical Translation and Computational Linguistics. 11(1), 22-31.

Manning, C.D. (2011). Part-of-Speech Tagging from 97% to 100%: Is It Time for Some Linguistics? In proceedings of CICLing (1) 2011. pp.171~189

Marinos, A., Gazzard, P., Krause, P. (2011). An SBVR Editor with Highlighting and Auto-completion. RuleML 2011

Marneffe, M.C., MacCartney Bill and Manning, C.D. (2006). Generating Typed Dependency Parses from Phrase Structure Parses. In LREC 2006.

Mich L. (1996). NL-OOPS: from natural language to object oriented requirements using the natural language processing system LOLITA. Natural Language Engineering. 2 (2), 167-181.

Moschoyiannis, S., Marinos, A. and Krause, P. (2010). Generating SQL Queries from SBVR Rules. 2010 international conference on Semantic web (RuleML'10). LNCS, Vol. 6403/2010, pp:128-143

MULTEXT'. Actas del 28th Simposio de la Sociedad Española de Lingüística, Madrid

Nentwich, C., James, R. (2010) Natural Rule Language (NRL). Version 1.4.0, Specification 7 April 2010. Available at: http://nrl.sourceforge.net/spec/

Nihalani, N., Silakari, S., Motwani, Mahesh (2011). Natural language Interface for Database: A Brief review, IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 2, March 2011

NRL Community (2005): The Natural Rule Language (NRL): Basic User Guide, Avaialble at: http://nrl.sourceforge.net/OMG. (2006). Object Constraint Language. (OCL) Standard v. 2.0, Object Management Group, Available: http://www.omg.org/spec/OCL/2.0/

OMG. (2007). Unified Modelling Language. (UML) Superstructure v. 2.1, Object Management Group. Available: http://www.omg.org/technology/documents/formal/uml.htm

OMG. (2008). Semantics of Business vocabulary and Rules. (SBVR) Standard v.1.0. Object Management Group, Available: http://www.omg.org/spec/SBVR/1.0/

OMG. (2010). SysML (System Modelling Language) v. 1.2, Available at: http://www.omgsysml.org/

Pau R., Cabot J. (2008). Paraphrasing OCL Expressions with SBVR. 13th international conference on Natural Language and Information Systems: Applications of Natural Language to Information Systems. (NLDB'08), LNCS 5039, pp. 311-316

Perez-Gonzalez, H. G., Kalita J. K. (2002). Automatically Generating Object Models from Natural Language Analysis.17th annual ACM SIGPLAN conference, OOP, Systems, languages & applications, 86-87.

Porter, M. F. (1997). An Algorithm for Suffix Stripping. Morgan Kaufmann Multimedia Information And Systems Series, Morgan Kaufmann Publishers, San Francisco, CA, 313-316.

Popescu, A., Etzioni, O, Kautz, H. (2003). Towards a theory of natural language interfaces to databases, Proceedings of the 8th international conference on Intelligent user interfaces, January 12-15, 2003, Miami, Florida, USA

Rompaey B. V., Bois B. V., Demeyer S., Rieger M. (2000). On the Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. EEE Transactions on Software Engineering. 33 (12), 800-817.

Seco N., Gomes P., Pereira F.C. (2004). Using CBR for Semantic Analysis of Software Specifications. LNCS - Advances in Case-Based Reasoning. 3155/2004, 41-43.

Shah, S.M.A., Anastasakis, K., Bordbar, B. (2009). From UML to Alloy and Back, 6th Workshop on Model Design, Verification and Validation (MODEVVA 09) published in ACM International Conference Proceeding Series; Vol. 413, pages 1-10, 2009

Sleator, D. and Temperley, D. (1993). Parsing English with a Link Grammar", In proceedings, Third International Workshop on Parsing technologies, Tilburg, The Natherland-Durbuy Belgium.

Spreeuwenburg S., Healy K. A. (2009). SBVR's Approach to Controlled Natural Language. Workshop on Controlled Natural Language 2009, Marettimo Island, Italy Available: http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-448/paper26.pdf

Steen, B., Pires, L.F. and Iacob, M. [2010] Automatic generation of optimal business processes from business rules. 14th IEEE International Enterprise Distributed Object Computing Conference Workshops, EDOCW 2010, 25-29 Oct 2010, Vitoria, Brazil.

Steinberg, D., et al. (2008). EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Professional, 2 edn.

Stolcke, A. (1995). An efficient probabilistic context-free parsing algorithm that computes pre-fix probabilities. Computational Linguistics. 21(2), 165-202

Surdeanu, M., Harabagiu,S., Williams, J., Aarseth, P. (2003). Using Predicate-Argument Structures for Information Extraction. In Proceedings of the 41st Annual Meeting on Association for Computational Linguistics (ACL'03) - Volume 1

Tommasi, M.D. and Corallo, A. (2006). SBEAVER: A Tool for Modeling Business Vocabularies and Business Rules. in 10th International Conference on Knowledge-Based Intelligent Information And Engineering Systems. LNCS, 2006, Vol. 4253/2006, pp:1083-1091

Toutanova, K. and Manning, C.D. (2000). Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. In Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000), pp. 63-70.

Toutanova, K., Klein, D., Manning, C.D. and Singer, Y. (2003). Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In Proceedings of HLT-NAACL 2003, pp. 252-259.

Uejima, H. , Miura, T., Shioya, I. (2003). Improving text categorization by resolving semantic ambiguity Communications, Computers and signal Processing, 2003 pp. 796-799

Vaziri, M., Vaziri, A., Jackson, D. (1999). Some Shortcomings of OCL, the Object Constraint Language of UML. Technology of Object-Oriented Languages and Systems (TOOLS '00), pp:555

Villalta, E. (2007). Restoring Indefinites to Normalcy: An Experimental Study on the Scope of Spanish algunosJ Semantics. 24(1): 1-25

Wahler M. (2008). Using Patterns to Develop Consistent Design Constraints, PhD Thesis, ETH Zurich, Switzerland Available: http://e-collection.ethbib.ethz.ch/view/eth:30499

Warmer J., Kleppe A. (2003). The Object Constraint Language – Getting Your Models Ready for MDA. Second Edition, Addison Wesley, Boston, MA, USA

AppendixA
Summaries of the Published Work

Following are summaries of the papers published during PhD research. These papers address various parts of the research.

A. 1 OCL Constraints Generation fromNatural Language Specification

IEEE/ACM 14thInternational EDOC Conference 2010, Vitoria, Brazil, October 2010, pp:204-213

The first paper was published in EDOC 2010, held in Brazil. The paper was based on the main idea of research that NL constraints can be automatically translated to OCL constraints [Bajwa, 2010]. In this paper, the researcher presented how SBVR can play a useful role in translation of NL specification of constraints to OCL invariants and OCL pre/post conditions.

This research paper presents a framework for dynamic generation of the OCL constraints from the NL specification provided by the user. Here, the user is supposed to write simple and grammatically correct English. The designed system can find out the noun concepts, Individual Concepts, verbs and adjectives from the NL text and generate a structural or behavioral rule according to the nature of the input text. This extracted information is further incorporated to constitute a complete SBVR rule. The SBVR rules are finally translated to OCL expressions. SBVR to OCL translation involves the extraction of OCL syntax related information i.e. OCL context, OCL invariant, OCL collection, OCL types, etc. and then the extracted information is composed to generate a complete OCL constraint, or pre/post-condition.

A. 2 SBVR Business Rules Generation from Natural Language Specification

AAAI Spring Symposium 2011 – AI4BA, San Francisco, USA, March 2011, pp:2-8

In 2011, a paper was published in AAAI spring symposium, held in USA that was addressing the SBVR rules aspect of this research [Bajwa, 2011a]. In this research, the researcher is generating SBVR rules from NL constraints and then mapping such SBVR rules to OCL. The NL2SBVR is a modular NL-based approach that generates SBVR business rules from English text with respect to a target Business domain. It takes two inputs: a single English statement and a UML class model. Here English statement is English specification of a business rule and the UML class model provides a business domain. To process the input English text first it is linguistically analyzed. In linguistic analysis of the English text, the English text is Parts-Of-Speech (POS) tagged. Then a rule based parser is used to further process the POS tagged information to extract basic SBVR elements e.g. noun concept, fact type, etc. Here, the SBVR vocabulary is mapped to a SBVR rule. Finally, to generate an SBVR business rules, the SBVR vocabulary is mapped to SBVR elements using the rule-based approach. These steps can be summarized as follows.

A.2.1 The Input Documents

The NL2SBVR approach takes two input documents: an English text document (.txt file) and a UML class model (.ecore file). The English text is taken as a plain text file containing only English constraint. Current version of the RuleGenerator handles only one English constraint at a time. The given English text should be grammatically correct. UML model is taken as ECORE or XMI format. We used Eclipse UML2 Ecore Editor to create a UML model and export it in XMI format.

A.2.2 The NLP Module

The core of NL2SBVR approach is a NLP module that consists of a number of processing units organized in a pipelined architecture. This NLP module is highly robust and is able to process complex English statements. The NLP system is used to lexically and syntactically process the English text and then perform semantic analysis to identify basic SBVR elements. The core system processes a text into three main processing stages:

1. Lexical Processing: The lexical processor comprises for sub-modules: a tokenizer, a sentence splitter, POS tagger, and a morphological analyzer. The input to lexical analyzer is a plain text file containing English description of the target SBVR business rule. Basic NLP techniques such as sentence splitting, tokenization, POS tagging, and morphological analysis are performed and output of this phase is an array list that contains tokens with their associated lexical information.

2. Syntactic Analysis: We have used an enhanced version of a rule-based parser for the syntactic analysis of the input text used in [11]. The text is syntactically analyzed and a parse tree is generated for further semantic analysis.

3. Semantic Analysis: In this semantic analysis phase, role labeling [12] is performed. The desired role labels are actor, co-actor, action, thematic object, and a beneficiary if exists. These roles will assist in identifying different SBVR elements in the next phase and also be used in constructing fact types from the extracted SBVR elements. In semantic analysis phase, after role labeling, the order is identified in which subject, verb, object, and adverb appears in the input English text. The output of the NLP module is an xml file that contains the parsed English text with all the extracted information.Basic SBVR elements e.g. Noun concept, Individual Concept, Object Type, Verb Concepts, etc. are identified from the English input that is preprocessed by the NLP module. Following mapping rules are used to identify the SBVR elements:

- All proper nouns are mapped to the Individual Concepts
- All common nouns appearing in subject part are mapped to the noun concepts.

- All common nouns appearing in object part are mapped to Object Type.
- All action verbs are mapped to Verb Concepts.
- All auxiliary verbs and noun concepts are mapped to the fact types.
- The adjectives and possessive nouns (i.e. ending at 's or coming after 'of') are mapped to the attributes.

All articles and cardinal numbers are mapped to quantification. All these rules are applied to the English text and the output is stored in an array list. Following example highlights the proposition of basic SBVR elements in a typical SBVR rule.

A.2.3 The UML Module

The UML module reads both ECORE and XMI format of a UML class model generated from Eclipse. The UML module extracts all classes, objects, and their respective attributes, operations and associations and finally maps them to SBVR vocabulary. Following section also describes how the SBVR vocabulary is mapped to the SBVR elements generated by NLP module.

1. Generating SBVR Vocabulary: The SBVR vocabulary is generated from the input UML model. All the classes are mapped to noun concepts, attributes of the classes are named as the Individual Concepts, and all the class operations are named as Verb Concepts. The associations and the generalizations are mapped to the binary fact types. Binary fact types are typically composed of two noun concepts and a Verb Concept. All these SBVR elements with their associated types are stored and exported as an array list.

2. Mapping with UML Model: Before translation of English text to SBVR rules, the input English text is mapped with the input UML model to ensure that generated SBVR rules will be semantically related to the target business domain. The mapping is carried out in SBVR elements and SBVR vocabulary. The noun concepts in SBVR rules are mapped to the UML classes. Individual nouns are mapped to the UML objects. Verb Concepts are mapped to methods. Adjectives and possession nouns (with of and 's) are tagged as attributes. A Fact Type is mapped to the associations and generalizations.

A.2.4 The SBVR Module

The SBVR module is based on a rule based parser that contains set of rules to map SBVR elements with SBVR vocabulary and generate complete SBVR rules. In this phase detailed semantic analysis of the English text is performed. Following section describes how the SBVR rules are generated.

*1. Generating SBVR Rule:*SBVR rules are generated from the output of the NLP module. To generate SBVR rules, the first step is to create a fact type. A fact type is created by mapping the noun concepts and Verb Concepts to the fact types available in the SBVR vocabulary array list. Atomic formulization is used to map the input text to a suitable target fact type in SBVR vocabulary. The mapped fact type is used to generate a SBVR rule by applying a set of logical formulations. As For the different types of syntactic structures used in English language, respective types of logical formulations have been defined. Following are the details that how we have incorporated these logical formulations to map English language text into SBVR rule.

*2. Applying SBVR Notation:*The last step in SBVR rule generation is to apply a SBVR notation. RuleGenerator supports both SBVR notations: SBVR Structured English and RuleSpeak. To apply Structured English the noun concepts are underlined e.g. person; the *Verb Concepts* are italicized e.g. *can have*; the **keywords** are bolded i.e. SBVR keywords e.g. **each**, **at least**, **at most**, **obligatory**, etc; the Individual Concepts are double underlined e.g. black car.

It is obligatory that a person's age*should be* **at least** 18 years.

The SBVR produces a SBVR rule in the form of text string that is further formatted using the SBVR notation i.e. Structured English. The output SBVR module is saved and exported in two separate files: an xml file contains the SBVR vocabulary; a text file contains the formatted SBVR rule. The presented approach not only assists the business rule analysts and architects by generating precise SBVR rules from NL specification in a simple and quick manner. As a next step, we are hoping to investigate usability aspects of the tool directly via empirical methods involving teams of developers.

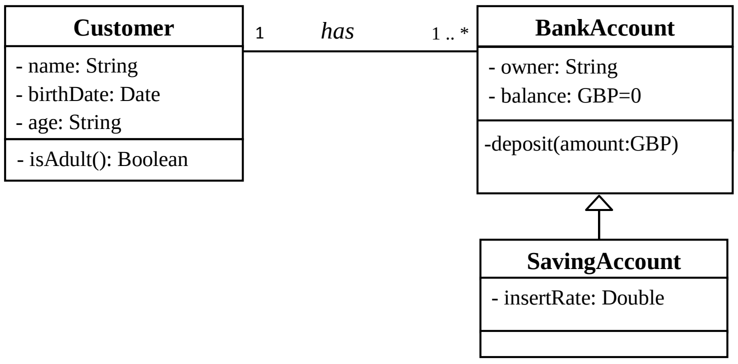
A. 3 Transformation Rules for Translating Business Rules to OCL Constraint

7thEuropean Conference on Modelling Foundations and Applications (ECMFA 2011), Birmingham, UK, June 2011, pp:132-143

This paper was published in ECMFA 2011, held in Birmingham, UK. This paper was focusing on the set of transformation rules used to transform a SBVR rule to an OCL constraint [Bajwa, 2011b].SBVR to OCL transformation is performed in two phases. In first phase, the SBVR constraints specification is mapped to the target UML model and in second phase the SBVR information is mapped to OCL constraints using a set of transformation rules. Detailed description of both phases is given here:

A.3.1 Mapping SBVR Rules to UML Model

In this phase, the SBVR rules are mapped to UML models for semantic verification before the SBVR rules are mapped to OCL constraints. Semantic verification is essential to validate that the target OCL constraints will be consistent with the target UML model. To illustrate the process of mapping SBVR rules to the UML model we have taken an example shown in Figure A.3.1



FigureA.3.1.A UML class model

The mapping process starts with the syntax analysis of SBVR rules to extract various elements of the SBVR rule i.e. noun concepts, Verb Concepts, fact types, etc. Following section describes the process of mapping classes and their respective associations with a common SBVR rule.

*1. Mapping Classes:*The general noun concepts in SBVR rules represent the UML classes. Verb Concepts specify methods of a class. Adjectives are tagged as attributes. For example, in a SBVR rule "**It is obligatory that each** customer*canhave* **at least one**bank account **only if** customer*is* **18** years old.", both noun concepts 'customer' and 'bank account' are matched to all classes in the UML class model shown in the Figure A.3.1 and the noun concepts are replaced with the names of the classes, if matched.

*2. Mapping Class Associations:*Associations in a UML class model express relationship of two entities in a particular scenario. A UML class model may consist of different types of associations, e.g., packages, associations, generalizations, and instances. Typically, these associations are involved in defining the context of an OCL constraint, so it is pertinent to map these associations in the target SBVR specification of business rules.

3. Mapping Packages: A package in a UML class model organizes the model's classifiers into namespaces. In SBVR, there is no specific representation of a package. Hence user has to manually specify the package name for a set of classes. The package names are also defined in the OCL constraints, so the package information is also mapped to the SBVR rules.

4. Mapping Associations: Associations in a UML model specify relationships between two classes. Simple associations can be unidirectional, bidirectional, and reflexive. Unidirectional associations in UML are mapped with unary (based on one noun concept) fact types in SBVR and the bidirectional associations in UML are mapped with binary (based on two noun concepts) fact types in SBVR. Direction of the association is determined by the position (subject or object) of the noun concepts and Object Types in SBVR.

Mapping Generalizations: Generalization/inheritance,in two classes,specifies that one class inherits the functionalities of the other. In SBVR the relationship of general noun concept (super class in UML) and individual noun concept (sub class in UML) is used to identify the inheritance feature. If a class A inherits the class B then the class B will also be the part of OCL context of class A.

*Mapping Instances:*The instances of the classes can also appear in a UML class model. The Individual Concepts in SBVR are mapped to the instances (objects). The defined instances also become part of the OCL contexts and OCL constraints. So, the instances are also mapped in SBVR rules.

In SBVR to UML mapping, the classes that do not map to the given UML class model are ignored.

A.3.2 Mapping SBVR Rules Into OCL Constraints

We present an automated approach for the translation of the SBVR specification into the OCL constraints. Our approach not only softens the process of creating the OCL syntax but also verifies the formal semantics of the OCL expressions with respect to the target UML class model. As OCL is a side-effect free language, hence it is important that each OCL expression should be semantically verified to the target UML model. A prototype tool "SBVR2OCL" is also presented that performs the target transformation. The SiTra library based model transformation framework is used for SBVR to OCL transformation using a set of mapping rules that map .

Mapping of SBVR rules to OCL code is carried out by creating different fragments of OCL expression and then concatenating these fragments to compile a complete OCL expression. Typically, OCL expression can be of two types: OCL invariant and OCL query operation. In this paper, we will present only the creation of OCL invariants and the creation of OCL query operation is part of the future work.

A. 4 SBVR vs OCL: A Comparative Analysis of Standards

14thIEEE International Multi-topic Conference, Dec 2011, Karachi, Pakistan, pp:261-266

In 2011, another paper by the researcher was published in 14th IEEE INMIC, held in Pakistan. This paper presented the key findings during a study of SBVR and OCL standards. A comparison of both standards is also presented in this paper to highlight various similarities and differences in both standards [Bajwa, 2011c]. This study helped the researcher in SBVR to OCL transformation.The part of the comparison related to SBVR to OCL transformation is summarized below:

A.4.1 Syntactical Features

*1. Vocabulary vs Classifiers:*SBVR vocabulary can be of two types: keywords and user defined elements. On the other hand, similar to SBVR vocabularies, OCL expressions can refer to *Classifiers*, e.g., types, classes, interfaces, associations (acting as types), and data types. Common keywords in OCL are`context`, `inv`, `pre`, `post`, etc.

*2. Noun Concept vs Context:*In SBVR metamodel, a Noun Concept can be an Object Type or an Individual Concept. Typically common nouns in English are classified as Object Types and proper nouns are classified as Individual Concepts.In an OCL expression, *Context* is typically represented using a UML class. SBVR Object Type and Individual Concept can be equivalent to a context in an OCL expression.

3. Verb Concepts vs Classifier AnyType: In SBVR, the Verb Concepts (action verbs) typically represent operations performed by/for a business entity. Action verbs in English can be matched to the method and operation names without side-effect in OCL. The Verb Concepts (action verbs) in SBVR metamodel can be equivalent to classifier *AnyType* in OCL metamodel. Similarly, OCL attributes can be equivalent to SBVR's *Characteristics*.

4. Fact Types vs Associations: Associations' ends are commonly used in OCL types. Similarly, in SBVR associations are supports by different types of Fact Types, e.g., associations in SBVR are represented using Associative Fact Types, aggregations are represented using the Categorization Fact Types, and generalizations are represented using the Partitive Fact Types.Similarly, OCL's association multiplicity can be equal to SBVR's quantification such as universal quantification and non-universal quantification.

*5. Projections vs Collections:*A set of Projections are defined in SBVR to handle one or more than one variables. Similarly, OCL introduces Collections to provide support for managing multiple variables. The SBVR's Set Projection is equivalent to OCL's Set Collection and SBVR's Bag Projection is equivalent to OCL's Bag Collection. There are some types of collection such as *Sequence* and*Closed Projection*that are not supported in OCL.

*6. Structural Rule vs Invariant:*The SBVR structural rules represent the structure of a business models and their underlying entities. Similar to SBVR structural rules, invariants are used in OCL to represent a structural constraint.

*7. Behavioural Rule vs Pre/Post Condition:*The behavioural rules govern the behaviour of business activities and operations. Akin to behavioural rules in SBVR, OCL's pre/post conditions are particularly specified to handle behaviour of respective methods of classes and objects.

A. 4.2 Principal Features

The principal features of SBVR & OCL are discussed below:

*1. Conceptual Modeling:*The primary focus of bothlanguages (SBVR and OCL) is same i.e. conceptual modeling just their application domains are different such as SBVR is primarily used for business modeling (in combination with BPMN/BPEL), while OCL is used for software modeling (in combination with UML) and is employed for large scale object oriented models.

*2. Declarative Languages:*SBVR and OCL are both declarative language. SBVR rules should be expressed declaratively in natural-language sentences for the business audience. Similarly, OCL support declaration of OCL constraints used for software models.

*3. Requirement Engineering:*SBVR is typically used capture software/business requirements in natural languages (such as English). Contrary to SBVR, OCL is employed at later stages of software development such as in graphical modelling (UML / SysML / BPMN). Here, OCL's duty is to ensure precise modeling and representation of non-functional requirements.

*4. Side-Effect Free:*Both SBVR and OCL are side-effect free languages. SBVR based rules are side-effect free as all SBVR rules are distinct from any enforcement defined for it. Similarly, OCL is a pure expression language and OCL constraints are side-effect free. Hence, the side-effect free OCL expression cannot change anything in the model and the state of the system.

*5. Well-Formed Expression:*The SBVR business rules should be expressed in such a way that they can be validated for correctness by business people. Business rules should be expressed in such a way that they can be verified against each other for consistency. Similarly, OCL expressions are strictly typed. All the OCL constraints are type-checked and syntactically parsed to check that they are well-formed expressions.

A.4.3 Technical Features

A set of technical features of both SBVR and OCL are compared in this section.

1. SBVR is based on Formal Logics: The formal semantics of SBVR is based on typed predicate logic, arithmetic, set and bag comprehension with some additional basic results from modal logic. Here, the logic is essentially classical logic, so mapping to various logic-based languages is simple. Similarly, OCL also has its roots in mathematical logic. OCL is based on set theory and predicate logic and has a formal mathematical semantics.

*2. Formal Semantics:*A set of logical formulations have been defined in SBVR 1.0 document to provide a foundation for formal semantics. Typically, a business glossary or an enterprise vocabulary based information models are used by the business stakeholders for formal semantics. More formal semantics can be added through business facts and business rules. Similarly, OCL constraints are also semantically formal as OCL formal semantics are described using UML. The semantics of OCL expressions are consistent to a UML class model.

3. Two-value Logic vs Three-value Logic: SBVR's underlying logic is isomorphic (standard truth-functional logic) rather than epistemic logic. The truth functional logic is two-valued, with negated existential formulae being used to avoid the use of null values.Contrary to SBVR, OCL is based on a three-valued logic. OCL's Boolean expression can result in true, false or undefined. Here, the three-valued logic can result in unexpected results.

4. Inherent Extensibility: An extended SBVR vocabulary is created by including the SBVR vocabulary into another business vocabulary that has other designations. The SBVR Vocabularies given by this specification are based on the English language, but can be used to define vocabularies in any language.Similarly, OCL inherits UML vocabulary (classes, associations, methods, etc) to complete basic OCL expressions.

The comparison of SBVR with OCL (together with its commercially-available tool support) in terms of syntactical, principal and technical features has helped to explore SBVR and OCL's commonalities and discords. The comparison shows a remarkable similarity between the two, such as both are based on formal reasoning. The identified commonalities can lead to a transformation from one standard to other.

A. 5 Resolving Syntactic Ambiguities in NL Specification of Constraints using UML Class Model

13th International Conference on Computational Linguistics and Intelligent Text Processing (CICLing 2012), Delhi, India, March 2012, pp:178-187

A paper was published in CICLing 2012 (held in India) to present a set of identified syntactic ambiguities in NL specification of constraints and an approach to resolve such syntactic ambiguities [Bajwa, 2012a].To address lexical and attachment ambiguities a novel approach is presented in this paper. We have identified that the both ambiguities are due to the absence of the context and by suing the context of the English text the correct interpretation of the ambiguous words and phrases is possible. In NL2OCL project, to translate NL specification of constraints to OCL constraints, two inputs are required: English specification of a constraint and a UML class model. We propose the use of the information (such as classes, methods, attributes, associations, etc.) available in the input UML class model for correct syntactic analysis.The used approach for addressing the both types of syntactic ambiguities is explained below:

A.5.1 Solution for Resolving Attachment Ambiguity

The attachment ambiguity can be resolved using the context. For generating correct dependencies of input English sentences, we again use the information on hand in the input UML class model. As, attachment ambiguity is due to the ambiguous role of noun with a preposition in a sentence. To resolve attachment ambiguity, three (can be four or more) nouns are mapped to the class names in the input UML class model. Once the three classes are identified, the associations in those three classes are analyzed. With the help of the associations in the candidate classes the relationships in nouns are correctly identified. For example, the case of attachment ambiguity shown in Figure A.5.1involves three nouns ‘pay’, ‘employees’, and ‘bonus’. All these three nouns are mapped to three classes (such as ‘Employee’, ‘Pay’, and ‘Bonus’) in the UML class model given in FigureA.5.2. After this mapping, the associations in all three classes are analyzed. The Stanford parser wrongly identifies that noun ‘bonus’ is attached to the noun ‘employees’. However, the UML class model shows that there is no relationship in classes ‘Bonus’ and ‘Employee’. While, there is a relationship in class ‘Pay’ and class ‘Bonus’. By using this information, we can correct the wrong dependencies.

English: The pay is given to all employees with bonus.

Typed Dependency (Collapsed) :
```
det(pay-2, The-1)
nsubjpass(given-4, pay-2)
auxpass(given-4, is-3)
det(employees-7, all-6)
prep_to(given-4, employees-7)
prep_with(employees-7, bonus-9)
```

Figure A.5.1. Incorrect typed dependencies (collapsed) generated by the Standord Parser

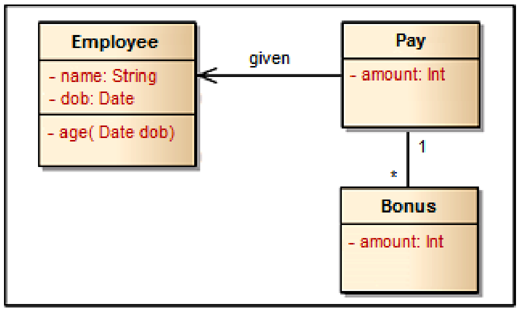


Figure A.5.2.A UML class model

We have generalized the used approach so that all variations of the discussed type of attachment ambiguity can be handled. For this purpose, the analysis of the relationships in classes of a UML class model such as associations (directed and un-directed), aggregations and generalizations can play a key role.

A.5.2 Solution for Resolving Lexical Ambiguity

As, we have explained earlier that the absence of the context is the major reason of ambiguity. For correct POS tagging of all English sentences especially the case of lexical ambiguity (homonymy), we aim to use the available information in the target UML class model such as class names, attribute names, method names, associations, etc. In syntactic analysis, once we get the output of the Stanford POS tagger, we map all the words and their tags with the UML class model and confirm that all POS tags are correctly identified.

The process of mapping of POS tagged text to the UML class model is very simple. The POS tags of all the words are mapped to the elements of the UML class model. A set of mappings were defined for this purpose as shown in Table A.5.1. If the token matches to an operation-name or a relationship name then it is a verb or if the ambiguous token matches to a class-name or attribute-name then it is classified as a common noun or proper noun.

Table A.5.1. Mapping of English elements to UML class model elements

UML class model elements		**English language elements**
Class names	→	Common Nouns
Object names	→	Proper Nouns
Attribute names	→	Generative Nouns, Adjectives
Method names	→	Action Verbs
A ssociations	→	Action Verbs

By using the information shown in Table I, we can correctly POS tag the example of homonymy discussed in Section 2.2. In Figure A.5.3, it is shown that 'books' is an association in two classes 'Customer' and 'Item'. By using such information, it is identified that 'books' cannot be a noun in the context of UML class model. However 'books' can be a verb and the correct POS tag of token 'books' should be 'VBZ' as the token 'books' is it is postfix of MD 'can'.

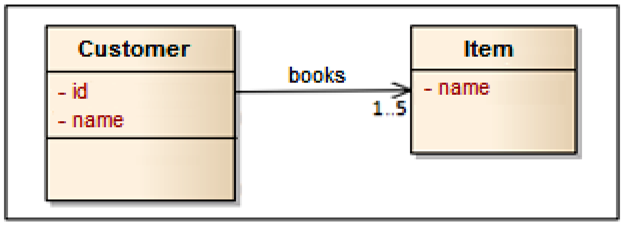


Fig A.5.3.A UML Class model

After POS tag correction, the parse tree and dependencies are also corrected (See Figure A.5.4).

English:	A customer books two items.
Tagging:	[A/DT] [customer /NN] [books/NN] [two/CD] [items/NNS] [./.]

Figure. A.5.4. Incorrect Parts-of-Speech tags, generated by the Stanford POS Tagger

The primary objective of the paper was to address the challenge of resolving various cases of syntactic ambiguity such as attachment ambiguity and homonymy. By resolving the said cases of syntactic ambiguity the accuracy of machine processing can be improved. To address this challenge we have presented a NL based automated approach that uses a UML class model as a context of the input English (constraints) and by using the available information in the UML class model (such as classes, methods, associations, etc.) we can resolve attachment ambiguity and homonymy. The results show a significant improvement in the accuracy of the Stanford POS tagger and the Stanford parser. By improving the accuracy of the Stanford POS tagger and the Stanford parser, the accuracy of English to OCL translation is also improved to 92.85% that was earlier 84.7%.

A. 6 Semantic Analysis of Software Constraints

The 25th International FLAIRS Conference, Florida, USA, May 2012, pp:8-13

The presented approach was using metadata of UML Class Model to resolve identified syntactic ambiguities. Two more papers were published in the 25th edition of FLAIRS, held in USA: the focus of one paper was the semantic Analysis of Software Constraints [Bajwa, 2012b]. For translating English specification of constraints to OCL constraints, the NL2OCL approach was used. In the NL2OCL approach, two inputs are given: a txt file containing English specification of a constraint, and a UML class model in EMF (Eclipse Modeling Framework) ECORE format. First English specification is syntactically and semantically analyzed to extract OCL elements and then finally an OCL expression is generated.

The Royal & Loyal case study has also been solved by Wahler [2008] in his PhD thesis. We aim to compare the results of our approach to Pattern based approach as Wahler's approach is the only work that can generate OCL constraints from a natural language. There are 26 English constraints in the Royal & Loyal case study. Wahler solved 18 English constraints into OCL out of 26 using his (pattern-based) approach. In comparison to Wahler's pattern based approach, our NL-based approach has successfully translated 25 constraints to OCL.

A. 7 Addressing Semantic Ambiguities in Natural Constraints

The 25th International FLAIRS Conference, Florida, USA, May 2012, pp:262-267

This paper highlights the identified set of semantic ambiguities in NL constraints. An approach is also presented in the paper to address the identified set of semantic ambiguities in NL constraints [Bajwa, 2012c]. The present approach helps in improving semantic role labeling and quantifier scope resolution in terms of accuracy. A set of ambiguities in shallow and deep semantic parsing are identified that are due to the absence of the context. However, these semantic ambiguities can be resolved by using the context of the English text. In the NL2OCL project, to translate English specification of constraints to OCL constraints, two inputs are required: English specification of a constraint and a UML class model. We propose the use of the information (such as classes, methods, associations, multiplicity, etc.) available in the input UML class model to handle semantic ambiguities.The used approach for addressing the both types of semantic ambiguities is explained in remaining part of the section.

A.7.1 Addressing Semantic Ambiguities

It is a fact that the semantic ambiguities in English constraints are due to absence of the context of the constraint. As, a UML model is a typical context of the OCL constraints, we use the UML class model shown in Figure A.7.2 to address the identified semantic ambiguities.

English: A customer cannot place more than two orders.

Figure A.7.1. Input English Constraint

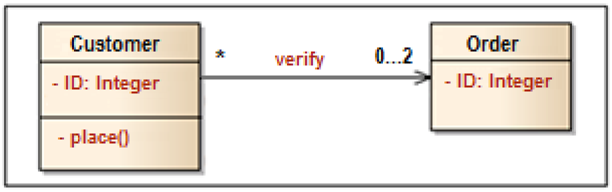


Figure A.7.2. A UML class model

To identify correct semantic roles, we worked out a mapping in English constraints, UML class model and SBVR based semantic roles.

English Elements	UML Elements	SBVR based Semantic Roles
Common Nouns	Classes	Object Type
Proper Nouns	Objects	Individual Concept
Generative Noun, Adjective	Attributes	Characteristic
Verbs	Methods	Verb Concepts
	Associations	Fact Type

Table A.7.1. Identifying Semantic Roles

The first case of semantic ambiguity was related to assignment of semantic roles to a verb in English constraint. It is shown in Figure A.7.2 that 'Customer' and 'Order' are two classes while 'place' is name of a method. Due to the fact that methods in a UML class model are action performed by the class, we classify verb 'place' as a Verb Concept (see Table A.7.1). If the verb 'place' is an association among classes 'Customer' and 'Order', it is classified as a Fact Type.

We can identify correct semantic role by mapping information to the UML class model by checking that verb is an operation or an association. If a verb is operation it is mapped to 'Verb Concept' else it is mapped to a 'Fact Type'. Moreover, for the sake of confirmation we also map the common nouns such as 'Customer' and 'Order' to the classes in the UML class model. After identifying the correct semantic roles, following output was generated (see Table A.7.2) for example "A customer cannot place more than two orders."

English Elements	Assigned Semantic Roles
A	-
Customer	Object Type
Cannot	-
Place	Verb Concept
more than two	-
Orders	Object Type

TableA.7.2. Semantic roles assigned to input English sentence

The second case of semantic ambiguity was related to the order of predicate arguments extracted for a predicate. To resolve this type of ambiguities the information of English constraint given in Figure A.7.1 was again mapped to the information of the UML class model shown in Figure A.7.2. After mapping we found that 'Customer' and 'Order' are two classes and there is a directed association between these two classes. The directed association shows that the 'Customer' is an agent or an actor and 'Order' is a patient or a thematic object. In the light of this information it is simple to identify that the predicate arguments should be like place(customer, order). Another benefit of such mapping is that if English sentence in passive voice the same predicate will be generated, e.g., place(customer, order).

A.7.2 Addressing Semantic Ambiguity in Quantification

To address the semantic ambiguity, first we identified the candidate quantifier operators in English constraints. Then the identified quantifiers are mapped to the multiplicities of classes in a UML class model to confirm the quantifications. We have figured out following four types of the quantifications in English constraints.Output of quantification handling for the example discussed in the Figure 4 is shown in the Figure 5.

English Elements	Assigned Semantic Roles
A	Universal Quantification
customer	Object Type
cannot	-
place	Verb Concept
more than two	At-least n Quantification
orders	Object Type

Table A.7.3. Semantic roles assigned to input English sentence

After shallow and deep semantic parsing, a final semantic interpretation is generated. A simple interpreter was written that uses the extracted semantic information and assigns an interpretation to a piece of text by placing its contents in a pattern known independently of the text. Figure 6 shows an example of the semantic interpretation we have used in the NL2OCL approach:

English: A customer cannot place more than two orders.

Semantic Interpretation:

```
        ( place
  (object_type = (∀ X ~ (customer ? X)))
                  (object_type = §Y ~ (order ? Y))))
```

Figure A.7.3. Semantic Interpretation of English constraint

Much work has been done in the field of natural language ambiguity identification and resolution. Some of the researchers have presented approaches to identify the various types of ambiguities in a natural language text especially the natural language software requirements. Hence, the resolution of semantic ambiguities in natural language specifications of software requirements and software constraints become more critical

A. 8 Translating Natural Language Constraints to OCL

JKSU - Computer and Information Sciences, June 2012, 24(2): Elsevier

Another paper presenting the results of Royal and Loyal modal case study was presented in Journal of King Saud University - Computer and Information Sciences [Bajwa, 2012d]. This paper highlights that the researcher's NL-based approach is more accurate than the pattern based approach [Wahler, 2008]. The Royal and Loyal case study is presented in this paper. The average F-value of results is calculated 84.15% that is encouraging for initial experiments. The results show that that other language processing technology such as information extraction systems, and machine translation systems, have found commercial applications with precision and recall figure well below this level. Thus, the results of this initial performance evaluation are very encouraging and support both the NL2OCL approach and the potential of this technology in general.

A. 9 On a Chain of Transformations for Generating Alloy from NL Constraints

7th IEEE International Conference on Digital Information Management (ICDIM 2012), Macau, August 2012, pp:93-98

An extension of this work is accepted in IEEE ICDIM 2012 (held in Macau), that focuses on generation of Alloy code from NL constraints [Bajwa, 2011e]. This work is also used in qualitative analysis of our approach presented in Chapter 7, Section 7.4 as the researcher generates Alloy of OCL (generated by our NL approach) and if Alloy is generated correctly, it means that OCL is also correct. The details of this evaluation are given in Section 7.4 of this thesis. The contribution of this paper is generation of Alloy from NL and then using this Alloy for analysis of the models. The analysis of the model can be carried out from within the NL2OCL, using the UML2Alloy and the Alloy Analyzer APIs. More specifically, the UML class diagram and the automatically generated OCL constraints were automatically transformed to Alloy using the API of the UML2Alloy.

Once the Alloy model is automatically generated, we can analyse it with the help of the Alloy Analyzer API. This means that the Alloy Analyzer will attempt to find instances, which conform to the model and its constraints using combinations of up to four *File* and *Directory* instances. After producing a number of acceptable instances, the Alloy Analyzer returned the instance depicted in Figure 7. This was automatically transformed from the Alloy Analyzer analysis notation to UML Object Diagrams by UML2Alloy. The instance shows a directory (*Directory0*), which is not part of the directories hierarchy. Moreover we see that *Directory1* is indirectly a parent of itself (through Directory2).

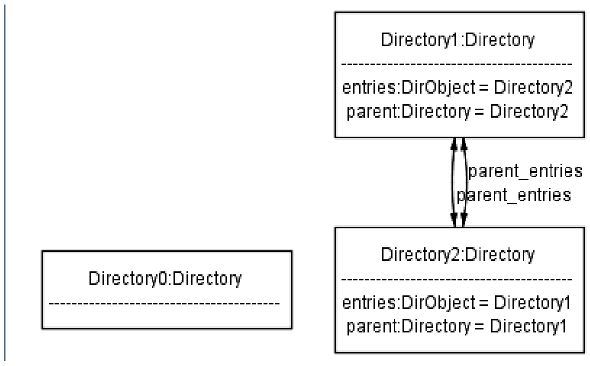

Directory1:Directory
entries:DirObject = Directory2
parent:Directory = Directory2
parent_entries
parent_entries
Directory0:Directory
Directory2:Directory
entries:DirObject = Directory1
parent:Directory = Directory1

Figure A.9.1. Instance provided by the Alloy Analyzer

This is clearly an instance that is not desirable. Inspecting our initial model, we can assume that *Constraint 2* needs to be augmented to express that a directory may not be directly **or indirectly** a parent of itself (i.e. we need to express that the *parent* association is acyclic). In order to do that we would need to express transitive closure using natural language in the NL2Alloy tool. However, we cannot do that since the OCL itself is missing a transitive closure operation. Instead of transitive closure the UML standard uses recursion to express transitive closure. More precisely, in recursion is used to express the *allParents()* operation to express that a Generalization relation between UML Classes is acyclic and directed.

This research paper presents a framework for dynamic generation of the Alloy code from the NL specification provided by the user. Here, the user is supposed to write simple and grammatically correct English. The designed system can find out the required information to generate a SBVR representation and then transform to a complete SBVR rule, after mapping with the input UML model. The SBVR rules are transformed to OCL expressions and finally translated to Alloy code.

APPENDIXB
TESTING RULES OF THE RULE SET

In this Appendix, the researcher demonstrates the manually crated examples to test rules in the used rule set in the NL2OCL tool.Following are the statistics of the number of the methods, transformation rules, classes and others test by the researcher.

Name of Constructs to be tested	Number of Constructs	Number of Tested Constructs
Transformation Rules	20	20
Classes	61	61
Method	83	83

Table B.1 –Statistics of the tested constructs

The examples are explained below:

B. 1 Test At Most Quantification

Following is the example demonstrating that the use of 'less than' quantification and it is mapped to '`at most`' in SBVR and in OCL, it is mapped to `age < 30`.

English:	A person's age should be less than 30 years.
SBVR:	It is obligatory that aperson's***age****shouldbe*at most30years.
OCL:	**package**: ecoreuml **context** Person **inv**: self.age <30 end**package**

B. 2 Test At Most Quantification with Negation

It is demonstrated in the following example that the use of ‘not less than’ quantification and it is mapped to ‘`at least or exactly`’ in SBVR and it is mapped to `age >= 30`in OCL.

English: A person's age should not be less than 30 years.

SBVR: It is obligatory that aperson'sageshouldbeat least or exactly30years.

OCL:
```
package: ecoreuml
context Person
inv: self.age >=30
endpackage
```

B. 3 Test At Least Quantification

The following example demonstrates the use of ‘more than’ quantification in NL constraint and it is mapped to ‘`at least`’ in SBVR and it is mapped to `age > 30`in OCL.

English: A person's age must be more than to 30.

SBVR: It is obligatory that aperson'sagemustbeat least30.

OCL:
```
package: ecoreuml
context Person
inv: self.age >30
endpackage
```

B. 4 Test At Least Quantification with Negation

It is demonstrated in the following example that the use of ‘not more than’ quantification in NL constraint and it is mapped to ‘`at most or exactly`’ in SBVR and in OCL, it is mapped to `age <= 30`.

English: A person's age must not be more than to 30 years.

SBVR: It is obligatory that aperson'sagemustbeat most or exactly30years.

OCL:
```
package: ecoreuml
context Person
inv: self.age <=30
endpackage
```

B. 5 Test Exactly Quantification

The following example demonstrates the use of ‘equal to’ quantification in NL constraint and it is mapped to ‘`exactly`’ quantification in SBVR and it is mapped to `age = 30`in OCL.

English: A person's age must be equal to 30.

SBVR: It is obligatory that aperson'sagemustbeexactly30.

OCL:
```
package: ecoreuml
context Person
inv: self.age =30
endpackage
```

There is another possible way of representing ‘equal to’ quantification in NL constraint. If a tool has a positive sense and no particular quantifier is mentioned, our tool considers it ‘equal to’ quantification.

English: A person's name can be Ahmad.

SBVR: It is obligatory that aperson's***name****can be*Ahmad.

OCL:
```
package: ecoreuml
context Person
inv: self.name =Ahmad
endpackage
```

B. 6 Test Exactly Quantification with Negation

It is demonstrated in the following example that the use of ‘not equal to’ quantification in NL constraint and it is mapped to ‘`not exactly`’ in SBVR and in OCL, it is mapped to `age <> 30`.

English: A person’s age must not be equal to 30.

SBVR: It is obligatory that aperson's***age****must* not*be*exactly30.

OCL:
```
package: ecoreuml
context Person
inv: self.age <>30
endpackage
```

There is another possible way of representing ‘not equal to’ quantification in NL constraint. If a tool has a negative sense and no particular quantifier is mentioned, our tool considers it ‘not equal to’ quantification.

English: A person's name cannot be Ahmad.

SBVR: It is obligatory that aperson's***name****cannot be*Ahmad.

OCL:
```
package: ecoreuml
context Person
inv: self.name<>Ahmad
endpackage
```

B. 7 Test Multiple Quantification

There is a possibility that user can represent two quantification in single NL constraints, e.g., “less than and more than”. Example of such case is given below.

English: A person's salary should be less than 8000 and more than 4000.

SBVR: It is obligatory that aperson's***salary****should be*at most8000andat least4000.

OCL:

```
package: ecoreuml
context Person
inv: self.salary< 8000 and self.salary > 4000
endpackage
```

B. 8 Test Self

Following is the test case that involves self in an invariant.

English: A male person's gendershould be male.

SBVR: It is obligatory thatperson's*gendershould be*male.

OCL:

```
package: ecoreuml
contextCustomer
inv: self.gender = male
Endpackage
```

B. 9 Test size()

The following example demonstrates the translation of `size()`.

English: A person must participate in at least one loyalty program.

SBVR: It is obligatory that a person*mustparticipate* in at least one loyaltyprogram.

OCL:

```
package: ecoreuml
context Person
inv: self.program ->size() >= 1
Endpackage
```

B. 10 Test includes()

In the following example, the translation of `includes()` is tested.

English: A member must have a membership card.

SBVR: It is obligatory that a member*musthave*at least one membershipcard.

OCL:

```
package: ecoreuml
contextMember
inv: self.cards -> includes(membership.card)
Endpackage
```

B. 11 Test excludes()

The translation of `excludes()` is tested in the following example.

English: A member must not have a membership card.

SBVR:	It is obligatory that a membermust nothaveamembershipcard.

OCL:	`package: ecoreuml`
	`contextMember`
	`inv: self.cards -> excludes(membership.card)`
	`Endpackage`

B. 12 Test isEmpty()

The following example illustrates the translation of `isEmpty()`.

English:	A member must not have saving account.
SBVR:	It is obligatory that a membermust nothavesaving_account.

OCL:	`package: ecoreuml`
	`contextMember`
	`inv: self.saving_Account ->isEmpty()`
	`Endpackage`

B. 13 Test includes()

In the following example, the translation of `includes()` is tested.

English:	The name of first account of a member should be Current.
SBVR:	It is obligatory that the first account of a membershould beCurrent.

OCL:	`package: ecoreuml`
	`contextMember`
	`inv: self.account -> first().name = Current`
	`Endpackage`

B. 14 Test exist()

The translation of `exist()` is tested in the following example.

English:	There must be at least one account name Saving for a member.
SBVR:	It is obligatory thatthere must beat least one accountname Saving for a member.

OCL:	`package: ecoreuml`
	`contextMember`
	`inv: self.account ->exist(name=Saving)`
	`Endpackage`

B. 15 Test isUnique()

In the following example, `isUnique()` is tested.

English: The account number of a member must be unique.

SBVR: It is obligatory that the account***number*** of a member *must be* unique.

OCL:
```
package: ecoreuml
contextMember
inv: self.account -> isUnique(acc|acc.number)
Endpackage
```

B. 16 Test forAll()

The following example illustrates the translation of `forAll()`.

English: The account number of all members must be different.

SBVR: It is obligatory that the account***number*** of a member *must be* unique.

OCL:
```
package: ecoreuml
contextMember
inv: self.account ->forAll(m1,m2 | m1.number<> m2.number)
Endpackage
```

B. 17 Test select()

The translation of `select()` is tested in the following example.

English: There must be one card with at least 100 points.

SBVR: It is obligatory that the account***number*** of a member *must be* unique.

OCL:
```
package: ecoreuml
contextMember
inv: self.card -> select(point = 100)->size()=1
Endpackage
```

B. 18 Test Implication

Following is the test case that involves implication in an invariant.

English: A male person's title should be 'Mr.'.

SBVR: It is obligatory that***male***person's***title****Should be*'Mr.'

OCL:
```
package: ecoreuml
contextperson
inv: self.isMaleimplies self.title='Mr.'
Endpackage
```

B. 19 Test If

Following is the test case that involves if expression in an invariant.

English:	If person's gender is male then person's title should be 'Mr.'.
SBVR:	If person's ***gender*** is male then it is obligatory thatperson's***title****Should be*'Mr.'

OCL:	`package: ecoreuml`
	`contextperson`
	`inv: ifself.gender = male then`
	`self.title='Mr.'`
	`endif`
	`Endpackage`

B. 20 Test If-Else

In the following test case, an invariant with if-else expression is tested.

English:	If person's gender is male then person's title should be 'Mr.' else person's title should be 'Ms.'.
SBVR:	If person's ***gender*** is male then it is obligatory thatperson's***title****Should be*'Mr.' else person's***title****Should be*'Ms.'

OCL:	`package: ecoreuml`
	`contextperson`
	`inv: ifself.gender = male then`
	`    self.title='Mr.'`
	`     else`
	`        self.title= 'Ms.'`
	`endif`
	`Endpackage`