On A Chain of Transformations for Generating Alloy from NL Constraints

Imran Sarwar Bajwa, Behzad Bordbar, Kyriakos Anastasakis, Mark Lee
School of Computer Science
University of Birmingham, UK
i.s.bajwa@cs.bham.ac.uk, b.bordbar@cs.bham.ac.uk, k.Anastasakis@cs.bham.ac.uk, m.g.lee@cs.bham.ac.uk

Abstract—***Multi-Paradigm Modelling*** **uses models from multiple domains to leverage the tools, techniques and expertise provided by each of the individual domains. Recent advances in model transformation technology allow automated production of one model from another to improve the application of multi-paradigm techniques. Systems development starts with the requirements gathering phase, which usually comprises of a textual description of the system requirements provided in *Natural Language* (*NL*). It is therefore evident that there is a clear scope for incorporating NL Processing techniques in Multi-Paradigm Modeling. However, using NLP methods pushes the boundaries of Multi-Paradigm Modeling to an extreme; indeed NLs are inherently ambiguous and open to interpretation. In this paper, we propose a novel approach based on standards (such as SBVR) that can cope with syntactic and semantic ambiguities in NL specifications and can map them to formal languages such as Alloy. The tool implementing our approach is currently the only available tool for translating NL specifications to formal languages such as Alloy, etc.**

Keywords- UML, Alloy, SBVR, Natural Language, NL2OCL

I. Introduction

Using more than one model of a system, commonly known as multi-paradigm modelling [1], has received considerable attention in the last decade. Indeed, models of the systems in different domains such as software engineering (UML [2], OCL [3]), formal methods (Alloy [4], B [5]), and business modeling (SBVR [6]) produce different viewpoints which allow benefiting from tools, techniques and expertise provided by each domain. In addition, recent advances in model transformation technology, most notably Model Driven Development [7] (MDD), have allowed production of one model from another automatically for example OCL/UML to Alloy [8], SBVR to OCL [9], SBVR to UML [10], UML/OCL to SBVR [11], OCL to B [5], SBVR to SQL [12], etc. Such automated transformations has made easy and simple to reuse the existing information.

In NL2Alloy project, we aim to automatically generate Alloy from the natural language (NL) specification and use that Alloy for analysis of UML class models. For automated generation of Alloy from NL specification, a sequence of transformations (see Figure 1). First of all NL constraint is syntactically and semantically analysed to generate a logical representation that can be mapped to formal languages such as Alloy. Here, the logical representation is based on the Semantic of Business Vocabulary and Rules (SBVR) [6] standard. Afterwards, SBVR based logical representation is mapped to OCL constraints. Finally, the OCL constraints are mapped to Alloy expressions for a UML class model in OCL2Alloy transformation [8].

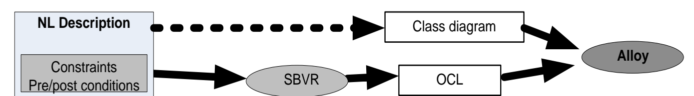

Figure 1: From Natural Language Specification to Alloy

In NL2Alloy transformation, syntactic and semantic analysis of NL specification is the key phase. Any error or mistake generated at this stage propagates in rest of the stages and results in wrong OCL and wrong Alloy. For higher accuracy in syntactic analysis, we have used the Stanford parser to generate the parse tree and the (typed) dependencies and are 84.1% [7] accurate. However, we have identified a few cases the Stanford parser is unable to generate (typed) dependencies. In natural language processing (NLP), such cases are classified as attachment ambiguity (Kiyavitskaya, 2008), where a prepositional phrase or a relative clause in sentence can be lawfully attached to one of the two parts of that sentence [10]. An example of such cases is shown in Figure 2.

English: A directory object is assigned to all files with directory.

Typed Dependency (Collapsed):

```
det(Object-3, A-1)
nn(Object-3, Directory-2)
nsubjpass(assigned-5, Object-3)
auxpass(assigned-5, is-4)
root(ROOT-0, assigned-5)
det(files-8, all-7)
prep_to(assigned-5, files-8)
prep_with(object-8, Directory-10)
```

Figure 2. Typed dependencies generated by the Stanford Parser

In Figure 2, it is shown that the typed dependencies generated by the Stanford parser are wrong such as `prep_with(files-8, Directory-10)`. However, the correct typed dependency for this example should be `prep_with(pay-2, Directory-10)` to represent the

actual meanings of the example i.e. a Directory Object with Directory is assigned to all the files. This problem becomes more critical when we map these (typed) dependencies to SBVR vocabulary and OCL. Wrong dependencies generated by the Stanford parser result in wrong SBVR and wrong OCL, while the wrong OCL is mapped to wrong Alloy or result in incomplete OCL to Alloy transformation.

This paper has three main contributions. Firstly, we present an outline of the NL2Alloy's architecture which decouples the design of the tool so that NL representation can be transformed to Alloy via multiple separate modules. Secondly, a novel approach is used to resolve attachment ambiguity in NL specification of Alloy. By resolving the attachment ambiguity, accurate Alloy code can be generated. Thirdly, we present how the NL2Alloy tool can be helpful in analysis of models.

The rest of the paper is structured as follows. Section 2 describes the preliminary concepts of the research; section 3 highlights main phases of NL to Alloy approach and the working of NL2Alloy approach is explained with the help of a running example in section 4; section 5 describes results and evaluation of the tool. The paper ends with a conclusion section.

II. Preliminaries

The preliminary concepts such as OCL and Alloy are described in this section.

1) Object Constraint Language (OCL)

OCL is a formal language used to annotate a UML model with the constraints [3]. The typical use of OCL is to represent functional requirements using class invariants [3, Section 7.3.3], pre and post conditions [3, Section 7.3.4] on operations and other related expressions on a UML model. OCL supports two types of expressions: constraints and pre/post conditions. A constraint is a restriction on state or behaviour of an entity in a UML model [2]. The OCL constraint defines a Boolean expression. If the constraint results true, the system is in valid state. OCL is a strongly typed formal specification language with precise semantics. All well-formed expressions must conform to the rules of OCL.

2) Alloy

Alloy [4] is a declarative textual modeling language based on first-order relational logic. An Alloy model consists of a number of *signature* declarations, *fields*, *facts* and *predicates*. Each *signature* denotes a set of *atoms*, which are the basic entities in Alloy. Atoms are *indivisible* (they cannot be divided into smaller parts), *immutable* (their properties remain the same over time) and *uninterpreted* (they do not have any inherent properties). Each *field* needs to be declared under a *signature* and represents a relation between two or more *signatures*. Each *field* denotes a set of tuples of atoms. *Facts* are declarative statements in first-order logic that define constrains on the declared *signatures* and *fields*. *Predicates* are in essence parameterized constraints that can be referenced from within other *predicates* or *facts*.

The Alloy language is supported by a tool, the *Alloy Analyzer* which supports fully automated analysis of Alloy models. The tool can produce random instances of a model (*simulation* functionality). It can also check if the model satisfies certain desirable properties. These properties can be expressed in the Alloy language and the tool checks if the properties are satisfied (*assertions* checking functionality). Moreover it provides support to debug over-constrained models by locating the parts of the model that cause the inconsistency (*UnSAT core* functionality)

The Alloy Analyzer works by transforming an Alloy model to a Boolean expression that can be analysed by SAT solvers embedded within the Alloy Analyzer. A user-specified *scope* on the model elements is used to bound the domain. The *scope* is a positive integer number, which limits the number of *atoms* for each *signature* in an instance of the system that is analysed by the solver. If an instance that violates the assertion is found within the scope, the assertion is not valid. However, if no instance is found, the assertion might be invalid in a larger scope. For more details on the notion of *scope*, please refer to [4, Sec. 5].

Alloy is ideal for analysing structural properties of systems [4] and can therefore be considered a natural choice for statically analysing UML Class Diagrams. In particular, existing work has transformed UML and OCL to Alloy manually for analysis. For example, Dennis et al. [35] use Alloy to expose hidden flaws in the UML design of a radiation therapy machine.

3) UML2Alloy

UML2Alloy works by automatically transforming UML Class Diagrams enriched with OCL constraints into an Alloy model. This Alloy model can then be automatically analysed using the Alloy Analyzer. There are clear similarities between UML Class Diagrams and Alloy. From a semantic point of view both Alloy and UML models can be interpreted by sets of tuples [4], [25]. Alloy is based on first-order logic and is well suited for expressing constraints on Object-Oriented models. Similarly, OCL has extensive constructs for expressing constraints as first-order logic formulas. In spite of such similarities, the UML and the Alloy have some fundamental differences [26]. For example, Alloy makes no distinction between sets, scalars and relations, while the UML distinguishes between the three. To bridge some of those semantic differences between UML and Alloy model elements a UML profile for Alloy has been developed [26].

III. NL2Alloy: Sketch of the Solution

To address the above challenges we have integrated our existing tools and developed new modules to create a multi-paradigm NL-based approach that generates Alloy code via UML, SBVR and OCL. The NL to Alloy transformation is performed by using a chain of transformations as NL/UML to SBVR, SBVR to OCL, and OCL to Alloy. NL2Alloy architecture is shown in Figure 3.

NL2Alloy approach takes two input documents: an English text document and a UML model document. Then the tool produces and SBVR representation. This representation is

used by the user to double-check if the correct English is produced. This is a step towards curbing the complexities associated to the ambiguities of the Natural Languages []. This is the only semi-automated part of our approach. The remaining steps of the transformation are fully automated. SBVR is transformed into OCL and then into the Alloys module. These steps are discussed in the following chapters. It is possible to produce the UML diagram via one of the many Class diagram extraction tools. However, this step, which is a minor extension of work remains a task for future.

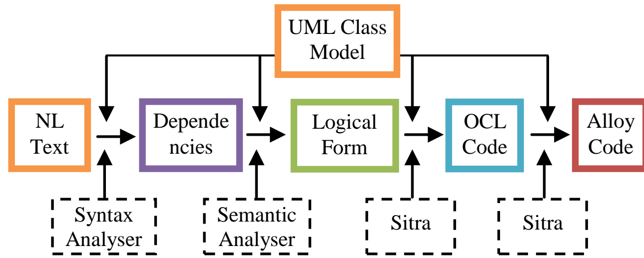

Figure 3. Architecture of NL2Alloy

1) NL to SBVR Transformation

This is the most important phase of NL2Alloy transformation as we generate a semantically unambiguous representation such as SBVR business rules from NL specification of Alloy code. To overcome ambiguity of a natural language, basic natural language processing (NLP) (lexical analysis [27], syntax analysis [35], and semantic analysis [33]) phases are applied to understand the actual meanings of the NL statement and then map NL statement to a SBVR statement. Following sections explains the transformation of NL text to SBVR rules.

a) Lexical Analysis. First phase in analysis of natural language specification of Alloy text is lexical analysis. Following steps are performed to extract lexical information:

i. POS Tagging. In this step, the input English text is tokenized and part-of-speech (POS) tagging is performed using the Stanford POS tagger (Toutanova, 2003).

English: There is exactly one directory that has no parent.

Tags: `[A/DT] [directory/NN] [object/NN] [is/VBZ] [assigned/VBN] [to/TO] [all/DT] [files/NNS] [with/IN] [directory/NN] [./.]`

Figure 4. Part-of-Speech tagged text

ii. Lemmatization. In lemmatization phase, the inflectional endings are removed and the base or dictionary form of a word is extracted, which is known as the lemma. We identify lemma (base form) of the tokens (all nouns and verbs) by removing various suffixes attached to the nouns and verbs e.g. in Figure 4, verb “assigned” is analyzed as “assign+ed”. Similarly, the noun “files” is analyzed as “file+ s”.

b) Syntactic Analysis. We have used the Stanford parser to parse the pre-processed English text. The Stanford parser is 84.1% accurate (Cer, 2010). However, the Stanford parser is not capable of voice-classification. Hence, we have developed a small rule-based module classifies the voice in English sentences. In syntax analysis phase, three steps are performed as below:

i. Generating Syntax Tree. We have used the Stanford parser to generate parse tree and (typed) dependencies (Marneffe, 2006) from NL text. To address the identified cases of attachment ambiguity, discusses in the Section 1, we need the context of the NL statement that is a UML class model is the context of the Alloy code. Therefore, we have used the UML class model shown in Figure 6 to correct dependencies. We have used the given relationships in the UML class model such as the associations (directed and un-directed) to deal with the attachment ambiguity. For example in Figure 6, it is shown that ‘Bonus’ is associated to ‘Pay’ and there is no association in Variable and Memory classes. By using this associations among these classes, we can correct the dependency as `prep_with(files-8, Directory-10)` instead of the `prep_with(object-8, Directory-10)` identified by the Stanford Parser.

Typed Dependency (Collapsed) :

```
det(Object-3, A-1)
nn(Object-3, Directory-2)
nsubjpass(assigned-5, Object-3)
auxpass(assigned-5, is-4)
root(ROOT-0, assigned-5)
det(files-8, all-7)
prep_to(assigned-5, files-8)
prep_with(object-8, Directory-10)
```

Figure 5. Corrected (typed) dependencies

ii. Voice Classification. In Alloy generation, an active voice sentence is treated differently from a passive voice sentence. The Stanford Parser does not classify the voice of English 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. Similarly, 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.

c) Semantic Analysis. In semantic analysis phase, we aim to understand the exact meanings of the input English text; to identify the relationships in various chunks and generate a logical representation. For semantic analysis English constraints, we have to analyze the text in respect of particular context such as UML class model. Our semantic analyzer performs following three steps to identify relations in various syntactic structures:

i. Shallow Semantic Parsing: In shallow semantic parsing, the semantic or thematic roles are typically assigned to each syntactic structure in a English sentence. We use SBVR vocabulary as the target semantic roles due to the fact that the mapping of SBVR vocabulary to OCL is easy and straightforward. We have identified mappings of English text elements to SBVR vocabulary (see Table 1).

Table 1: Mapping class model to English

English Text elements	SBVR Vocabulary
Common Nouns	Object Type
Proper Nouns	Individual Concept
Generative Noun, Adjective	Characteristic
Action Verbs	Verb Concepts
Subject + verb + Object	Fact Type

Following is the procedure used for semantic role labeling of English constraints:

To identify predicates, first of all system identifies the words in the sentence that can be semantic predicates or semantic arguments. In English text, a predicate can be in the form of a simple verb, a phrasal verb or a verbal collocation. Similarly, the predicate arguments can be nouns in subject and object part of a sentence. In English, nouns can have pre-modifiers such as articles (determiners) and can also have post-modifiers such as prepositional phrases, relative (finite and non-finite) clauses, and adjective phrases.

Once the predicates are identified, semantic roles are assigned by using the mappings given in Table 2. Role classification is performed as the syntactic information (part of speech and syntactic dependencies). The output of this phase is shown in Figure 7.

A $_{\textbf{Object_Type}}$[directory object] $_{\textbf{verb_concept}}$[is assigned] to all $_{\textbf{Object_Type}}$ [files] with $_{\textbf{Object_Type}}$ [directory].

Figure 7. Semantic roles assigned to input English sentence

ii. Deep Semantic Analysis. The computational semantics aim at grasping the entire meanings of a natural language sentence, rather than focusing on text portions only. For computational semantics, we need to analyze the deep semantics of the input English text. The deep semantic analysis involves generation of a fine-grained semantic representation from the input text. Various aspects are involved in deep semantics analysis. However, we are interested in quantification resolution (see Figure 8) and quantifier scope resolution:

i. In English constraints, the quantifiers are most commonly used. We not only cover all two traditional types (Universal and Existential) of quantifications in FOL but also we have used two other types: Uniqueness and Solution quantification..

ii. Besides, the quantification resolution, we also need to resolve the scope of quantifiers in input English text.. Moreover, the multiplicity given in the target UML class model also helps in identifying a particular type of quantification. For example, in figure 7, the multiplicity '0…1' specifies that customer can get at most one credit card. This will be equal to At-most n quantification in SBVR.

$_{\textbf{Univeral_Quantification}}$[A] $_{\textbf{Object_Type}}$[directory object] $_{\textbf{verb_concept}}$[is assigned] to $_{\textbf{Univeral_Quantification}}$[all] $_{\textbf{Object_Type}}$ [files] with $_{\textbf{Object_Type}}$ [directory].

Figure 8. Semantic roles assigned to input English sentence

iii. Semantic Interpretation: After shallow and deep semantic parsing, a final semantic interpretation is generated that is mapped to SBVR and OCL in later stages. 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 9 shows an example of the semantic interpretation we have used in the NL to OCL approach:

```
(assign
    (object_type = (∃=1X ~ (DirectoryObject ? X)) AND
          (object_type = (∃=1Y ~ (Directory? Y)))
    (object_type = (∀Z ~ (Files ? Z))))
```

Figure 9. Semantic roles assigned to input English sentence.

2) SBVR to OCL Transformation

Once we get the SBVR based logical representation of English constraint, it is mapped to the OCL by using model transformation technology using SiTra. A set of model transformation rules were used for SBVR to OCL transformation (Bajwa, 2011). For model transformation of NL to OCL, we need following two things to generate OCL constraints:

i. Select the appropriate OCL template (such as invariant, pre/post conditions, collections, etc)

ii. Use set of mappings that can map source elements of logical form to the equivalent elements in used OCL templates.

a) OCL Templates: We have designed generic templates for common OCL expressions such as OCL invariant, OCL pre-condition, and OCL post-condition. User has to select one of these three templates manually. Once the user selects one of the constraints, the missed elements in the template are extracted from the logical representation of English constraint. Following is the template for invariant:

In the all above shown templates, elements written in brackets '[ ]' are required. We get these elements from the logical representation of English sentence. Following mappings are used to extract these elements:

i. *UML-Package* is package name of the target UML class model.
ii. *UML-Class* is 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 we get them 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 we get them 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.

3) OCL to Alloy Transformation

The Alloy module maps an OCL expression to Alloy code by using model transformation that incorporates the mapping rules between OCL and Alloy. Every OCL *invariant* expression maps to an Alloy *fact* statement. OCL conjunction, disjunction and negation statements (i.e. *and*, *or, not)* have a direct mapping to the Alloy conjunction (*&&*), disjunction (*//*) and negation (*!*) operators. Most of the OCL operations on collections have a corresponding Alloy expression.

For example, the *forAll()* and *exists()* operations can be mapped to the *all* and *some* Alloy expressions respectively. Similarly the *size()* OCL operation can be represented by the Alloy set cardinality operator (#). The *isEmpty()* and *notEmpty()* operations are expressed using the no and some Alloy keywords. More detailed information on the OCL to Alloy transformation can be found in [30].

IV. A Running Example

To explain the presented approach, we have applied our approach to the following NL text:

Consider a model of file system in which *every entity is a DirectoryObject. A DirectoryObject could be a File or a Directory. Each Directory may include a number of DirectoryObjects, which are the entries of the directory. If a DirectoryObject has a Directory as its entity, it is its parent. A DirectoryObject is assigned to all Files with Directory. There is a root Directory without any parents. A directory cannot not be a parent of itself.*

The above example was used described in Section 3. Figure 6 UML Class Diagram of a Simple File System Model depicts a UML class diagram of the file system. This UML Class Diagram model was generated manually from the text description, but tools like CM-Builder [] can be used to automate this task.

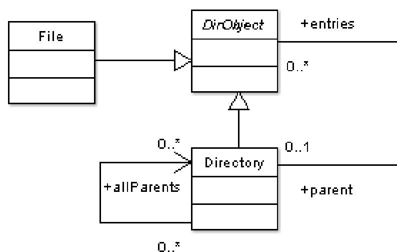

Figure 6 UML Class Diagram of a Simple File System Model

In the following we present the transformation of the NL statements to SBVR, then to OCL and finally to Alloy.

Constraint1*:*

English: There is exactly one directory that has no parent.

SBVR: **It is obligatory** that there *is* **exactly** one directory that *has* no parent.

OCL:
```
context Directory
inv oneRootDirectory : Directory.allInstances() ->
select ( d : Directory | d.parent -> isEmpty() ) ->
        size() = 1
```

Alloy:
```
fact { Directory_oneRootDirectory[]}
    pred Directory_oneRootDirectory[]{
  # { d: Directory | no d . parent } = 1}
```

Constraint2*:*

English: A directory may not be a parent of itself.

SBVR: **It is possibility** that a directory *may* not *be* a parent of itself.

OCL:
```
context Directory
    inv: self.parent -> excludes (self)
```

Alloy:
```
  fact{all      self:      Directory      |
Directory_notAncestorOfItself[ self ] }
pred Directory_notAncestorOfItself[self: Directory]{
            self !in self.parent }
```

---

1) Analysis

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. First we try to simulate the model with a *scope* of [31]. 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).

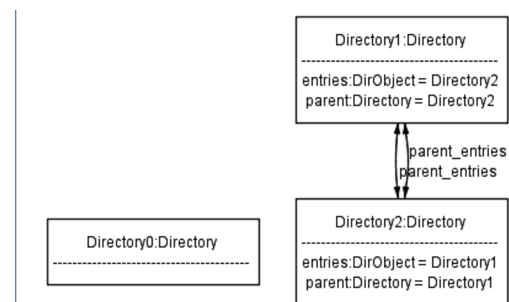

Figure 7 Instance provided by the Alloy Analyzer

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 [32 p. 55] recursion is used to express the *allParents()* operation to express that a Generalization relation between UML Classes is acyclic and directed.

In a similar approach we use an auxiliary self association on

the Directory class as shown in Figure 7. This self association relates a Directory to all its direct and indirect parents (through the allParents association end). We replaced Constraint1, so that instead of the "parent" it uses the "allParents" reference. After this change, simulating the system provided only valid instances.

V. EVALUATION

NL2Alloy tool was used to translate 10 examples, similar as solved in section 4. All examples were containing a UML mode and different English descriptions of examples to generate Alloy code. The largest English example was composed of 23 words and the smallest sentence was composed of 9 words. We calculated total required (sample) elements in all 10 examples and extracted (correct, incorrect, missing) elements from English description. The Calculated recall, precision and f-values of the solved examples are shown in table I.

Table I: Evaluatin results of NL to Alloy

Type	N_{sample}	$N_{correct}$	$N_{incorrect}$	$N_{missing}$	***Rec***	***Prec***	***F-Value***
Data	48	43	4	1	89.58	91.48	90.07

The average F-value is calculated 82.78 that is encouraging for initial experiments. We cannot compare our results to any other tool as NL-based constraint tool is a novel idea. However, we can note that other language processing technologies, 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 NL2Alloy approach and the potential of this technology in general.

VI. CONCLUSION

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.

REFERENCES

[1] Vangheluwe, H., Lara, J.Computer automated multi-paradigm modelling: Meta-modelling and graph transformation. In Winter Simulation Conference, pages 595 - 603. Dec 2003. New Orleans.

[2] OMG. 2007. Unified Modeling Language (UML), OMG Standard, v. 2.3.

[3] OMG. 2006. Object Constraint Language (OCL), OMG Standard, v. 2.0.

[4] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, London, England (2006)

[5] Kitchin, D.E., McCluskey, T.L. and West, Margaret M. B vs OCL: comparing specification languages for Planning Domains. In: Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)

[6] OMG. 2008. Semantics of Business vocabulary and Rules (SBVR), OMG Standard, v. 1.0.

[7] Michael Azoff , The Benefits of Model Driven Development: MDD in Modern Web-based Systems, Butler Group, Marc 2008, Available at: http://www.ca.com/~/media/Files/whitepapers/the-benefits-of-model-driven-development.pdf

[8] Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. UML2Alloy: A Challenging Model Transformation, ACM/IEEE 10TH International Conference on Model Driven Engineering Languages and Systems, LNCS, Vol. 4735, pages 436-450, 2007

[9] Bajwa, I.S., Lee, M.G. Transformation Rules for Translating Business Rules to OCL Constraints. in ECMFA 2011- Seventh European Conference on Modelling Foundations and Applications, Birmingham, UK, June 2011

[10] Raj, A., Prabhakar, T.V., Hendryx, S. Transformation of SBVR business design to UML models, Proceedings of the 1st India software engineering conference, February 19-22, 2008, Hyderabad, India

[11] Cabot J., et al.: UML/OCL to SBVR Specification: A challenging Transformation, Journal of Information systems doi:10.1016/j.is.2008.12.002 (2009)

[12] Moschoyiannis, S., Marinos, A., Krause, P.J.: Generating SQL Queries from SBVR Rules. In RuleML(2010) 128-143

[13] S. Shah, K.Anastasakis, B. Bordbar, 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

[14] Bajwa I., Behzad B., Lee M., OCL Constraints Generation from Natural Language Specification. EDOC 2010 – 14th IEEE EDOC Conference, Vitoria, Brazil, pp. 204-213. (2010)

[15] Bajwa, I.S., Lee M.G., Behzad B. SBVR Business Rules Generation from Natural Language Specification. AAAI 2011 Spring symposium – AI for Business Agility, San Francisco, USA, pp. 2-8. (2011)

[16] Richters, M. A Precise Approach to Validating UML Models and OCL Constraints. Universitaet Bremen. Berlin : Logos Verlag, 2002. BISS Monographs, No. 14.

[17] Toutanova. K., Manning, C.D. 2000. Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. In Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: 63-70.

[18] Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture{Practice and Promise. The Addison-Wesley Object Technology Se-ries. Addison-Wesley (2003)

[19] Akehurst, D.H., Boardbar, B. et al. SiTra: Simple Transformations in Java, ACM/IEEE 9TH International Conference on Model Driven Engineering Languages and Systems, LNCS, Vol. 4199, pages 351-364, 2006

[20] K. Anastasakis, "A Model Driven Approach for the Automated Analysis of UML Class Diagrams," University of Birmingham, PhD Thesis , 2009

[21] Mendel, L. Modeling By Example. M.Eng. Thesis. 2007. Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[22] Object Management Group. UML Superstructure Specification 2.3. 2009.

[23] Zettlemoyer, L.S., Collins, M. (2009). Learning Context-dependent Mappings from Sentences to Logical Form. in Joint Conference of the Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP), 2009.

[24] Baral, C., Dzifcak, J., Gonzalez, M.A., Zhou, J. 2011. Using Inverse lambda and Generalization to Translate English to Formal Languages, in 9th International Conference on Computational Semantics (IWCS 2011), Oxford, UK, pp:35-44

[25] Marie-Catherine de Marneffe, Bill MacCartney and Christopher D. Manning. 2006. Generating Typed Dependency Parses from Phrase Structure Parses. In LREC 2006.

[26] NL2Alloy Webpage, Available at: http://www.cs.bham.ac.uk/~bxb/NL2OCLviaSBVR /NL2Alloy.html

[27] Dennis , G., Seater R., Rayside D., and Jackson D. Automating Commutativity Analysis at the Design Level. In ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, pages 165–174. ACM Press, 2004.