Semi-automatic Generation of UML Models from Natural Language Requirements

Deva Kumar Deeptimahanti
Lero- The Irish Software Engineering Research Centre
University of Limerick
Limerick, Ireland
deva.kumar.deeptimahanti@lero.ie

Ratna Sanyal
Indian Institute of Information Technology- Allahabad
Uttar Pradesh, India
rsanyal@iiita.ac.in

ABSTRACT
Going from requirements analysis to design phase is considered as one of the most complex and difficult activities in software development. Errors caused during this activity can be quite expensive to fix in later phases of software development. One main reason for such potential problems is due to the specification of software requirements in Natural Language format. To overcome some of these defects we have proposed a technique, which aims to provide semi- automated assistance for developers to generate UML models from normalized natural language requirements using Natural Language Processing techniques. This technique initially focuses on generating use-case diagram and analysis class model (conceptual model) followed by collaboration model generation for each use-case. Then it generates a consolidated design class model from which code model can also be generated. It also provides requirement traceability both at design and code levels by using Key-Word-In-Context and Concept Location techniques respectively to identify inconsistencies in requirements. Finally, this technique generates XML Metadata Interchange (XMI) files for visualizing generated models in any UML modeling tool having XMI import feature. This paper is an extension to our existing work by enhancing its complete usage with the help of Qualification Verification System as a case study.

Categories and Subject Descriptors
D.2.1 [**Software Engineering]:** Requirements/ Specifications - *Methodologies (UML model generation, structured), Tools.* D.2.2 [**Software Engineering]:** Design Tools and Techniques- *Object-oriented design methods, Computer-aided software engineering (CASE).* I.2.7 [**Artificial Intelligence**]: Natural Language Processing – *Text analysis.*

General Terms
Design, Experimentation

Keywords
Requirement engineering, Natural Language Processing, Unified Modeling Language

1. INTRODUCTION
Analyzing requirements, building analysis and design models are cumbersome and complicated tasks, which need automated support. Errors caused during these phases can be quite expensive to fix later on. One of the main reasons for potential problems is the way requirements are specified and analyzed. Software requirements are most often specified in Natural Language (NL). In a typical software industry, Software Requirement Specifications (SRS) is written in NL to enhance communication between different stakeholders [1]. Michael Jackson [2], mentions “requirement engineering is where informal meets formal”. So NL SRS are very useful at this development stage for understanding between both the groups. However, requirements described in NL can often be ambiguous, incomplete, and inconsistent [1]. Moreover, the interpretation and understanding of anything described in NL has the potential of being influenced by geographical, psychological and sociological factors. It is usually the job of requirements analysts to detect and fix potential ambiguities, inconsistencies, and incompleteness in requirements. But, human reviewers can overlook defects in NL requirements, which can lead to multiple interpretations and difficulties in recovering implicit requirements if analysts do not have enough domain knowledge [3, 4]. Furthermore, it is not possible to perform complete automated analysis of NL requirements in order to detect and resolve these problems. But, tool support for automating some of these tasks is highly desirable.

However, building models from NL requirements is difficult and time-consuming task. Design models such as use-cases, sequence diagrams, and class diagrams are usually used to document requirements in a more structured way. Some of these models have formal semantics and are amenable to automated analysis. Many approaches and tools have been proposed to bridge the gap between requirement analysis and design phases by developing Object Oriented (OO) models from requirements [5-8]. Section 2 discusses some of these approaches and addresses their limitations. Our previous work, Static UML model Generator from Analysis of Requirements (SUGAR), [9, 10] just focused on generating static UML models such as use-case and analysis class models from NL requirements. This paper is an extension to our earlier work [9-11] describing a technique named as UML Model Generator from analysis of Requirements (UMGAR), to provide semi-automated support for developing both static and dynamic UML models from NL requirements and evaluated with the help of a case study. Our technique exploits three efficient NLP techniques for parsing sentences and performing various analyses such as pronoun resolution, morphological analysis.

Next section discusses the brief literature review along with limitations of existing research followed by contributions from UMGAR. Section 3 describes the technologies used, process architecture of UMGAR. Section 4 describes the use of UMGAR with a case study. Finally section 5 concludes the paper and provides our future focus.

2. BACKGROUND AND MOTIVATION

Several research works identify the possible usage of Natural Language Processing (NLP) techniques over NL requirements [13], precisely:

1. Scanning of requirement documents
2. Searching requirements from these documents
3. Extracting requirements from documents
4. Tagging the text for identifying many things
5. Finding similar or duplicate requirements
6. Finding probably ambiguous requirements

In a pioneering work, Ryan [13] claims that NLP is not suitable to be used in requirement engineering (RE), as NL would not provide a reliable level of understanding, and even if it could, using such resulting system in RE is highly questionable. The primary reason is the complexity of the requirements themselves. But later on Kof [14] opposes Ryan's [13] and claims as:

1. NLP usage in RE is not to understand text but to extract concepts contained in NL document;
2. Even though domain model generated from NL document is incomplete as information that is thought to be "common domain knowledge" is omitted in the requirements text. As the task of requirement engineer is to detect such omissions, such incomplete domain model can act as an indicator for some omissions in text.

Kamsties [15] mentioned that problems with requirements can possibly be avoided either by detecting ambiguity or by using a restricted form of such language. Berry [16] proposed following features to reduce disadvantages of using NL specifications.:

1. Learn to write less ambiguously and less imprecisely
2. Learn to detect ambiguity and imprecision
3. Use a restricted natural language which is inherently unambiguous and more precise

In a nutshell, even though several problems exist with NLP in processing text, there are quite a number of NLP applications developed in RE. In relation to generate UML models from NL requirements, there have been several attempts at providing tooling support. Based on an extensive literature review, due to space limitation the following previously cited papers provide a short critique of existing tools for automatically generating UML models from NL requirements. We divide this discussion into two categories as structural models and behavioral models.

2.1 Structural Model Generation

A Natural Language- Object Oriented Production System (NL-OOPS) LOLITA was proposed by Mich [7] which generates OO analysis models from SemNet obtained by parsing NL SRS documents. It considers nouns as objects and identifies relationship among objects using links. This approach lacks accuracy in selecting objects for large systems and cannot differentiate objects and attributes.

Börstler [17] provides a tool for constructing an object model automatically based on pre-specified key words in use-case description. The verbs in the key words are transformed to behaviors and nouns to objects, but require excessive user interaction to associate behavior to the object. Nanduri and Rugaber [18] developed a tool using syntactic knowledge by extracting objects, methods and associations and generates object diagram from NL SRS. However, these models are validated manually and user needs to have extensive domain knowledge.

CM-Builder analyzes requirements texts and builds a Semantic Network, to construct an initial UML Class Model [5]. This model can be visualized in a graphical CASE tool by converting it into standard data interchange format, CDIF where human analyst can make further refinement to generate final UML models. However, CM-Builder makes extensive use of NLP techniques for generating only analysis class model.

Linguistic assistant for Domain Analysis (LIDA) [8] identifies model elements through assisted text analysis and validates by refining the text descriptions of the developing model. LIDA needs extensive user interaction while generating models because it identifies only a list of candidate nouns, verbs and adjectives, which need to be categorized into classes, attributes or operations based on user's domain knowledge. Another tool has been developed by Popescu et.al [19] with the aim of identifying ambiguity, inconsistency and under specification in requirement documents by creating object-oriented models automatically by parsing NL SRS according to constraining grammar. These are later diagrammed which enable human reviewer to detect ambiguities and inconsistencies.

2.2 Behavioral Model Generation

There are relatively few attempts at providing tools for generating behavioral models like sequence or collaboration models from NL use-case specifications, from which design class model is generated. Li [20] reports a semi-automatic approach to translate narrative use-case descriptions to sequence diagrams using syntactic rules and parser. He proposed eight syntactic rules to handle simple sentences which need human intervention which are insufficient to handle different types of verb phrases.

Use Case Driven Development Assistant Tool (UCDA) [6] generates Class Model by analyzing NL requirements. It assists in generating use-case diagrams, use-case specifications, robustness diagrams, collaboration diagrams and class diagrams. Our approach is similar to this, but UMGAR uses accurate NLP tools in extracting models and provided design traceability mechanisms and grammatical rules for collaboration diagram generation. Main disadvantage of UCDA is that it depends on Rational Rose, a very expensive environment, for visualizing UML models. Montes et.al [21] and Diaz et.al [22] developed a tool to generate conceptual model, sequence diagrams, and state diagrams by analyzing a system's textual descriptions of the use-case scenarios in Spanish language.

Yue et.al [23] proposed a method to generate activity diagrams from use-case specifications using transformation rules. In our case, we generated collaboration diagrams from use-case specifications, as activity diagrams fails in representing which objects execute which activities, and the order in which messaging works between them. Similarly a commercial tool named Ravenflow [24] provides mechanism to generate activity diagrams (process diagrams) from structured text written using rewriting rules. This has major limitation in representing alternative flows.

2.3 Comparative Analysis

It is obvious from the previous discussion that there have been several efforts to exploit the NLP technologies for automating requirements analysis phase. In this section, we provide a brief comparative analysis of existing tools and their respective limitations that have provided the motivation for developing UMGAR. This discussion is based on the comparative analysis of NLP systems provided by Li et.al [25].

Table 1 shows comparison among available tools with respect to features mentioned above (Y- Yes, N- No, High- H (High user interaction), Medium- M, and Low-L).

Table 1. Comparison of available tools

Feature	**NL-OOPS [7]**	**RECORD [17]**	**Nanduri [18]**	**CM-Builder [5]**	**LIDA [8]**	**Popescu [19]**	**UCDA [6]**	**Li [20]**	**Montes [21, 22]**	**SUGAR [9, 10]**
Use-case Model	N	N	N	N	N	N	Y	N	N	Y
Analysis Class model	Y	N	Y	Y	Y	Y	Y	N	Y	Y
Interaction diagram	N	N	N	N	N	N	Y	Y	Y	N
Design class model	N	Y	N	N	N	N	Y	N	Y	N
XMI support	N	N	N	Y	N	N	N	N	N	N
Normalize requirements	N	Y	N	Y	N	N	Y	Y	N	Y
User Interaction	H	H	H	M	H	M	L	H	M	L

Following limitations are interpreted using table:

1. Most existing tools do not provide automatic normalization of NL requirements. Absence of such causes information loss when processing complex requirements.
2. Most of them just focus on generating analysis class model; and only tools reported in [6, 17, 21, 22] generate design class model. None of the existing techniques are able to generate full set of possible UML models.
3. Only CM-Builder [5] provide an intermediate representation of the generated models to visualize these in CDIF supported UML modeling tools.
4. The existing tools also limit the size of requirements to be handled and can work on very small set of requirements (typically <200 words) [25] and require developers support to refine models and identify inconsistencies in requirements.

2.4 Contributions of UMGAR

Existing tools provide varying level of automation. We believe that complete automation of this activity with NLP may be impossible in the near future. However, more advanced automation can be provided. To address existing limitations, UMGAR makes following contributions aimed to support automatic generation of possible UML models from SRS written in natural language without need of extensive domain expertise and user interaction:

1. UMGAR follows Use-case Driven Object- Oriented Analysis and Design (OOAD) [26] techniques for object elicitation from complex NL requirements using efficient NLP tools like Stanford Parser [27], JavaRAP [28], and WordNet 2.1 [29], which can handle large requirements documents.
2. It provides a XMI parser to generate XMI file [12] and can be visualized in any UML modeling tool which has XMI import feature.
3. We have proposed eight syntactic reconstruction rules [9-11] to handle large NL SRS and normalize complex requirements into simple ones to reduce their ambiguity. It can also handle compound word morphological analysis where WordNet [29] fails, identified 247 determiners which are specific for requirements engineering.
4. Provides traceability from requirements down to design and implementation phases using Key-Word-In-Concept (KWIC) and concept location feature [30] respectively.
5. UMGAR uses a glossary to avoid any communication gaps among team members and creates unambiguous requirements.
6. We have proposed eight rules for generating collaboration diagram from use-case specification template covering major event flows which occurs in use-case specifications.

3. THE APPROACH

UMGAR aims to assist requirements analysts and designers in generating analysis and design class models from NL requirements using sophisticated NLP technologies reported in [27-29]. Currently UMGAR can generate use-case diagrams, analysis class model, collaboration diagram, and design class model.

3.1 Process Architecture of UMGAR

UMGAR's process architecture as shown in figure 1 consists of two components:

1. Normalizing requirements component (NLP Tool Layer)
2. Model Generator component

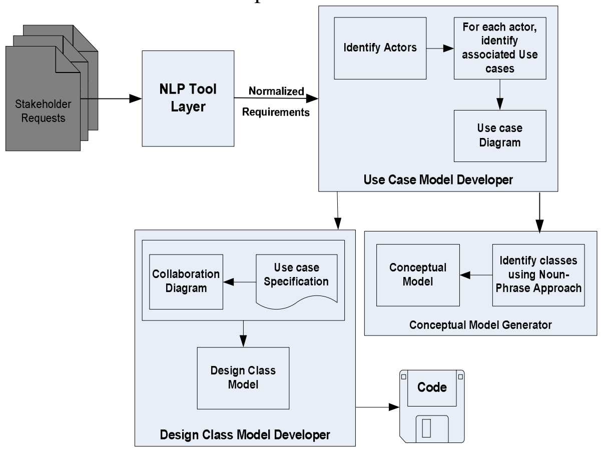


Figure 1. The process architecture of UMGAR

3.1.1 Normalizing requirements component

This component has two sub-components which aims at normalizing (rewriting) NL requirements to remove ambiguity in sentence structures and introduces control constructs to organize interactions in the statement sentence structure [31].

3.1.1.1 Syntactic Reconstruction

UMGAR accepts stakeholders' requirements in NL format as input and decompose complex sentence structure into simple sentence structures to extract all possible information from the requirements document. Since, there is no tool available to perform automatic syntactic reconstruction of NL sentences; we have defined eight syntactic reconstructing rules that have been implemented in UMGAR [9-11] UMGAR scans each sentence to test whether that requirement satisfies the Statement sentence structure which is of the form "Subject: Predicate" or "Subject: Predicate: Object", and applies rules accordingly. Subject and object are usually represented as a noun phrase (denoted by NP) and a predicate as verb phrase (denoted as VP). Currently UMGAR accepts requirements expressed in active-voice form. If a requirement sentence does not satisfy the following rules, then it prompts a message to user to change sentence accordingly to the statement structure.

3.1.1.2 Technologies Used

Following NLP tools are applied on normalized requirements:

1. *Stanford Parser* [27] is a highly optimized probabilistic context-free grammar (PCFG) NL parser implemented in java. It is used to generate parse tree for each requirement from which artifacts like actors, use-cases, classes, methods, associations, and attributes can be extracted.
2. *WordNet2.1* [29]- UMGAR uses WordNet to perform morphological analysis for converting plurals into singulars. But it fails in case of handling plural form for compound words, for which we implemented twenty-one rules to nullify the effect of obtaining more classes. For example, for "Verification officers", WordNet strip off suffix and return just the word "Verification" which creates ambiguity.
3. *JavaRAP* [28] helps resolve pronouns up to third person pronouns which is an implementation of classic Resolution of Anaphora Procedure (RAP) algorithm given by Lappin and Leass. JavaRAP [28]. UMGAR uses this to replace all the possible pronouns with its correct noun form.

3.1.2 Model Generator component

This component generates various OO models like use-case diagram, analysis class model, and collaboration diagram and design class model from normalized requirements. This component consists of following three sub- components:

3.1.2.1 Use-case Model Developer

From normalized statement sentence ("Subject: Predicate" or "Subject: Predicate: Object"), UMGAR identifies subject and object as actors, predicates as use-cases, and associates actors and use-cases using parse tree generated from Stanford Parser [27].

3.1.2.2 Analysis class model developer

UMGAR uses a combination of Noun-phrase technique [32] and RUP [33] to generate analysis class model. This component uses Stanford Parser [27] to identify all candidate classes from requirements and generates analysis class model by attaching attributes and methods with associated class object. Glossary is checked against candidate classes to eliminate redundant classes. To eliminate ambiguity, morphological analysis using WordNet [29] is performed to suppress plural form of class objects to singular form.

3.1.2.3 Design class model developer

Use-case scenario should be manually specified using the rule "Who do what to whom?" which describes who initiates the message, and what it wants to send and to whom. The collaboration diagram is generated using proposed rules in section 4.3.1. Stanford Parser is used to parse both basic and alternative flows in the use-case specification template to identify sender, receiver and the messages between them. UMGAR generates a design class model from the generated collaboration diagram.

Figure 2 shows the sequence of steps followed while using UMGAR. Also process of generating models is explained clearly along with a case study in the following section. More explanations are given in Section 4.

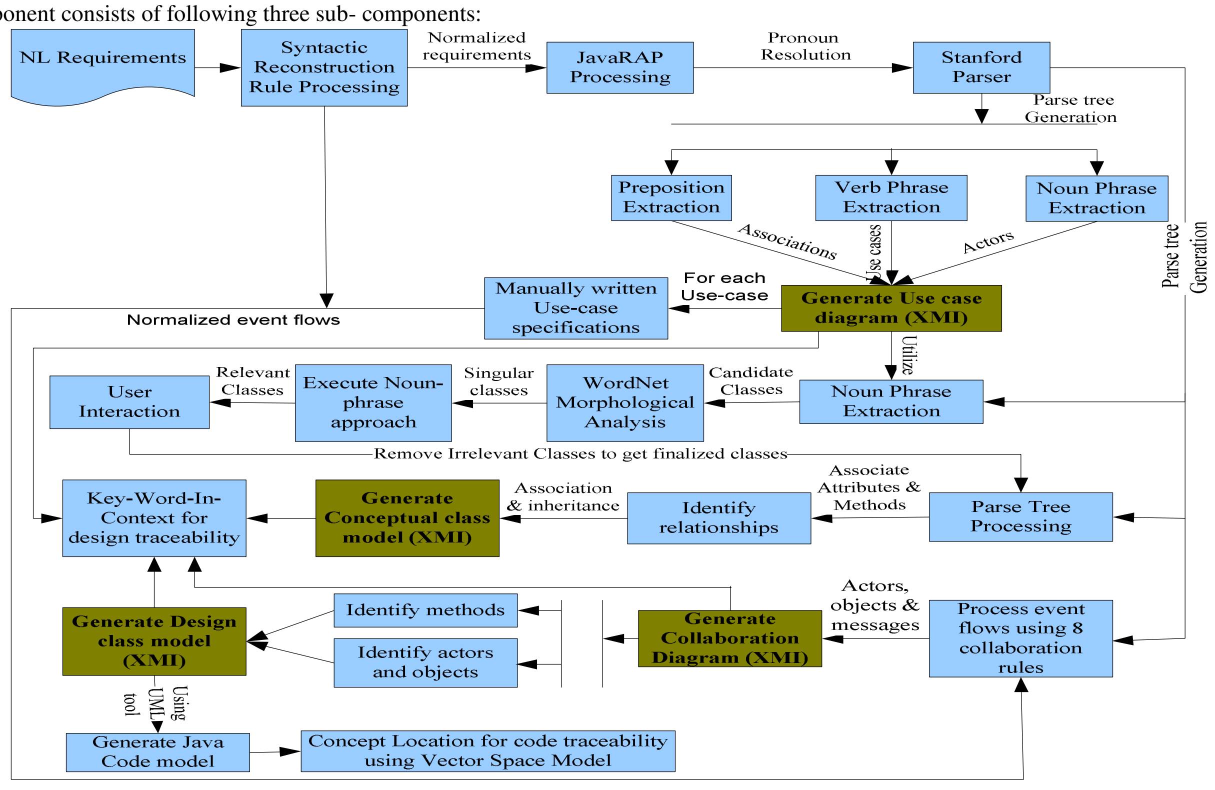

NL Requirements
Syntactic Reconstruction Rule Processing
Normalized requirements
JavaRAP Processing
Pronoun Resolution
Stanford Parser
Parse tree Generation
Preposition Extraction
Verb Phrase Extraction
Noun Phrase Extraction
Associations
Use cases
Actors
Parse tree Generation
Manually written Use-case specifications
For each Use-case
Generate Use case diagram (XMI)
Normalized event flows
Utilize
User Interaction
Relevant Classes
Execute Noun-phrase approach
Singular classes
WordNet Morphological Analysis
Candidate Classes
Noun Phrase Extraction
Remove Irrelevant Classes to get finalized classes
Key-Word-In-Context for design traceability
Generate Conceptual class model (XMI)
Association & inheritance
Identify relationships
Associate Attributes & Methods
Parse Tree Processing
Identify methods
Identify actors and objects
Generate Design class model (XMI)
Generate Collaboration Diagram (XMI)
Actors, objects & messages
Process event flows using 8 collaboration rules
Using UML tool
Generate Java Code model
Concept Location for code traceability using Vector Space Model

Figure 2. Work Flow of UMGAR

4. USING UMGAR

This section explains the process of using UMGAR along with a case system "Qualification Verification System (QVS)", which has twenty-six requirements with some complex requirements as follows:

Candidate will register with system to hire services. Candidate can provide information about his academic, work experience and referee details. Candidates may make verification request during registration or some time later. Candidate updates his details at any time. System will inform service seeker about approximate time period required to provide service. System records details of candidate. System shall keep status of each request up-to-date. System shall interact with education institute systems to verify originality of degree. System asks type of service required. Service may be Standard, Silver or Gold. System enters request as a record in system. System informs about outcome to the candidate. Referees send recommendation letters using system on behalf of students. Customers pay fee for each type of service. Customer may be candidate or employer. Standard service verifies details of education. Silver service verifies details of education and profession. Gold service verifies details of education, profession and recommendation letters from referees. Employer accesses system to hire services of system for verification purposes. Employer registers with system. After, employer should provide system with information on type of service required along with candidate unique identification number. System records details of employer. Verification officer accesses system to retrieve verification and inquiry requests. Verification officer performs all types of verifications requested by candidate and employers upon retrieving requests. Verification officer also verifies authenticity of referee. Verification officer uses system to send to referees a request for a recommendation letter when candidate requests gold service.

Using proposed syntactic reconstruction rules [9-11], all these 26 requirements are normalized automatically into 33 simple sentences, so that each requirement can be of the form "Subject-Predicate" or "Subject-Predicate-Object". Using third reconstruction rule [10], compound requirement - "Candidate can provide information about his academic, work experience and referee details" can be reconstructed as "Candidate can provide information about his academic details. Candidate can provide information about his work experience details. Candidate can provide information about his referee details."

This normalized document is later parsed using JavaRAP tool [28] to replace pronouns by corresponding nouns. Specific determiners are removed from the simplified text using list of identified 247 determiners to remove ambiguity to further extent. Stanford Parser [27] displays parse tree for each simple requirement statement in subject, predicate and object form from which required information can be extracted as shown in Figure 7.

4.1 Use-case Model Developer

This section explains process for generating use-case model from processed NL requirements using following steps.

4.1.1 Identifying the Actors

Candidate actors are identified by examining who is using a system or who is affecting a system or for whom system is intended [26]. Actors mostly are subjects (Noun Phrases (NP)) and objects of a sentence. UMGAR extracts all such noun phrases which tend to be actors.

From this case study, a total of 9 noun phrases are identified namely: candidate, employer, verification officer, referee, standard service, silver service, gold service, customer and system. As actor is one who is interacting with system from outside, so System noun phrase is not treated as an actor. As a result, QVS finally has 8 actors as shown in Figure 3.

4.1.2 Identifying the Use-cases

The process of identifying use-case is related to actors [26]. The use-cases identified by UMGAR are mostly the predicates (verb phrases (VP)) in the sentence which are associated with an actor (subject) (NP: VP).

Each verb phrase should contain a verb denoted as VBZ (singular verb). VP's is an important category which has seven forms like transitive, intransitive, di-transitive, prepositional, intensive, complex transitive and non-finite [31]. Requirement structure mainly depends on the verb phrase type. So, VP's are properly analyzed to extract exact use-cases associated with the actor. Figure 3 shows 23 use-cases that are identified. 8 use-cases associated with System are not shown.

4.1.3 Relationships between actors and use-cases

Association relationships for the use-case diagram can be identified from the sentences which are of the form Subject-Predicate-Object, and by identifying prepositions between predicates and objects. In some cases, predicates internally contains NP which represent an actor (e.g.: Verb + Preposition + Noun), at such cases, UMGAR identifies prepositions like "from, to, about, with, in etc." and associates use-case and actor together. As actors like standard, silver and gold services share common use-cases, UMGAR identifies such commonalities and represent them together. So, generated use-case diagram shows a total of 20 use-cases. Use-case model generated and visualized in Enterprise Architect [35] is as shown in Figure 3.

4.2 Analysis Class Model Generator

UMGAR uses Noun-Phrase approach [32] and RUP [33] for identifying classes. Using Stanford parser [27], identified nouns are considered as candidate classes, verbs as methods and adjectives as attributes of associated class. UMGAR aims at associating these attributes and methods to the associated class.

Noun-phrase approach initially considers all identified nouns from the case study as candidate classes. To suppress plural forms of the noun phrases, morphological analysis on nouns are performed using "Word Net 2.1" [29]. After doing so, UMGAR obtains a total of 18 objects namely: Candidate or employer, candidate, verification officer, service seeker, registration, customer, gold service, education institute system, service, standard service, any time, employer, referee, silver service, verification request, system, request, "standard, silver or gold". These 18 classes are categorized into following classes, to include or exclude a particular class:

1. *Redundant classes:* UMGAR maintains a glossary with all noun phrases occurring in the system along with their synonyms. Using glossary, UMGAR identifies service seeker as similar to customer, as a result customer replaces service seeker context throughout the document.

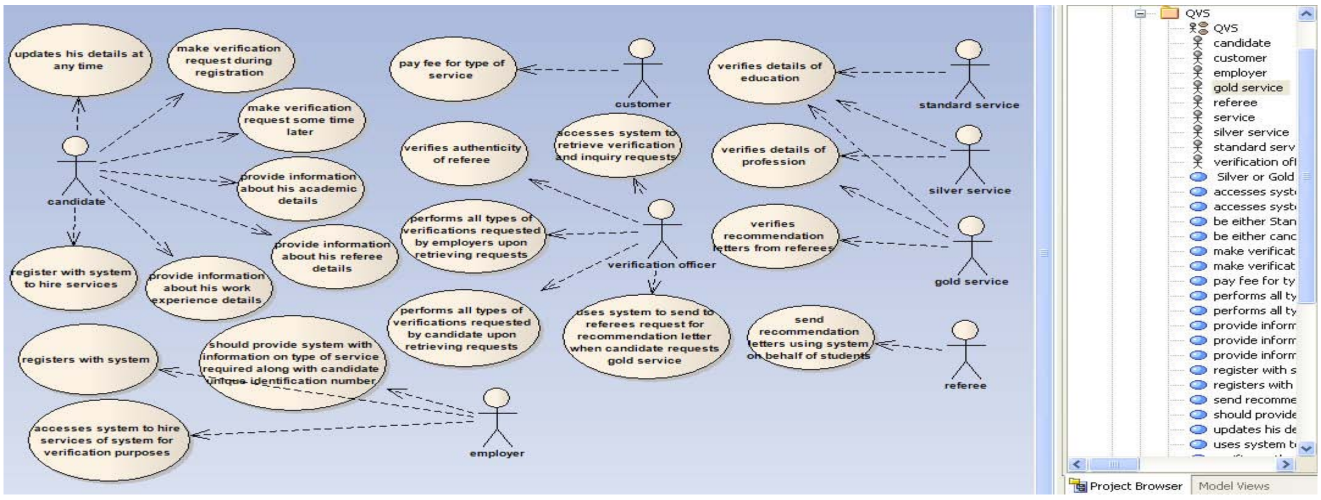
2. *Attribute classes:* If a particular class represents values or list of values (Boolean, list, etc.), such classes are treated as attributes of a class but not as a class. UMGAR maintain a list of such words observed from various requirement documents in a separate document which carries values or list of values (attribute sense), and if such words occurs in the noun lists are eliminated. For this case, UMGAR identifies “any time” as an attribute class which depicts some invariable time to update candidate details.
3. *Adjective classes:* If the identified noun-phrase contains an adjective, then it is treated as an adjective class. Adjective classes for this case are Standard Service, Silver Service and Gold Service. As none of these adjective classes are found in attribute class list, all these three classes are included to final class list. If any class is found in both adjective and attribute class list, UMGAR removes such class from the list.
4. *Irrelevant classes:* Manual interference is needed to identify irrelevant classes at this point. For this case study, from remaining classes, “request”, “verification request”, “candidate or employer”, “registration”, “standard, silver or gold” are eliminated. Words “request”, “verification request” and “registration” can be methods of a class, and “candidate or employer” and “standard, silver or gold” are already individually treated as classes.

Finalized classes after executing the Noun-Phrase technique is turned out to be candidate classes as shown in Figure 4.

4.2.1 Relationships among Objects

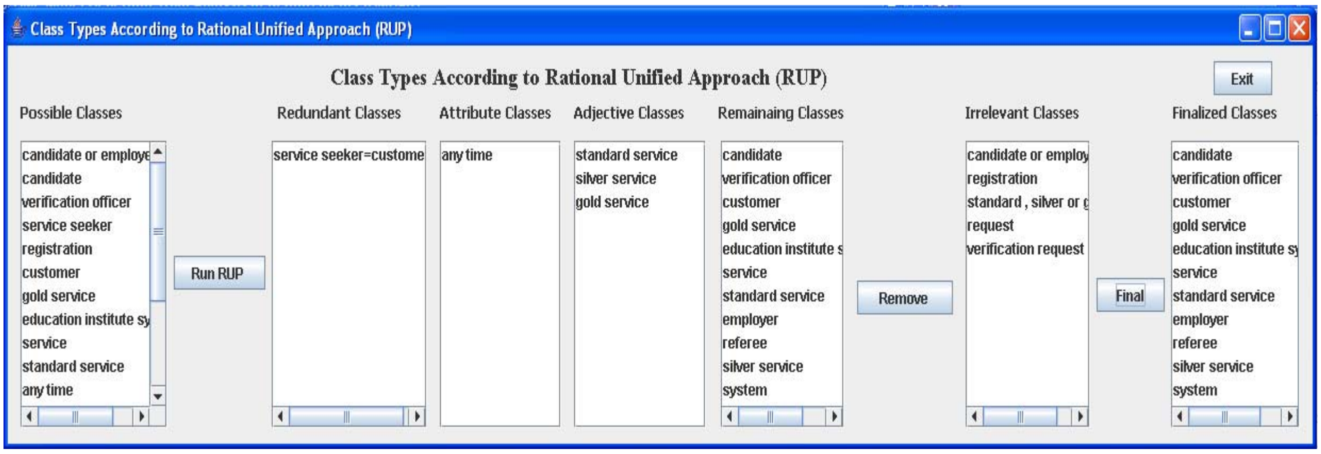
Using UMGAR, following relationships are identified from the case study. The current version of UMGAR can successfully handle association and inheritance relationships. We are working on implementing aggregation relationships.

1. *Association:* Identifying association relationship is done in two phases, first by searching prepositional phrases between noun phrases, such as “has, next to, part of, works for, contained in, and talk to”. And if the parsed sentence is in the form of Subject: Predicate: Object, then it associates subject and object accordingly. A total of 7 association relationships are identified. For example, “System records details of candidate”, which is of the form Subject: Predicate: Object, in which system (subject) and candidate (object) are associated with each other.



updates his details at any time
make verification request during registration
make verification request some time later
provide information about his academic details
provide information about his referee details
provide information about his work experience details
register with system to hire services
registers with system
should provide system with information on type of service required along with candidate unique identification number
accesses system to hire services of system for verification purposes
candidate
employer
pay fee for type of service
customer
verifies authenticity of referee
accesses system to retrieve verification and inquiry requests
performs all types of verifications requested by employers upon retrieving requests
performs all types of verifications requested by candidate upon retrieving requests
uses system to send to referees request for recommendation letter when candidate requests gold service
verification officer
verifies details of education
verifies details of profession
verifies recommendation letters from referees
send recommendation letters using system on behalf of students
standard service
silver service
gold service
referee
QVS
QVS
candidate
customer
employer
gold service
referee
service
silver service
standard serv
verification offi
Silver or Gold
accesses syst
accesses syst
be either Stan
be either canc
make verificat
make verificat
pay fee for ty
performs all ty
performs all ty
provide inform
provide inform
provide inform
register with s
registers with
send recomme
should provide
updates his de
uses system t
Project Browser
Model Views

Figure 3. Use-case Diagram for Qualification Verification System (QVS)



Class Types According to Rational Unified Approach (RUP)

Possible Classes	Redundant Classes	Attribute Classes	Adjective Classes	Remainaing Classes	Irrelevant Classes	Finalized Classes
candidate or employe	service seeker=custome	any time	standard service	candidate	candidate or employ	candidate
candidate			silver service	verification officer	registration	verification officer
verification officer			gold service	customer	standard , silver or g	customer
service seeker				gold service	request	gold service
registration				education institute s	verification request	education institute sy
customer				service		service
gold service				standard service		standard service
education institute sy				employer		employer
service				referee		referee
standard service				silver service		silver service
any time				system		system

Figure 4. Class identification using Noun-identification technique [32]

2. *Inheritance (Super-sub relationships):* UMGAR searches each requirement for phrases like “may be”, “is a type of” between two objects. If “may be” is found between subject and object, UMGAR represents inheritance relation (direction) from object to subject, and if “is a type of” phrase occurs then inheritance is from subject to object. For this case, UMGAR identified two requirements having “may be” phrases for which inheritance relationship is depicted from object to subject, namely from employer and candidate to customer, and from standard service, silver service and gold service to service.
3. *Composition/ Aggregation among objects:* The identification of this relationship needs human intervention. However, UMGAR can identify such chances if requirement document contains phrases like comprises, have, include, possess, contains, and “is a part of” between subject and object. If “comprises, have, include, possess and contains” occurs between subject and object, then composition (strong dependency) relation exists from object to subject, and if “is a part of” phrase occurs, then aggregation (weak) relation occurs from subject to object.

Using Stanford Parser [27], each requirement is parsed to extract verbs and adjectives associated with the noun phrase which are methods and attributes of class (NP) respectively. Finally, generated analysis class model is as shown in Figure 5

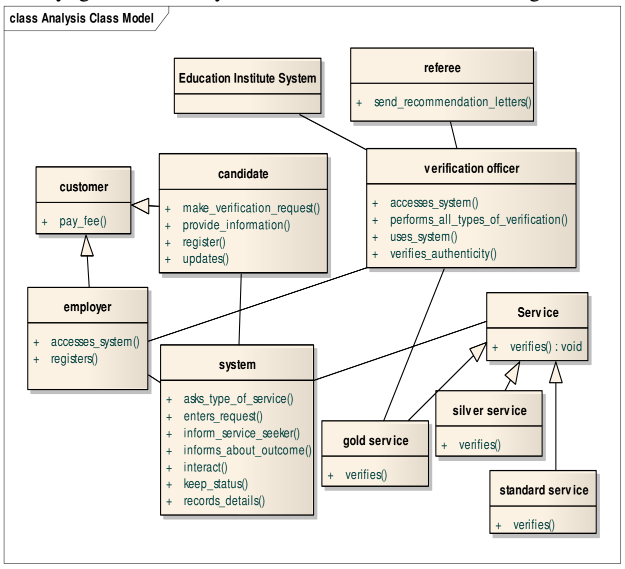


Figure 5. Analysis class diagram for QVS

LIDA [8] identifies OO artifacts based on frequency of occurrence. Later on, developer based on his/her domain knowledge has to categorize them manually into classes, attributes and methods. However, UMGAR overcomes this problem using efficient NLP tools. Table 2 shows the artifacts automatically generated by UMGAR for QVS without any human intervention.

Table 2. OO Artifacts automatically extracted using UMGAR

Classes	Attribute	Method	Association	Inheritance
11	-	23	7	5

4.3 Design Class Model Developer

This section describes the process of generating collaboration and design class model using “Perform Verification” use-case specification of “Qualification Verification System (QVS)”. Figure 6 shows the use-case execution scenario describing both basic and alternative flows of the use-case specification [33].

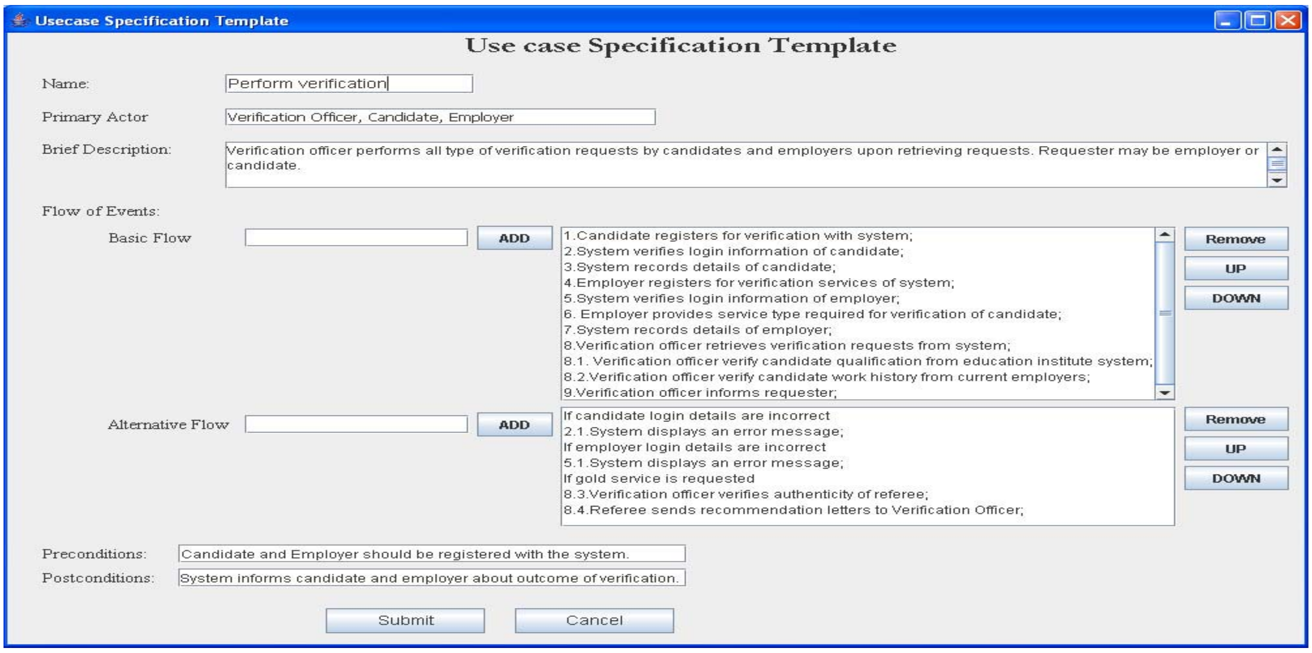
4.3.1 Collaboration Diagram Generator

UMGAR parses each use-case specification to elicit both actors and objects associated with the use-case, from which collaboration diagram is generated. Each use-case identified during use-case diagram development should be properly specified using the rule “Who do what to whom?”, which describes who initiates the message, and what it want to send and to whom according to use-case specification template. In order to perform this task, UMGAR reconstructs use-case specification into simple sentences in the form of Subject: Predicate or in Subject: Predicate: Object using previous eight syntactic reconstruction rules. For collaboration diagram generation, verb phrase (VP) structures are analyzed. Structure of an event mainly depends on the VP types as described in section 4.1.2. UMGAR handles all such possibilities to identify receiver objects and messages. Following grammatical rules are used for generating collaboration diagram:

1. Subject (NP) in the sentence is considered as sender object.
2. Object (NP) is considered as receiver object. And Predicate (VP) can also contain noun phrase which can be treated as receiver object based on the VP structures.
3. The verb phrase between subject and object is taken as message passed between objects.
4. If sentence is having subject and predicate, without any object, then sequence stated in the use-case specification helps to identify the relation between both messages.
5. Conditional statements represent sequence of statements; and can be handled by keeping If clause at the beginning of the sentence and an end_If clause at the end of the sentence.
6. Concurrent statements show sequence of actions performed at the same time, and are handled by inserting Start_ConCurrent clause at the beginning and End_Concurrent clause at the end of concurrent statements.
7. Iterative statements are handled by inserting Start_While statement at the beginning and End_While at the end of the iterative statements.
8. Synchronization statements are handled by keeping Start_Sync word after the first sentence to show the synchronous message started and after the last sentence End_ Sync word is used.

UMGAR first extracts associated actors with this use-case from the primary actor’s field in the use-case specification template. Stanford Parser [27] is used to identify objects and associated messages between those objects from the parse tree generated for each flow of event (both basic and alternative flows).

The Use-case specification described in figure 6 is simple without any conditional, concurrent, iterative or synchronization statements. As a result, only first four rules are sufficient for generating collaboration diagram from this use-case scenario. Out of 15 event flows (11 are basic flows and 4 alternative flows) having internal sub-flows, 13 events use first three rules as they are in Subject: Predicate: Object form to identify sender object, receiver object and message between the objects. Only 2.1 and 5.1 alternative flows are in Subject: Predicate form, so rule 4 is used to trace the message sequencing. For event 2 (“System verifies login information of candidate”), the parse tree generated by UMGAR using Stanford Parser is shown in Figure 7.



Use case Specification Template

Name:	Perform verification
Primary Actor	Verification Officer, Candidate, Employer
Brief Description:	Verification officer performs all type of verification requests by candidates and employers upon retrieving requests. Requester may be employer or candidate.

Flow of Events:

Basic Flow

1.Candidate registers for verification with system;
2.System verifies login information of candidate;
3.System records details of candidate;
4.Employer registers for verification services of system;
5.System verifies login information of employer;
6. Employer provides service type required for verification of candidate;
7.System records details of employer;
8.Verification officer retrieves verification requests from system;
8.1. Verification officer verify candidate qualification from education institute system;
8.2.Verification officer verify candidate work history from current employers;
9.Verification officer informs requester;

Alternative Flow

If candidate login details are incorrect
2.1.System displays an error message;
If employer login details are incorrect
5.1.System displays an error message;
If gold service is requested
8.3.Verification officer verifies authenticity of referee;
8.4.Referee sends recommendation letters to Verification Officer;

Preconditions:	Candidate and Employer should be registered with the system.
Postconditions:	System informs candidate and employer about outcome of verification.

Figure 6. Use-case Specification Template for "Perform Verification" Use-case

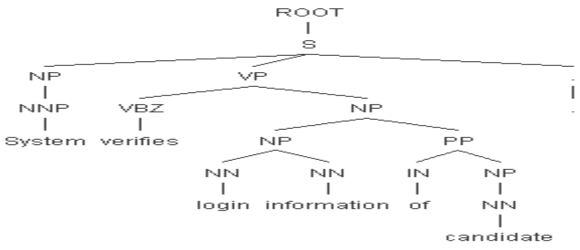


Figure 7. Parse tree for Basic Flow-event 2

According to Figure 7, the system acts as sender object, candidate as receiver object and "verifies login information" is message passed between sender and receiver objects. Figure 8 shows the collaboration diagram generated for the "Perform Verification".

4.3.2 Design Class Model Generator

Figure 9 shows the design class model generated using the collaboration diagram generated for "Perform Verification" scenario. All 8 artifacts (3 actors and 5 objects) of collaboration are considered as design classes and message associated with each actor and object is attached as methods to these classes with proper association relationships. Inheritance relationship is explicitly stated in brief description on use-case specification template, "Requester may be employer or candidate". UMGAR identifies the relationships from the event flow sequence specified in use-case specification using rules described in section 4.2.1.

4.4 Key-Word-In-Context

After generating the OO models, in order to trace requirements from models, UMGAR implements Key-Word-In-Context (KWIC) approach, to provide traceability between requirements and OO design models. This feature traces requirements associated with the keyword and display them, gives user a chance to edit and change the requirements which are reflected in entire system. Figure 10 shows the features of KWIC, when the term "Gold" a class name of QVS in the analysis class model is searched for, KWIC window shows the requirements associated with this term.

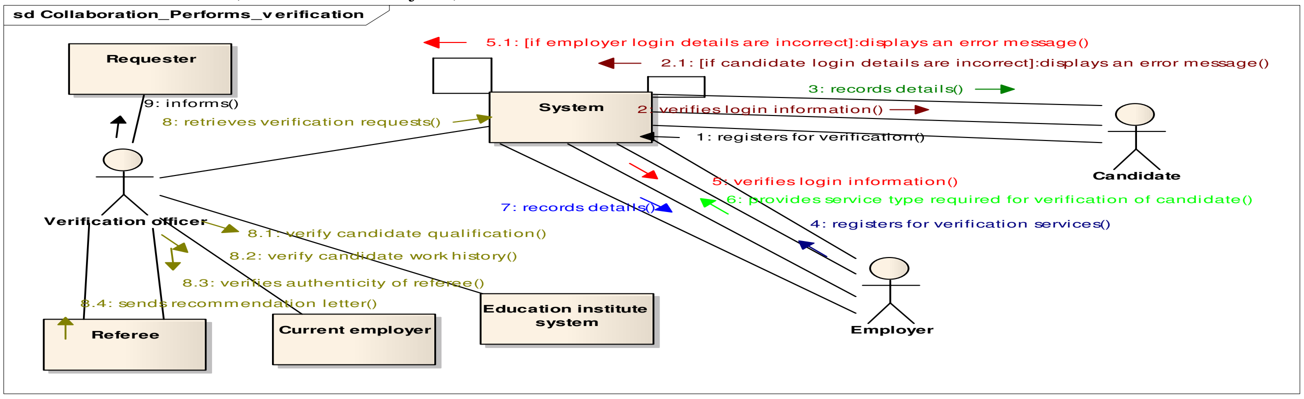

sd Collaboration_Performs_verification
Requester
5.1: [if employer login details are incorrect]:displays an error message()
2.1: [if candidate login details are incorrect]:displays an error message()
3: records details()
System
2: verifies login information()
1: registers for verification()
9: informs()
8: retrieves verification requests()
Candidate
5: verifies login information()
6: provides service type required for verification of candidate()
7: records details()
4: registers for verification services()
Verification officer
8.1: verify candidate qualification()
8.2: verify candidate work history()
8.3: verifies authenticity of referee()
8.4: sends recommendation letter()
Referee
Current employer
Education institute system
Employer

Figure 8. Collaboration diagram for Perform Verification use-case

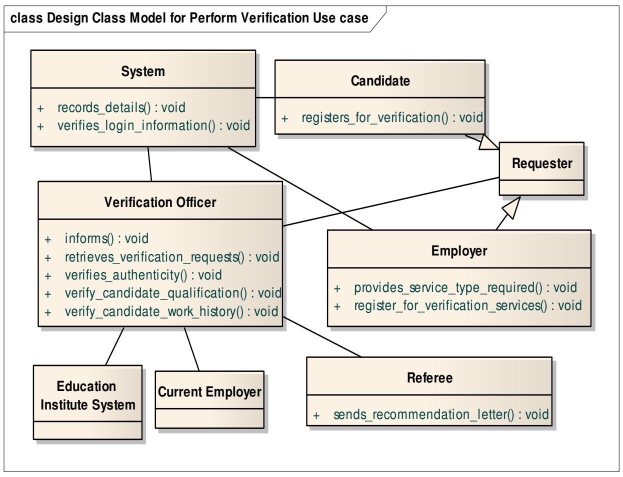
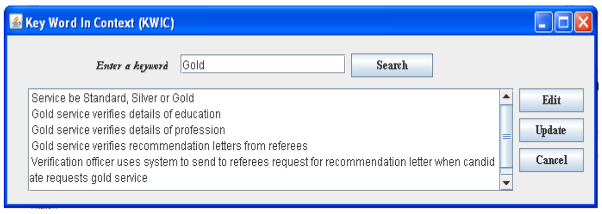

class Design Class Model for Perform Verification Use case
System
+ records_details() : void
+ verifies_login_information() : void
Candidate
+ registers_for_verification() : void
Requester
Verification Officer
+ informs() : void
+ retrieves_verification_requests() : void
+ verifies_authenticity() : void
+ verify_candidate_qualification() : void
+ verify_candidate_work_history() : void
Employer
+ provides_service_type_required() : void
+ register_for_verification_services() : void
Education Institute System
Current Employer
Referee
+ sends_recommendation_letter() : void

Figure 9. Design class model for selected use-case Scenario



```
Service be Standard, Silver or Gold
Gold service verifies details of education
Gold service verifies details of profession
Gold service verifies recommendation letters from referees
Verification officer uses system to send to referees request for recommendation letter when candid
ate requests gold service
```

Figure 10. Key Word In Context (KWIC)

4.5 XMI Parser

Previous tools developed for generating UML models from NL requirements use canvas or integrated to some modeling tool. The diagrams visualized on canvas have problems regarding editing and updating as we observed in SUGAR [9, 10]. In order to avoid such problems, UMGAR provides an efficient XMI parser to support the visualization and manipulation of the generated models in any modeling tool, which has XMI import feature. The XMI file [12] generated by UMGAR's parser follows XMI v2.1 specifications. This parser is currently supporting all models generated by UMGAR. Models can be modified after imported into modeling tool. We have successfully tested this XMI parser over Argo-UML [36] and Enterprise Architect [35] UML modeling tools.

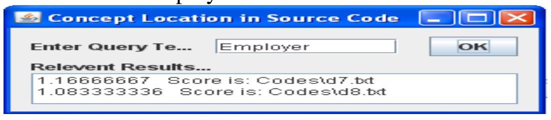
4.6 Code Generation and Concept Location

Java Code model is generated for each design class model developed using code generation feature of UML modeling tool. Even though this code model is abstract, this feature helps user to start coding as soon as requirements are gathered. Main goal for code generation is to explain the importance of concept location [30] feature in providing traceability between requirements and code by providing search functionality for a particular requirement in code. This search technique is implemented using Vector Space Model (VSM) technique [34] to identify the code file associated with a particular requirement or query. As software tends to change regularly affecting all artifacts, a developer should be able to locate code for a particular requirement. So a concept location feature is helpful in such circumstances enabling traceability from requirements to code by retrieving all possible source code files according to the relevancy of the requirement query used, making maintenance an easy way.

The VSM is traditionally used where the collection of documents are placed in term-space (consists of terms or words) and it is required to find the most relevant document for a given query. The similarity between the query and all documents in the collection is computed and the best matching documents are returned. Implementation process of concept location includes the following steps:

1. Using Stop Word methodology, all frequently occurring terms in code such as class, braces, parenthesis, visibility access levels are removed.
2. Each term in a document is represented in the form of m × n matrix, where m is the number of terms and n is the number of documents.
3. Entry a_{ij} in this matrix represents the weight of ith term in jth document.
4. For each search query, results are retrieved depending on the weight of the related term in descending order.

VSM can also be used to perform requirements similarity. Figure 11 shows the documents (indexed source code files) retrieved for term "Employer".



```
1.16666667    Score is: Codes\d7.txt
1.083333336   Score is: Codes\d8.txt
```

Figure 11. Concept location for term "Employer"

5. CONCLUSIONS AND FUTURE WORK

This paper has presented a semi-automated technique to assist developer in generating UML based analysis and design models from normalized NL requirements. Our comparative evaluation of the current tools developed for the similar objectives (described in section 2.3) has enabled us to assert that UMGAR has several advantages over them based on its features. UMGAR successfully addresses the problems caused by lack of domain knowledge by using efficient NLP tools like Stanford Parser, WordNet2.1 and JavaRAP. We have presented and discussed various features of UMGAR by using "Qualification Verification System (QVS)" case system. Our results are quite encouraging in terms of automatic identification of OO elements found from the case systems by UMGAR. UMGAR is also able to visualize UML diagrams in any UML modeling tool that has XMI import feature. Currently UMGAR requires human interaction during elimination of irrelevant classes and identification of aggregation/composition relationship among objects. UMGAR can be applied across all domains over unlimited requirements (size) expressed in NL. Future work for extending UMGAR will focus on the following issues:

1. The generated design class model lacks method signatures as it is developed from collaboration diagram. Multiplicity among classes is still to be addressed.
2. Generating state charts diagrams from use-cases to test class models without the need of generating code, so that test cases are based on requirements to test the system behavior.
3. In future, we plan to evaluate benefits and limitations of using UMGAR with industrial case studies and analyze how developers will be benefitted through these semi-automatic assisting models (with and without) in building good and robust domain model. Currently we have evaluated this technique using the case systems published in previous existing research papers [6, 8] for comparative analysis,

where UMGAR obtained better results with respect to number of OO artifacts identified.

ACKNOWLEDGEMENTS

This research was started at Indian Institute of Information Technology- Allahabad, India and later supported by the Programme for Research in Third-Level Institutions (PRTLI 4) funded by the Higher Education Authority (HEA), Republic of Ireland and in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie). First author is thankful to Dr. Muhammad Ali Babar for his useful feedback on earlier version of this paper and for providing problem context of the case study.

REFERENCES

[1] Ambriola, V. and Gervasi, V. Processing natural language requirements. *Proceedings of the 12th International conference on automated software engineering (ASE),* 36-45, (1997)
[2] Jackson, M. Problems and requirements [software development], *Proceedings of the second international symposium on requirement engineering,* pp. 2, (1995).
[3] Boyd, N. Using Natural Language in Software Development. *Journal of OO Programming 11(9),* pp. 45-55, (1999)
[4] Osborne, M. and MacNish, C. K. Processing natural language software requirement specifications. *In Proc. of 2nd IEEE Int. Conf. on Req. Engineering,* pp. 229-236, (1996)
[5] Harmain, H. M. and Gaizauskas, R. CM-Builder: an automated NL-based CASE tool. *In Proc. of the 15th IEEE Int. Conf. on Automated Software Engineering,* pp. 45-53, (2000)
[6] Kalaivani S, Dong L, Behrouz H. F and Eberlein, A. UCDA: Use Case Driven Development Assistant Tool for Class Model Generation. *In Proc. of 16th Int. Conf. on Software Engineering and Knowledge Engineering,* Banff, Canada, pp. 324-329, (2004).
[7] Mich, L. NL-OOPS: from natural language to object oriented requirements using the natural language processing system LOLITA. *Nat. Lang. Eng., 2, 2*, pp. 161-187 (1996)
[8] Overmyer, S. P., Benoit, L. and Owen, R. Conceptual modeling through linguistic analysis using LIDA. *In Proc. of the 23rd Int. Conf. of Software Engineering (ICSE),* Toronto, Canada, pp. 401–410, (2001)
[9] Deeptimahanti, D. K. and Sanyal, R. An Innovative Approach for Generating Static UML Models from Natural Language Requirements. *Springer Berlin Heidelberg, Communication in computer and Information Science, Advances in Software Engineering, Springer, Vol. 30,* page 147 (2009)
[10] Deva Kumar, D. and Sanyal, R. Static UML Model Generator from Analysis of Requirements (SUGAR). *International Conference on Advanced Software Engineering and Its Applications (ASEA 2008),* pp. 77-84 (2008)
[11] Deeptimahanti, D. K. and Babar, M. A. An Automated Tool for Generating UML Models from Natural Language Requirements. *IEEE / ACM Int. Conf. on ASE,* 2009
[12] OMG XML Metadata Interchange. *Object Management Group, MOF 2.0/XMI Mapping, v.2.1.1 (2007),* http://www.omg.org/docs/formal/07-12-02.pdf (2007)
[13] Ryan, K. The role of natural language in requirements engineering. *Proceedings of the IEEE Int. Symposium on Requirements Engineering*. San Diego, CA, pp. 240-242., (1993).
[14] Kof, L. Natural Language Processing for Requirement Engineering: Applicability to large Requirements Documents, *Requirement engineering 9(1),* pp. 40-56, (2004).
[15] Kamsties, E. and Paech, B. Taming Ambiguity in Natural Language Requirements. *In ICSSEA, Paris, Foundations of decision and computing sciences, 29 (1-2),* pp. 89-101, 2000.
[16] Berry, M D. Ambiguity in Natural Language Requirements Documents, Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs, *Lecture Notes in Computer Science, Vol. 5320/2008,* pp. 1-7, 2008
[17] Börstler, J. User-Centered Requirements Engineering in RECORD - An Overview. *Proc. of Nordic Workshop on Programming Environment Research'96,* Denmark, pp. 149-156.
[18] Nanduri, S. and Rugaber, S. Requirements validation via automated natural language parsing. *Journal of Management Information Systems 1995-96; 12(3):* pp. 9-19, 1996
[19] Popescu, D., Rugaber, S., Medvidovic, N. and Berry, D. M. Reducing Ambiguities in Requirements Specifications Via Automatically Created Object-Oriented Models. *In Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs. Springer,* pp. 103-124. 2008.
[20] Li, L. A Semi-Automatic Approach to Translating Use Cases to Sequence Diagrams. *Proc. of the Technology of Object-Oriented Languages and Systems*, pp.184, June 07-10, 1999.
[21] Montes, A., Pacheco, H., Estrada, H. and Pastor, O. Conceptual Model Generation from Requirements Model: A Natural Language Processing Approach. *LNCS. Vol. 5039* , pp. 325-326, Springer, 2008
[22] Isabel Diaz, Lidia Moreno, Inmaculada Fuentes and Pastor, O. Integrating Natural Language Techniques in OO-Methods. *Computational Linguistics and Intelligent Text Processing, LNCS, Vol. 3406,* pp. 177-188, Springer 2005.
[23] Tao Yue, Lionel C Briand and Yvan Labiche, An Automated Approach to Transform Use Cases into Activity Diagrams, *Modelling Foundations and Applications, LNCS, Volume 6138*, pp. 337-353, (2010)
[24] RAVENFLOW, http://www.ravenflow.com/
[25] Li K. Dewar R.G. and Pooley R.J. Object-Oriented Analysis Using Natural Language Processing, *in proc. of ICYCS'05*, Beijing, China, 2005
[26] Simon B., Steve M. and Farmer R. *Object-Oriented Systems Analysis and Design Using UML.* Publisher McGraw Hill, 2005.
[27] Klein, D. and Manning, C. Stanford Parser 1.6. *Stanford Natural Language Processing Group*, City, 2007.
[28] JavaRAP, last accessed 2nd December, 2010, http://aye.comp.nus.edu.sg/~qiu/NLPTools/JavaRAP.html,.
[29] WordNet 2.1, last updated http://wordnet.princeton.edu/wordnet/, 27th October, 2010
[30] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., An Information Retrieval Approach to Concept Location in Source Code, *in Proceedings 11th IEEE Working Conference on Reverse Engineering (WCRE'04),* pp. 214-223, 2004.
[31] Roberts, P. *Patterns of English.* Publisher Harcourt, Brace, and World, Inc., New York, 1956
[32] Rebecca Wirfs-Brock and McKean, *A. Object Design: Roles, Responsibilities, and Collaborations.* Addison-Wesley, 2003, ISBN 0201379430.
[33] Kruchten, P. *The Rational Unified Process An Introduction*, 3rd edition. Addison-Wesley Professional, 2003.
[34] Garcia, E. *Description, Advantages and Limitations of the Classic Vector Space Model,* 2007. http://www.miislita.com/term-vector/term-vector-3.html.
[35] Enterprise Architect 7.1, http://www.sparxsystems.com.au/
[36] ArgoUML 0.30.2, http://argouml.tigris.org/