Experimenting With Extracting Software Requirements Using NLP Approach

Yara Alkhader*, Amjad Hudaib+, Bassam Hammo◊
*Department of Computer Information Systems
University of Jordan
Email: yy_alkhader@yahoo.com
+Department of Computer Information Systems
University of Jordan
Email: ahudaib@ju.edu.jo
◊Department of Computer Information Systems
University of Jordan
Email: b.hammo@ju.edu.jo

Abstract—**Requirement engineering is a fundamental step in the production of high quality software. Many attempts have been conducted to automate some aspects of the requirements engineering process. In this paper, we present a framework that provides the requirements engineers with an environment, which accepts English natural language requirements as input and automatically generates the corresponding UML class diagram designs. Moreover the framework can highlight the possibility of specification reusability through a reverse engineering process which saves the requirements engineers both time and efforts.**

I. Introduction

Building useful software is constrained by identifying and collecting the right system requirements [12], [11]. The collected requirements are communicated between the producers and consumers of a software in order to validate customer's needs with system specifications. In that sense, the specification document which is the output of the requirements engineering activity can be considered as the contract describing system functional and non functional specifications [13].

Realizing the importance of the specification document, many of the work in requirement engineering field has been targeting the way specification documents describe their requirements [10]. Some approaches used formal languages, while others used semiformal, informal and even a hybrid of formal and informal languages [5], [4].

However, the use of the informal natural languages remains the most practical. This is due to the facts that natural languages are: inherently object oriented, powerful, expressive and widely communicated between a large scale of people. Even though natural languages appears to be the most appropriate mean to describe requirements, sever problems emerged while using them in specification documents: first of all, they might be ambiguous, inconsistent and incomplete [9], [13]. Secondly, they are never understood by computers directly without preprocessing [9]. Therefore some reinterpretations of the natural language requirements are usually conducted by the requirements engineer before proceeding with system design and development [5].

This reinterpretation is non-trivial and error prone. It needs a considerable amount of experience and it is time consuming. What makes it even implausible is the fact that requirements evolves in order to reflect real world changes. The changes in the specification document require reinterpreting the specifications into models and updating the software accordingly [8].

In this paper, we present a framework that automate the reinterpretation of natural language requirements into models and is capable to reveres engineering the models into natural language requirements allowing updates to be applied on the software.

II. Previous Work

In [5], natural language was transformed into the formal VDM++ language using XML and TLG to break the gab between formal and informal languages. Their work automated the management of formal requirements keeping them compatible with their natural language counterpart.

However, in [1], they built a requirements engineering supporting environment that analyzes and synthesizes different views given requirements written in natural languages. In their work they used a shared repository and multiple viewers and modelers to provide different interfaces for the given natural language requirements.

Where [8] automated the transformation of natural language into the semiformal UML using role post technique, which is a conceptual framework used to produce object oriented static views. In their work they first translated natural language requirement into the 4W language which is a constrained language and then used the generated set of requirements as input to their automation process.

In [9], the authors investigated the ability to determine software functionalities from software requirements

specifications expressed in natural languages. They illustrated the deficiencies and pointed the difficulties in processing natural languages. The objective of the study was to develop criteria for identifying functions. In their work they illustrated that the use of a simple method of determining functions is not productive.

A Knowledge-Based Natural Language System (KBNL) was introduced in [2]. The system presumed the existence of a model that describes the world and how language is related to the w0orld. The system parsed the English expressions analyzed them and then it converted them into the knowledge base representation.

III. Our Model

Our framework is divided into three layers as shown in Figure 1. The first layer is responsible for preprocessing natural language requirements. This layer interacts with the natural language requirements repository where natural language is stored. The second layer is the core of our framework. Our framework does most of the processing. This layer generates an XML representation for the inputted natural language requirements, which is stored in the XML requirements repository. The third layer responsibility is to generate UML diagrams out of the previously generated XML representations. Afterward the UML diagrams generated are stored in the UML class diagram repository.

The main components of our framework are as follows:

- *Natural language preprocessor*: its main responsibilities are to ensure that the inputted requirements have no spelling errors and are well structured in terms of using.
- *Natural language processor*: generates XML representation for the given requirements. In this representation, natural language tokens are annotated with meta language presenting their part of speech and their part of sentence. Figure 2 present a natural language requirement sample collected from [5], its XML representation is depicted in Figure 3.
- *Manual domain processor*: in this process, the requirements engineer uses domain knowledge to manually eliminate redundancy and resolve similarities.
- *Rule based functional analyzer*: this process represents the core of our framework. In this process, a set of rules are used to resolve ambiguity problems, which frequently occur in specification documents namely: compound names, collection of objects, pronouns, connectors and relations, in order to determine objects, attributes and operations.
- *XML schema mapper*: this process generates an XML schema for the processed XML representation. The schema represents the mapping between the XML structure and the targeted model which is the UML class diagram in our case. Figure 3 presents a snapshot of the XML schema generated for the XML requirements in Figure 4. The XML schema representation is used later on for the generation of both the class diagram view depicted in Figure 5 and the natural language view depicted in Figure 6.
- *Natural language extractor*: it uses a set of rules describing the creation of English statements. These rules are used to generate the simple natural language view depicted in Figure 6.

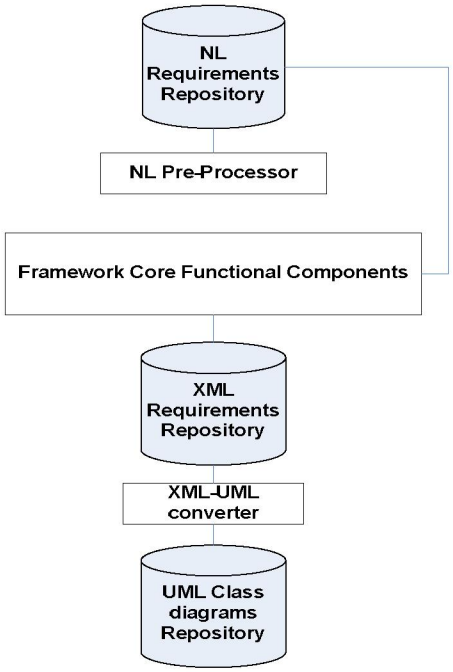

Fig. 1. The Framework high level architecture

Bank verifies ID and PIN giving the balance. ID and PIN are integers, balance is a real number.

Fig. 2. Sample natural language requirements

```
<?xml version="1.0" encoding="UTF-8"?>
<paragraph>
  <Token category="NN" POS="s" ID="0">Bank</Token>
  <Token category="NNS" POS="v" ID="1">verifies</Token>
  <Token category="NN" POS="obj" ID="2">ID</Token>
  <Token category="CC" Conn="noun" ID="3">and</Token>
  <Token category="NN" POS="obj" ID="4">PIN</Token>
  <Token category="VBG" ID="5">giving</Token>
  <Token category="NN" POS="obj" ID="6">balance</Token>
  <Token category="." ID="7">.</Token>
  <Token category="NN" POS="s" ID="8">ID</Token>
  <Token category="CC" ID="9">and</Token>
  <Token category="NN" ID="10">PIN</Token>
  <Token category="VBP" POS="v" ID="11">are</Token>
  <Token category="NN" ID="12">integers</Token>
  <Token category="." ID="13">.</Token>
  <Token category="NN" POS="s" ID="14">balance</Token>
  <Token category="VBZ" POS="v" ID="15">is</Token>
  <Token category="NN" ID="16">real  number</Token>
  <Token category="." ID="17">.</Token>
</paragraph>
```

Fig. 3. XML representation for requirements in Fig. 2

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="Bank">
  <xs:annotation>
   <xs:documentation>0, 0</xs:documentation>
  </xs:annotation>
  <xs:attribute name="ID">
   <xs:annotation>
    <xs:documentation>2, 2</xs:documentation>
   </xs:annotation>
  </xs:attribute>
  <xs:attribute name="PIN">
   <xs:annotation>
    <xs:documentation>4, 4</xs:documentation>
   </xs:annotation>
  </xs:attribute>
  <xs:attribute name="balance">
   <xs:annotation>
    <xs:documentation>6, 6</xs:documentation>
   </xs:annotation>
  </xs:attribute>
 </xs:complexType>
 <xs:element name="Bank" type="Bank">
  <xs:annotation>
   <xs:documentation>0, 0</xs:documentation>
  </xs:annotation>
 </xs:element>
 <xs:complexType name="ID">
  <xs:annotation>
   <xs:documentation>8, 8</xs:documentation>
  </xs:annotation>
  <xs:attribute name="integers">
   <xs:annotation>
    <xs:documentation>12, 12</xs:documentation>
   </xs:annotation>
  </xs:attribute>
 </xs:complexType>
 <xs:element name="ID" type="ID">
  <xs:annotation>
   <xs:documentation>8, 8</xs:documentation>
  </xs:annotation>
 </xs:element>
 <xs:complexType name="PIN">
  <xs:annotation>
   <xs:documentation>10, 10</xs:documentation>
  </xs:annotation>
  <xs:attribute name="integers">
   <xs:annotation>
    <xs:documentation>12, 12</xs:documentation>
   </xs:annotation>
  </xs:attribute>
 </xs:complexType>
 <xs:element name="PIN" type="PIN">
  <xs:annotation>
   <xs:documentation>10, 10</xs:documentation>
  </xs:annotation>
 </xs:element>
 <xs:complexType name="balance">
  <xs:annotation>
   <xs:documentation>14, 14</xs:documentation>
  </xs:annotation>
  <xs:attribute name="real  number">
   <xs:annotation>
    <xs:documentation>16, 16</xs:documentation>
   </xs:annotation>
  </xs:attribute>
 </xs:complexType>
 <xs:element name="balance" type="balance">
  <xs:annotation>
   <xs:documentation>14, 14</xs:documentation>
  </xs:annotation>
 </xs:element>
</xs:schema>
```

Fig. 4. XML schema generated for requirements in Fig. 2

In our framework, we used the MIMB tool [7] to transform the XML schema into a UML class diagram and vice versa. We also used the GATE tool [3] as our NLP infrastructure. Figure 7 illustrates the over all data flow in our framework.

IV. EXPERIMENTS

In order to test the performance of our framework, we collected a test bed of three specifications documents namely: the dinning philosopher specifications document [8], the bank system specifications document [5], and the elevator specifications document [10].

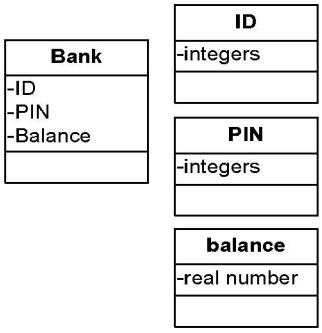

Fig. 5. Class diagram view for requirements in Fig. 2

Each Bank has a ID,
Each Bank has a PIN,
Each Bank has a balance.
Each ID has integers.
Each PIN has integers.
Each balance has a real number.

Fig. 6. Sample natural language requirements in Fig. 2

Table 1 illustrates our classification for each of the specifications document in the test bed. In addition to the previous test bed we collected a reasonable set of specifications documents written by computer science students taking a software engineering course in the University of Jordan.

We classified the specification documents in our test bed into categories depending on their language characteristics. The main factor in the classification was the structure of the sentences used. When it has one subject and one verb we consider the sentence simple. Accordingly, the specification document which is formed of simple sentences is classified as a simple one. On the other hand if a sentence contained more than one subject and multiple verbs then it is considered as an intermediate sentence, and accordingly the specification document is classified as intermediate. Whereas, sentences with more than one subject and multiple verbs are complex and documents are then classified as complex.

Our classification process is preformed manually however in future works we plan to automate this process.

In our experiments each of the documents in the test bed was inputted into our framework without any modifications in order to be able to compare our results with the results obtained in previous works [8], [5], [10].

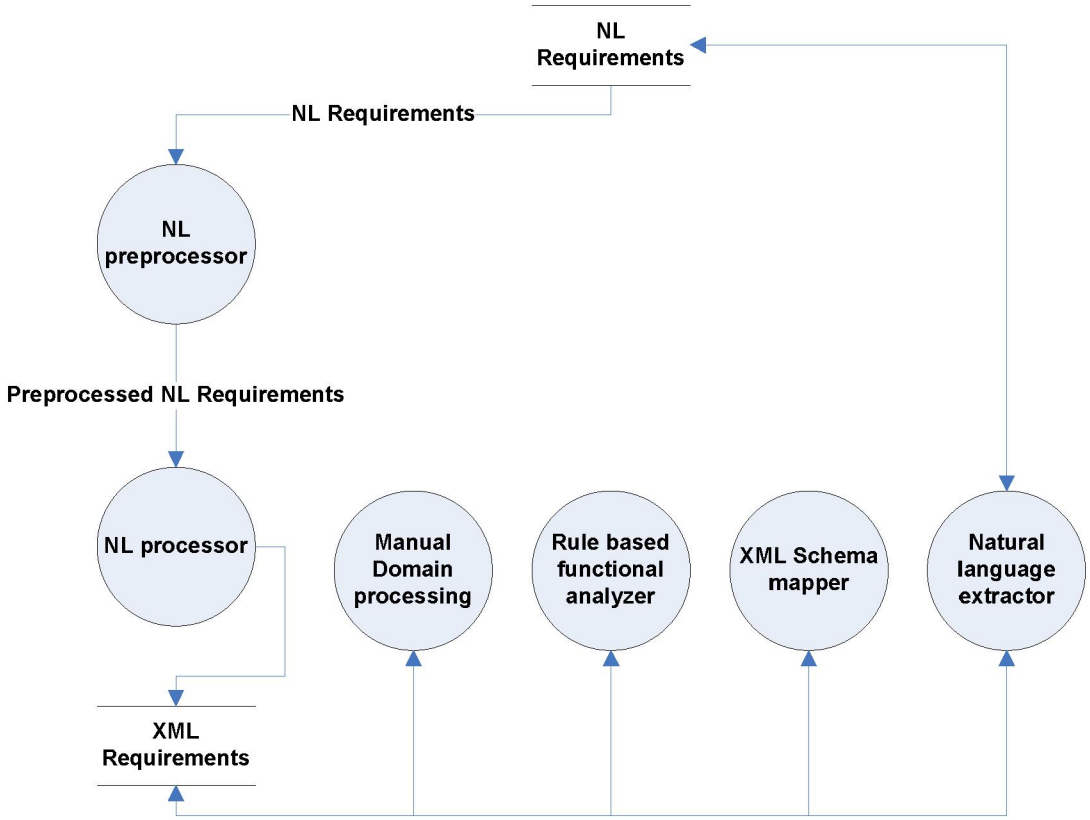

Fig. 7. Data flow diagram representing the processes and data conversion in our framework

For each document in the test bed, we generated a class diagram view and a natural language view. Afterward, we analyzed the gap between the inputted natural language, and the generated class diagram.

TABLE I
Test bed classification of specification documents

Simple	Intermediate	Complex
Dinning philosopher	Bank system	Elevator

We tested our framework using a set of specifications documents collected from computer science students taking software engineering course in the University of Jordan. The students are not English native speakers and they lack the experiences in writing specification documents. Next, the collected documents were characterized by their lack of standardization and their high ambiguity.

Documents in this set were subjected to the same classifications as in the previous set. However, those documents were tested first without any preprocessing. After that, we tested the documents after they preprocessed and the results of the two were compared.

I. RESULTS

After analyzing the results of each of the conducted experiments we concluded the following:

- Our framework achieved a higher identification percentage for both objects and attributes findings than the previous researches and thus better accuracy was achieved. Our high identification percentage was because we analyzed every token in the specification document. This is rationalized by the fact that every word listed in the specifications document should carry significant information related to the problem description or it should not be present in the document in the first place. Therefore, every token should be analyzed and processed. Table II presents a comparison of the number of objects and attributes identified in our work and in each of the previously conducted researches.
- Our framework was capable of generating a natural language view from the generated class diagram view. This feature is not available in any of the previously conducted researches. Table II presents a comparison regarding this feature as well.

- Preprocessing requirements in the sense that poorly structured requirements are converted to highly structured ones resulted in more accurate and complete output from out framework.
- Our framework identifies relations if they were expressed using a constrained language set. However, none of the inputted specifications had that feature and thus our framework was not able to identify those relations.

II. Conclusions

The results we achieved from our experiments emphasized the advantages of automation in requirements engineering. It highlighted the problems software engineers suffer while processing natural language specifications documents and how our automated framework reduces the time, the efforts and the experiences needed to model and maintain these documents. Therefore, our framework serves as a support environment for software engineers where they can: automatically elicit objects and attributes, and automatically generate and preserve a class diagram view for the given specifications document. Our framework can also serve as a tool for maintaining specifications documents where updates on specifications can be propagated to class diagrams.

III. Future Works

Majority of the future works anticipated in our works are in favor of an over all enhancement from a framework into a fully functional system that uses semantics and domain knowledge in order to analyze specifications documents. In order to do better gab analysis, we plan to enhance the system with a backward engineering process that is capable to reverse the generated the UML class models into natural language for system maintenance.
We are planning to build our own parser for natural language text and to expand our framework to work with other languages like Arabic. We are also looking forward to equip our natural language generator with domain knowledge rules to make the generated language smoother. In addition, we will automate the manual process of classifying specifications documents.

TABLE II
Comparing results between our model and the previous works

Experiment name	Comparison criteria		Our model	Previous work
Dinning Philosopher Spec. Document	Modeling			
		No. of objects identified	3	2
		No. of attributes identified	4	2
		No. of relations identified	0	1
	Overall modeling accuracy			
		In term of objects	More	Less
		In term of attributes	More	Less
		In term of relations	Less	More
	Specification maintenance		Present	Not present
Bank System Spec. Document	Modeling			
		No. of objects identified	8	3
		No. of attributes identified	29	11
		No. of relations identified	0	0
	Overall modeling accuracy			
		In term of objects	More	Less
		In term of attributes	More	Less
		In term of relations	Same	Same
	Specification maintenance		Present	Not present
Elevator Spec. Document	Modeling			
		No. of objects identified	1	1
		No. of attributes identified	12	4
		No. of relations identified	0	0
	Overall modeling accuracy			
		In term of objects	Same	Same
		In term of attributes	More	Less
		In term of relations	Same	Same
	Specification maintenance		Present	Not present

REFERENCES

[1] V. Ambriola, V. Gervasi, Environmental Support for Requirements Writing and Analysis. Information Science and Technology Institute, Pisa, Italy, 1999.

[2] J. Barnett, K. Knight, I. Mani, and E. Rich, "Knowledge and natural language processing", Communications of the ACM, vol. 33, no. 8, 1990, pp. 50-71.

[3] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, C. Ursu, M. Dimitrov, M. Dowman, N. Aswani, and I. Roberts, Developing Language Processing Components with GATE, University of Sheffield, 2005.

[4] D. Duffy, C. Macnish, J. Mcdermid, and P. Morris, "A framework for requirements analysis using automated reasoning", Proceedings of the 7th International Conference on Advanced Information Systems Engineering, Lecture Notes In Computer Science, vol. 932, 1995, pp. 68-81.
[5] B. S. Lee, Automated Conversion from a Requirements Document to an Executable Formal Specification Using Two-Level Grammar and Contextual Natural Language Processing, University of Alabama, Birmingham, 2003.
[6] B. Lee, and B. Bryant, "Applying XML technology for implementation of natural language specifications", International Journal of Computer Systems Science & Engineering, vol. 18, no. 5, 2003, pp. 279-300.
[7] Meta Integration Technology, Inc., (2006). Reference Guide. Available: http://www.metaintegration.net/Products/MIMB
[8] H. G. Perez-Gonzalez, and J. K. Kalita, "Automatically generating object models from natural language analysis", Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Conference on Object Oriented Programming Systems Languages and Applications, 2002, pp. 86-87.
[9]S. Presland, and M. A. Hennell, "Detecting functionality in natural language text", Dept. of Statistical and Computational Mathematics, Tech. Rep. 1986, Unpublished.
[10] M. Saeki, H. Horai, and H. Enomoto, "Software development process from natural language specification", Proceedings of the 11th international conference on Software engineering, International Conference on Software Engineering, 1989, pp. 64-73.
[11] I. Sommerville, Software Engineering, 7th edition, Addison Wesley, 2004.
[12] K. E. Wiegers, Software Requirements, 2nd edition, Microsoft Press, 2003.
[13] W. M. Wilson, L. H. Rosenberg and L. E. Hyatt, "Automated analysis requirement specification", Proceedings of the 19th international conference on Software engineering, International Conference on Software Engineering, 1997, pp. 161-171.