Requirements Validation via Automated Natural Language Parsing

Sastry Nanduri and Spencer Rugaber

Georgia Institute of Technology

Abstract

Object Oriented Analysis (OOA) has become a popular method for analyzing system requirements. Unfortunately however, none of the current versions of OOA have included a validation technique tailored to the object oriented approach. Most, instead, merely recommend document reviews without specifying what kinds of problems to look for. This paper explores the question by applying a natural language parser to a requirements document, extracting candidate objects, methods and associations, composing them into an object model diagram, and then comparing the results to those determined by manual OOA. To do this, we have adapted an automated natural language parser and used it to examine several high level specifications. The results indicate that with a modest amount of effort, our technique can give valuable feedback to the analyst.

1 Introduction

1.1 Background

The first step in most object oriented design methods is the construction of an object model. Many guidelines exist for identifying the object classes, their relationships, and their attributes from a problem statement. Are these guidelines comprehensive enough for automating the construction process? Probably not, but can we instead use automatic analysis to generate a version against which a manually generated model can be validated? The best way to answer these questions is by actually trying to implement a program for generating an object diagram from a specification.

There are several papers which discuss the possibility of automatic construction of the object model. Honiden et al. [6] developed a standardized, formal OOA specifications process as a precursor to automation. Seki et al. [11] describe a process for deriving incrementally a formal specification from an informal specification. Abbot [1] details a method for generating program design from informal English description. All these papers only give suggestions as to how to automate the analysis process. We could find no description in the published literature of the actual implementation of an object model constructor. The purpose of this paper is to describe one such implementation.

1.2 Object oriented analysis

Traditional approaches, like Structured Analysis, focus mainly on the functionality of a system. OOA, on the other hand, focuses on the objects or static entities of the system and the associations among them. A main reason for its popularity is the fact that a system designed around static entities is more robust and less affected by subsequent changes in the requirements than one organized functionally.

Booch [2] was the first to formalize the object oriented approach. Now there are several popular object oriented methods such as OOA by Coad and Yourdon [4], Object Oriented Design (OOD) by Booch [3] and Object Modeling Technique (OMT) by Rumbaugh et al. [10]. The methods have much in common. They all start with the detection of objects in the system by textual analysis of the specification document. After the objects are identified, the system is understood in terms of their attributes and the interactions among them. As originally proposed, nouns in the specification document are good indicators of objects. Similarly, verbs and adjectives are good signals for the associations and attributes of the objects.

The method that we used for our project is OMT because of its popularity and extensive documentation. Of the three OMT models (object, dynamic, and functional), we are only concerned with the object model as it is the most conducive to textual analysis. The approach recommended by Rumbaugh et al. for object modelling has the following steps:

1. Identify objects and classes (nouns);

2. Identify associations between objects (verb phrases);
3. Identify attributes of objects and associations (adjectives);
4. Identify operations (verbs and adjectives);
5. Organize and simplify object classes using inheritance;
6. Iterate and refine the model.

2 Approach

The approach we took to automating the analysis process is as follows. We first collected a list of guidelines on object modeling from [10]. We expressed the guidelines in terms of the parsing rules for a publicly available natural language parser. We then applied these guidelines to the parser output of several high level specifications and analyzed the results. After refining the parsing rules, we repeated the process for several other specifications documents. Finally, we implemented a program that used the refined guidelines and a public domain graph drawing tool to display an object diagram for the specification analyzed.

More precisely, the steps we followed in building our analyzer were the following.

1. We gathered guidelines for creating an object model and for identifying nouns, verbs etc. from the OMT text.
2. We used a publicly available natural language parser to parse a specification document. We chose a link grammar based parser because it was easily available, because it was able to parse a wide range of English sentences, and because its dictionary was easily extendible.
3. We wrote a rule based post-processor. These rules are nothing but the guidelines gathered in step 1 expressed in terms of the parser's links.
4. We extended our tool to accumulated knowledge between sentences. The parser we used parsed each sentence of the input independently. So, for the construction of an object model from a specification document, we had to accumulate knowledge gained from each of the sentences. As there is no foolproof way of connecting the pronouns in a sentence to the appropriate antecedent in the previous sentence, we also had to apply some empirical rules for doing this.
5. Finally, we built and displayed the object model. For displaying the object diagram graphically, we used a publicly available graph drawing tool called Edge [8]. Edge is an easy to use, extendible graph editor. Graphs can be constructed either interactively or by creating an input file consisting of a list of nodes and edges. The program we developed for step 3 took the latter course and wrote the accumulated knowledge about the objects and associations into a file in a format understandable by Edge.

2.1 Natural language parsing using link grammars

The parser we used was developed by Daniel D. Sleator and Davy Temperley at Carnegie-Mellon University [12]. This parser is based on the theory of link grammars. A link grammar consists of a set of words (the terminal symbols of the grammar), each of which has one or more linking requirements. A sequence of words is a sentence of the language defined by the grammar if there exists a way to assign links among the words so as to satisfy the following three conditions:

1. Planarity: The links do not cross;
2. Connectivity: The links suffice to connect all the words of the sequence together;
3. Satisfaction: The links satisfy the linking requirement of each word in the sequence.

The linking requirements of each word are contained in a dictionary where they are expressed as a formula involving the operators **and** and **or**, parentheses, and connector names. The **+** or **-** suffix on a connector name indicates the direction relative to the word being defined in which the matching connector must lie. For example, the linking requirements for the words "Mary" and "ran" are shown below:

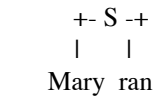
Mary: O- or S+

ran: S-

That is, "Mary" can act as a direct object if a corresponding verb is on the left or as a subject if a corresponding verb is on the right. "Ran" expects a subject on its left. The linking requirements are satisfied in the sentence "Mary ran" but not in the sentence "ran Mary". Hence, the latter is not accepted by the parser.

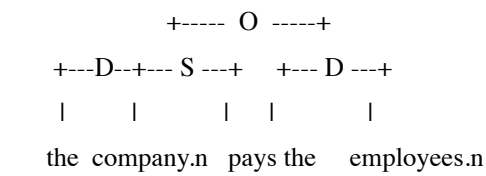
The output of the parser consists of all the words along with the links that satisfy the linking requirements. We found that the connectors used to express the linking requirements (**O** and **S** in the example above) are useful in identifying the nouns and verb phrases of a sentence and, thereby, the object classes and their associations. For

example, the output upon running the parser on the sentence "Mary ran" consists of the following connectors.



This output indicates that the word "Mary," which is on the left side of the link, has the connector **S+** and that the word "ran" has the connector **S-**. We can gather by looking at the **S+** connector that "Mary" is the subject in the above sentence. Likewise, "ran" is a verb as it has the connector **S-**.

For a better understanding of how the output of the parser can be used to identify the classes, relationships, etc., consider the following sentence "The company pays the employees". The parser's output for this sentence is:



"Company" is recognized as a noun (the ".n" suffix), "the" is a determiner, "pays" is a verb expecting a subject and a direct object, with "the employees" serving that role.

By using the guideline that any sentence of the form **Subject -S- verb -O- Object** implies that the classes Subject and Object have the association verb, we can infer that **company** and **employees** are the classes and that **pays** is an association between those two classes. Most of the guidelines for object identification can be expressed in terms of links in this way. Examples of the guidelines that we used to implement our program is given in the next section.

2.2 Object and association detection guidelines

Rumbaugh et al. [10] gives some practical tips on how to find classes, associations, and attributes in a problem statement. They also suggests how to eliminate bad classes, associations etc. Both these and the other guidelines in the literature are very general. A guideline of the form, "names that primarily describe individual objects should be restated as attributes", though very helpful for a human designer, cannot be incorporated into program logic easily. The main problem a programmer faces when he tries to incorporate such knowledge in his program is "how can the program find out which names describe individual objects?" Most of our effort in working on this project was spent on expressing the existing guidelines in terms of the connectors etc.. That is, we transformed the guidelines into precise rules in terms of the parser's output.

Examples of the rules that we used to implement our program are given below. For every word in the input sentence, it checks if any of the guidelines are satisfied. If this is the case, then the corresponding inference is made.

1. If word is "with" and, if it has **J** and **M** connectors, then the class described by the word with the **M** connector is an aggregation of the class described by the word with the **J** connector.

   Explanation: A sentence containing "building with floors..." indicates that **building** is an aggregation of **floors**.

2. For every verb, if there is an **EV** connector, get the **J** connector of the **EV** connector, if it exists. Or if there is a **V** connector, and the **V** connector has an **I** connector and the **I** connector has a **TO** connector and the **TO** connector has a **S** connector, get the final **S** connector. If either of the above mentioned connectors exist, the two words are possible classes and they have an association named by the verb.

   Explanation: A sentence of the form "A system is to be installed in a building" indicates that **system** and **building** have an association **installed**.

3. If a verb has **V** followed by **S**, get the **S** connector. Also get the **J** connector following the **EV** connector. These words are classes and they have the association verb.

   Explanation: A sentence containing "candidate is fired by the company" indicates that there is an association **fired** between **candidate** and **company**.

4. If the word is "has", then the **S** connector of the word is an aggregation of the **O** connector.

   Explanation: A sentence containing "building has floors" indicates that **building** is an aggregation of **floors**.

5. If the verb is "becomes", then the **O** connector is a state (attribute) of the **S** connector.

   Explanation: A sentence containing "person becomes candidate" indicates that **candidate** (candidacy) can be an attribute of the class **person**.

2.3 The post-processor

The post-processor that we developed applies the guidelines to the parser output and produces the list of objects, their attributes, and the associations among them. The post-processor is written in C and is integrated into the parser code. We took this approach because the number of guidelines that we had was not very large. If the number of guidelines becomes larger, it may prove advantageous to separate the post-processor code from the parser code.

The post-processor also makes use of a synonyms file while processing the output of the parser. The synonyms are used for two purposes:

- To avoid creating redundant objects for synonymous nouns in a specification document.
- To recognize plurals (e.g "companies", "company") and different tenses of verbs ("fire", "fires", "fired" etc.) as the same[1].

After all the sentences of the input document are parsed and processed, the post-processor writes the gathered information into two files *Output_file* and **graph**. *Output_file*, the name of which is specified by the user, contains the results in English. That is, it will have statements like "building is an aggregation of floors." The file **graph** has the same information written in a form understandable by Edge. As stated earlier, Edge is a graph drawing tool that can generate a graph from an input file consisting of a list of nodes and edges.

3 Example

This section describes the action of the system in analyzing the specification of a small database system for an employment agency. It is taken verbatim from [7].

3.1 The original specification

> Persons apply for positions, companies subscribe by offering positions, and companies hire candidates or fire employees. A person may apply only once, thus becoming a candidate, losing this status when hired by a company but regaining it if fired; a company may subscribe several times, the positive number of offerings being added up; finally, only persons that are currently candidates may be hired, and only by companies that have vacant positions. There are queries to check whether a person is a candidate, for finding out the company a person works for (provided that the person is not a candidate), and for finding out the number of vacant positions a company still has (provided that the company has ever subscribed). Initially, no person is a candidate and no company has subscribed.

3.2 Modified specification

The link grammar parser has difficulty parsing certain constructs requiring manual modification of the input specification. The intent is to make modifications dependent on the parser and not on domain knowledge required to understand the specification. After modification, the specification reads as follows.

> Persons apply for positions, companies subscribe by offering positions, and companies hire candidates or fire employees. A person may apply only one time. The person becomes a candidate when he applies. A person loses his status as a candidate when hired by a company. A person becomes a candidate again if he is fired by the company. A company may subscribe several times. Only persons that are still candidates can be hired by companies. Only companies that have vacant positions can hire candidates. There is a query to check if a person is a candidate. There is a query to find the company a person works for. There is a query to find out the number of vacant positions at a company. Initially, no person is a candidate and no company has subscribed.

3.3 Program output

The program generates the output shown in Table 1, where associations are indicated by triples of the form: Object Class -- Association -- Object Class; operations and attributes are indicated by stating their name and the class to which they belong; candidate subclasses and aggregations are suggested; and synonyms are placed in parentheses. Note also that the parser may produce duplicate suggestions which have been manually removed from the table below.

3.4 Object diagram

The results of running the parser are placed in a file that is then fed to the Edge graph drawing tool. For the input specification given above, the diagram shown in Figure 1 is produced. Classes are contained in rectangles; arcs denote associations. Classes contain three parts: the class name, attributes, and operations. Note that there are several ways in which the diagram could be easily improved. For example, an association that holds between one class and a second as well as between the first class and a subclass of the second could be replaced by a single association. Also, situations where an operation of a class and an association on a class have the same name could be detected and resolved.

4 Results

We applied the parser to four high-level specifications taken from the literature. These problems are quite commonly chosen as examples in various textbooks.

1. An alternative is a root word extractor such as used by the UNIX **spell** command.

.

Table 1: Parser Output for Example Specification

persons (person) -- apply -- positions
companies (company) -- hire -- candidates (candidate)
companies (company) -- fire -- employees (employee)
person -- apply -- time
candidate is an attribute value of class person
becomes (become) is an operation of class person
applies (apply) is an operation of class he (person)
person -- loses (lose) -- status
loses (lose) is an operation of class person
company -- hired (hire) -- person
candidate is an attribute value of class person
he (person) -- fired (fire) -- company
company -- subscribe -- times
still is an attribute value of class candidates (candidate)
hired (hire) is an operation of class persons (person)
vacant is an attribute value of class positions
hire is an operation of class companies (company)
Class candidate can be a subclass of class person
subscribed (subscribe) is an operation of class company

Helicopter landing:

The helicopter specification have been taken from [5]. Before running the parser on this specification, we had to rephrase some of the sentences as simple sentences. There was not much information that we could gather from the parser's output. The main problem was with the specification itself. The specification described the history of the problem rather than stating the requirements. We felt even a manual construction of object diagram is not possible from these specifications.

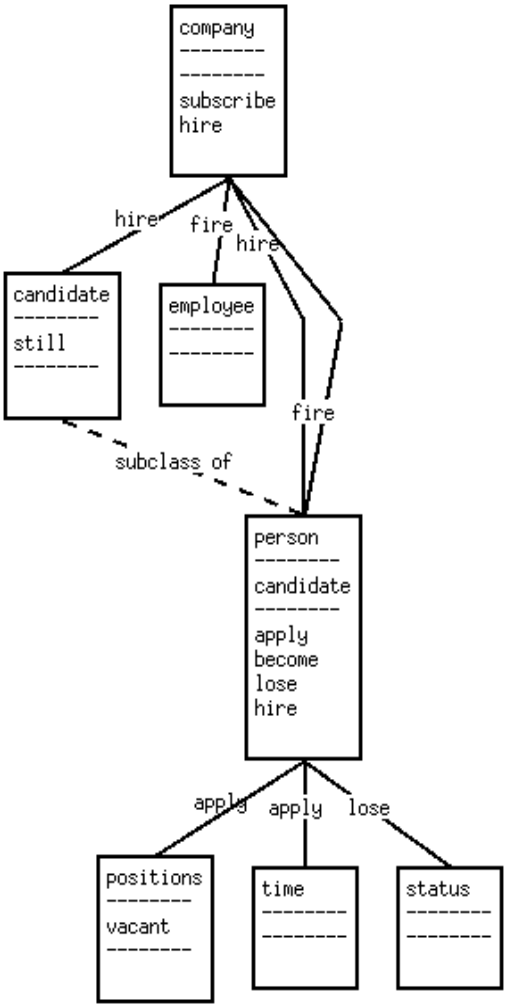
Automatic teller machine (ATM):

We gathered our second set of specifications from chapter 8 of [10]. Here, the problem was stated very clearly, probably because it was used to illustrate the construction of an object model. The results of applying our rules to these specifications were very encouraging. The resulting object diagram was reasonably close to that produced by hand. There were some differences, but these were all minor. For example, we got some extra classes like system, cost, etc. The main reason why our approach produced them was because recognizing these classes as bad classes required domain knowledge which neither the parser nor the program had.

The lift specification:

The lift specification is taken from [9]. Most of the differences in the object diagram produced from using our approach and that produced manually were due to the inadequacy of the parser in capturing some aspects of English grammar. And the rules that we used were not powerful enough to offset the parser's weakness. Here is an example of the type of problem we had. In the sentence, "Each lift has a set of buttons, one for each floor", ideally the parser should have recognized that the word "one" refers to a button. From the parser's output, we could not derive a general rule for recognizing the classes and associations correctly in a sentence of this form.

Figure 1: Object Diagram for Employment Database Specification



Employment database:

The results of applying our approach to these specifications were encouraging too. But, we realized that the rules approach to finding objects and association has some drawbacks. We found that some of design decisions cannot easily be captured through rules. For example, from the sentence, "there is a query to check if a person is a candidate", our program identified "query" as an object instead of an operation, which would be desirable if we were building a general database management system but suboptimal for a small, special-purpose program. Furthermore, because of the way the rule is defined, our program would have identified "way" as an object in the following sentence. "There is a way to check if a person is a candidate."

5 Conclusions and future work

With a relatively small amount of work (about three weeks and under 800 lines of code), we were able to build a tool capable of producing object diagrams that could be compared with those produced by hand. Among the uses of the resulting diagram would be detection of missing classes, suggestion of alternative design choices (attribute versus class or operation version association), and discovery of missing associations.

However, as described in the previous section, the object diagrams generated by our approach were not completely satisfactory. The reasons for the failure in producing completely acceptable object diagrams are the following.

- Parser inadequacy. While the breadth of English text that the parser accepts is quite impressive, it still cannot handle many sentences. For example, the parser does not accept hyphenated words, idiomatic expressions, and quotation marks. And it cannot connect a pronoun with the corresponding noun. We had overcome these problems to some extent by rephrasing the sentences in the specifications in a form acceptable to the parser. But, for the process to be completely automated the parser should be powerful enough to accept all types of sentences.
- Ambiguous or incomplete specifications. Sentences of the form "ATM accepts cash cards" can be difficult to deal with when the reader is a computer. Where does the ATM accept the cash card from? While an intelligent human understands this from the context, it is very difficult for a program to do the same.
- Lack of domain knowledge. In the ATM specification document a lot of domain knowledge was required to generate the object diagram. Knowledge of the type "bank holds account", "customers have cash cards" was assumed and not explicitly mentioned in the specification. Any practical design automator will need domain knowledge and common sense.

  There are two approaches to the solution of this problem. We can incorporate the domain knowledge in the parser. Or we can write the specifications without assuming any domain knowledge on the part of the user. Both the approaches have practical difficulties.
- Inadequacy of guidelines. Most of the rules we derived work for general cases. But, they cannot handle special cases. For example, consider the following sentence from the placement office specifications. "There is a query to check if a person is a candidate." We can make **query** a class or an operation of the class **person**. A human needs to look at the overall structure of the object diagram and use his or her experience to decide

whether to make query an object or an operation. As all our rules are based on the structure of the sentence, such decisions cannot be made by the program.

5.1 Future work

As noted earlier, a program that will automatically produce a perfect object diagram from the specifications document is still a dream. But our program can be used to validate an object diagram given the specification document and to generate a preliminary object diagram that can be refined by a human designer. The following enhancements suggest themselves as future directions for our research:

- Testing the tool on real-life (lower) level documents, particularly those with a specialized domain vocabulary.
- Comparing the results of our semi-automatic validation with that produced in a design review, both for thoroughness and cost-effectiveness.
- Testing the program with a wide range of specification documents and refining the guidelines extensively.
- Using a parser other than the link grammar parser to see if some of the parsing limitations can be overcome.

Acknowledgment

The authors wish to thank BNR, Inc. for their gift in support of this research.

References

[1] Russell J. Abbott. Program Design by Informal English Descriptions. Communications of the ACM, Volume 12, Number 11, November 1983, Pages 882-894.

[2] Grady Booch. Object-Oriented Development.*IEEE Transactions on Software Engineering,* Volume 12, Number 2, February 1986.

[3] Grady Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings Publishing., 1991.

[4] Peter Coad and Edward Yourdon. *Object-Oriented Analysis / Second Edition*. Yourdon Press,1991.

[5] Alan M. Davis. S*oftware Requirements / Revision / Objects, Functions, & States*. Prentice Hall.

[6] Shinchi Honiden, Nobuto Kotaka Yoshinori Kishimoto. Formalizing Specification Modeling in OOA. IEEE Software, Volume 10, Number 1, January 1993, Pages 54-66.

[7] M. I. Jackson. Developing Ada Programs Using the Vienna Development Method (VDM). *Software-Practice and Experience*, John Wiley & Sons, Volume 15, Number 3, Pages 305-318, March 1985.

[8] Frances Newbery Paulish and Walter Tichy. EDGE: An Extendible Graph Editor. *Software-Practice and Experience*, Volume 20, Number S1, June 1990, Pages 63-88.

[9] *Proceedings of the Fourth International Workshop on Software Specification and Design*. April 3-4, 1987, Monterey, California, IEEE Computer Society, 1993.

[10] James Rumbaugh, Michael Blaha, William Premerlani, Frederic Eddy, William Lorensen. Object-oriented Modeling and Design. Prentice Hall, 1991.

[11] M. Seki, H. Horai, H. Enomoto. Software Development Process from Natural Language Specification. 11th International Conference on Software Engineering, May 1989.

[12] Daniel D. Sleator, Davy Temperley. Parsing English with a Link Grammar, Carnegie-Mellon University, Department of Computer Science, March 1992.