NLARE, A Natural Language Processing Tool for Automatic Requirements Evaluation

Carlos Huertas
University of Baja California
Calzada Universidad 14418
Tijuana, BC. Mexico
+52 664 199 27 28
chuertas@uabc.edu.mx

Reyes Juárez-Ramírez
University of Baja California
Calzada Universidad 14418
Tijuana, BC. Mexico
+52 664 302 48 78
reyesjua@uabc.edu.mx

ABSTRACT
Most research works have found that an important root cause of software project failure comes from the requirements; their quality has an important impact over other artifacts. As the requirements are expressed in natural language, they can be an important source of defects. Aspects such as non ambiguity, completeness, and atomicity can be affected due the characteristics of natural language. Traditional practices focus on finding software bugs, as a corrective approach, until the project has been coded already, instead assuring quality since the beginning. By other hand, evaluating such quality attributes can be a difficult task. In this paper we propose some guidelines for a disciplined sentence structure for expressing the requirements, which allows natural language processing techniques to evaluate quality. We also propose a tool for automatic requirement evaluation based on the grammar structure of sentences expressed in natural language. With this tool we have a huge speed increase over manual evaluation. In order to validate our proposal we have implemented a set of experiments with real projects, assessing the impact of requirements quality over project results.

Categories and Subject Descriptors
I.2.7 [**Natural Language Processing**]: Text analysis
D.2.1 [**Requirements/Specifications**]: Tools

General Terms
Experimentation

Keywords
Natural language processing, software engineering, software bugs, software quality, software requirements.

CUBE 2012, September 3–5, 2012, Pune, Maharashtra, India. Copyright 2012 ACM 978-1-4503-1185-4/12/09…$10.00.

1. INTRODUCTION
Software requirements are a key element of any software development as they state goals, functions and constraints about the project. Requirements are usually defined at very early stages of development and therefore the impact of good and bad requirements will be reflected all over the project as next stages use vital information from requirements specifications [1].

Most researchers have found that wrong requirements are an important cause for projects failure [2], [3]. One significant cause is that requirements are written in natural language. This technique is used since early times of software engineering. By the time of this paper, natural language is still the most used way to write requirements [4], therefore, all natural language problems are inherent to requirements [5]. The study of all of these potential problems is out of the scope of this paper, so we focus on three of them, which are: lexical ambiguity, conjugation and incompleteness. Formal inspections [6] has been developed to find this problems before they turn into problems during production, were fixing them is much more expensive [7].

Some requirements that are not written in natural language are developed with formal languages like UML [8]. This greatly reduce problems in requirements since formal languages does not inherits the major problems of natural language, however this kind of techniques still have a low level of acceptance due the other problems being added, for example, the effort and skills necessary to understand a formal language. This limits the amount of people that can understand the document; this is one of the main causes we still see natural language as the most used way for requirements [4]. Even when the requirements are expressed in formal languages such as UML, there are very few chances to avoid deal with natural language, therefore this paper focus on natural language processing to help in requirements evaluation.

In the case of requirements expressed in natural language, the main problem is not the need of manually checking requirements but the amount of time and effort that is needed in the process when the amount of requirements exceeds the human resource capacity. An automatic evaluation can reduce this effort and over time increase the detection effectiveness.

In this paper, we propose a solution consisting of the development of a natural language processing tool called NLARE, which use a set of defined rules as well as regular expressions to look for problems on requirements specifications. This is done automatically and the results are expected to be a support for requirements engineer to try to reduce the potential problems that could be inherited to next stages. Thanks to the performance provided by the regular expression parser in NLARE, it is possible to have requirements in constant evaluation as it provides a good speed to keep up with the demand of quickly having running system by current standards [11]. Usually, software developers prefer a visual aid while developing software [9], this is a reason to use UML, and however the use cases descriptions are still based on natural language and are susceptible to be incorrect.

It is very important to notice that the tool is not intended for engineer replacement but instead to help reduce the error prone situation in requirements engineering. Furthermore, the NLARE approach is not intended to be a substitute for UML but is expected to have a positive impact since the modeling could be based on better requirements.

The development of this kind of tool is still limited compared with other stages of development, and the requirement elicitation demand new tools too [10], since the requirements stage is not linear anymore as most modern software process are iterative and incremental which increase the difficult of this process.

This paper is structured as follows. Section 2 contains a resume of related works that try to solve current issues by different approaches. In section 3, we show a formal definition of the problem. In section 4, the overall architecture for NLARE is discussed, while in section 5 we discuss the natural language processing involved in NLARE development. Section 6 provides information about the main experiment while in section 7 we provide our test results. Potential threats can be found on section 8. In sections 9 and 10 we discuss our conclusions and plans for future NLARE versions. Finally on section 11, all references are detailed.

2. RELATED WORK

After our research we found very limited amount of works for automated review of requirements expressed in natural language, so far related researches found all comes from NASA.

Among works found, we have the keyword matching based SAVVAS (Software Automated Verification and Validation and Analysis System) Front End Processor [12] and Information Retrieval (IR) based approach [13], [14]. The IR methods try to help for the traceable issue in requirements by finding common terms in both low-level and high-level requirements; this approach has shown improvement over manual evaluation in small projects [4]. The main limitation on this approach is that it can only be used within requirements of the same type and it use is not intended to identify requirements of a specific type.

Another approach is the ARM Tool (Automated Requirements Measurement), which aim is to measure requirements quality [15]. This tools checks for quality indicator previously defined by The Software Assurance Technology Center (SATC). This tool focuses on ambiguity, which is one of the problems we try to solve with our approach. A good attribute of this tool is that it has the knowledge of the SATC and the quality attributes defined by them, however that can be a disadvantage to different projects that does not share the same attributes.

Among others NASA works there is the PROPEL (Property Elucidator), which its main goal is the accurate and complete specification of system properties in natural language [16]. This tools aims to fight ambiguity, but instead of identify this problem this tool works preventing the generation of this ambiguous requirements [17]. Also this tool provides templates to capture often unconsidered details for property patterns, however the way PROPEL shows specification to users is restricted English sentence and graphical finite-state automata (FSA).

The last approach from NASA to be discussed here is one being developed using Linear Temporal Logic (LTL), which use a machine learning and natural language processing to identify 8 different patterns in temporal requirements [18], this however does not deal with ambiguity and has the disadvantage of only works with temporal requirements.

Finally there is another approach being developed by Sunny Hills Consultancy BV, a commercial company based on Netherlands, they have developed a tool called Requirements Assistant (RA) which according to their website [19] it contains the experience of more than 500 requirements reviews in a knowledge based system with the goal of identify ambiguity, inconsistency and incompleteness from requirements. The main problem we can identify in this approach is that it requires a requirements training data; therefore it becomes language dependent.

3. PROBLEM DEFINITION

A software functional requirement, when is expressed in natural language, is a statement that provides information about a required functionality to be included on the system [1].

In a formal way we can define a requirement as a set R composed by a finite number of words, therefore we have:

$$R = \{W_1, \ldots, W_n\} = W$$

The set W, as its going to be used under this research can be represented as a tuple composed by two elements: Word-type t, and a set of meanings M, which represents the different available meanings for the given word W_i, given this we have that:

$$W_i = \{t, \{M\}\}$$

With this, now we can represent R as follows:

$R = \{\{t, \{M\}\}_1, \ldots, \{t, \{M\}\}_n\}$.

The problems that are going to be addressed are: *Ambiguity*, *lack of atomicity*, and *incompleteness*.

Ambiguity: Under this research, we consider that the ambiguity B is generated when the number of elements in the set M for a W_i is greater than 1. This is expressed as follows:

$B = \{W_i (M)\} > 1$

Lack of atomicity: In this research, the element *t* can be a verb, noun, pronoun, conjunction, adverb, etc. When *t* is a conjunction between two sets of words, it causes a lack of atomicity in a requirement. Indicated as C, this is expressed as follows:

$C = \{W_i (t) \text{ -> conjunction}\} > 0$

Incompleteness: In order to validate the completeness, we propose that a requirement must have the following elements: Actor A, Function F and Detail.

We consider that A, F and D are all composed by a finite number words. Based on this, we extend the definition of the set R, now it must be seen as a composition of three sets, then we have:

$R = \{\{A\}, \{F\}, \{D\}\}$

In this research, a requirement is considered as incomplete if any of the following conditions are not met:

Condition #1: $\{A \subset W\} \neq \varnothing$

Condition #2: $\{F \subset W\} \neq \varnothing$

Condition #3: $\{D \subset W\} \neq \varnothing$

In the following sections we explain how these formal validations are possible with Natural Language processing. Also, we explain how the proposed solution has been designed.

4. NLARE ARCHITECTURE

In this section we describe our proposed tool called "Natural Language Automatic Requirement Evaluator" (NLARE), which has been developed as part of a master computer science degree.

There is different kind of requirements as well as different problem to solve [20-24]. The NLARE tool focus on functional requirements and aims to solve three of the main problems suffered by requirements that are likely to cause problems [24]: ambiguity, incompleteness, and atomicity. Next we describe these attributes in the meaning in which we analyze them for NLARE tool implementation.

Ambiguity: is produced when the requirement's text itself can be interpreted with different meanings for different readers.

Incompleteness: occurs when a requirement does not provide the needed information. This can happen mostly because the writer assumes that the reader is able to deduce missing information or only because there was not a grammar structure that tells that something is missing.

Atomicity: the requirement's text must address only one thing. This is a property of good requirements that is often overlooked; a requirement could be unambiguous and include all the needed information but still have problems if it addresses more than one thing, since that may produce partially verifiable requirements that should be avoided.

Finding a problem or bug at other stages in development cycle can be done, however this is a much more expensive [7] approach since a minor defect that started as a simple sentence in a requirement has been translated into a bigger and more complex problem to solve.

The NLARE tool use natural language processing to look grammatical patterns that could lead to defects, this is done using different technologies as the NLTK (Natural Language ToolKit) [26] for all processing and PyEnchant library [27] as high-level access to spell checking Enchant library [28].

To further understand of how NLARE architecture is designed we propose the following image that represents the five different modules that construct the main functionality.

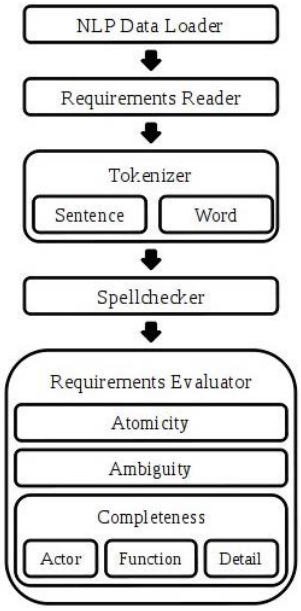

Figure 1. NLARE tool architecture.

4.1 NLP Data Loader

This module is responsible for looking and loading all the needed data for the processing. This data is composed by chunkers, corpora, grammars, taggers and tokenizers to evaluate requirement specifications. By the time of this paper data collection is over 400 MB.

4.2 Requirements Reader

NLARE is a command line tool only. This module automatically looks for requirements in the current directory, by default it will look for a plain text file called req.txt with UTF-8 character encoding which are then loaded into memory for faster processing.

4.3 Tokenizer

Here is where processing begins, in this case, there is two sub-modules which compute different tokenization.

4.3.1 Sentence Tokenizer

This is done to make sure each requirement is composed by only one sentence otherwise it will not comply with the atomic quality attribute. This checking is performed using the Punkt algorithm.

4.3.2 Word Tokenizer

In this step all requirements are divided on a substring array where each element is a word or symbol to begin further processing

4.4 Spellchecker

Once tokenizer leaves each requirement in a suitable way this module is responsible to check that each word be a correctly spelled English word otherwise notify the results engine, this verify is being done by accessing the PyEnchant library.

4.5 Requirements Evaluator

This is the core module in NLARE tool, and it is composed by three sub-modules each of them responsible for one quality attribute:

4.5.1 Atomicity sub-module

As it was explained before Tokenizer module checked for single sentence requirements but what this module does is check if the resultant sentence could be split into smaller sentences. This is possible when conjunctions are included in a requirement and therefore does not comply with atomic quality attribute.

4.5.2 Ambiguity sub-module

There are certain types of words that inject ambiguity into requirements. In this step the module is responsible of find this type of words which are foreign words, adjectives (comparative and superlative) and superlative adverb. If this type of words are found this is informed to results engine module.

4.5.3 Completeness sub-module

Here is where more processing is been done, as a regular expression parser try to find three key elements that are expected on functional requirements. These elements are:

4.5.3.1 Actor

It is important to delegate "who" will execute the functionality expressed in the requirement.

4.5.3.2 Function:

An obvious expected element in a functional requirement is indeed a function to perform, which is translated to "what".

4.5.3.3 Detail

Additional detail to complete conditionals for requirement functionality, this can be seen as "where" and/or "when".

This functional requirement elements are based on journalism theory of Six Ws[29] which states the needed information for a complete understand of an event. As it can be noticed by our approach only 4/6 of the Six Ws are accomplish, because the remaining "Why" and "How" are information that belongs to a different document and not to requirement itself.

Now that NLARE architecture has been defined, in next section we proceed to explain in detail the natural language processing involved in Evaluator Module.

5. PROCESING NATURAL LANGUAGE IN NLARE

Processing is divided on 3 main modules responsible for NLP and 2 additional modules for output formatting and results statistics.

For NLP we have the following modules explained below as well as an image with the main processes involved.

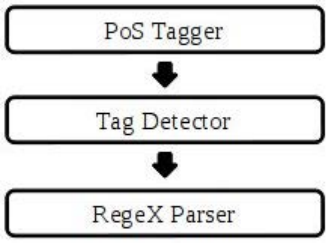

Figure 2. NLARE processing steps

5.1 PoS Tagger

This is the first step for evaluation. Using the Maxent POS Tagger [30] we are able to apply a part-of-speech (PoS) tag to each word in requirements and as result we get a new string of the original requirement with the PoS tag appended to each word. Main idea of this tagging is to work with a known set of tokens that are language independent, which means on further works it can be easily extended to other languages than English.

Example:

Input: Functional Requirement

FR1: The system must detect user country of origin by IP address

Output: Tagged String

FR1: [('The', 'DT'), ('system', 'NN'), ('must', 'MD'), ('detect', 'VB'), ('user', 'NN'), ('country', 'NN'), ('of', 'IN'), ('origin', 'NN'), ('by', 'IN'), ('IP', 'NNP'), ('address', 'NN')]

The tags meanings used for the previous example is as follows: DT: Determiner; NN: Noun, singular or mass; MD: Modal; VB: Verb, base form; IN: Preposition or subordinating conjunction; NNP: Proper noun, singular. A full list of possible tags and meanings can be found on Penn Treebank II [31].

The PoS tagger output is sent to next module for further checking looking for potentially ambiguous word as well as conjunctions.

5.2 Tag Detector

In fight for atomicity a tokenizer was used (Section 3.3) however there is still chances to split requirements in shorter sentences if conjunctions are found, in this case the tag detector looks for Coordinating Conjunctions which were tagged as 'CC' (Section 4.1).

If any of this CC tags are found, the requirement is tagged as "Not Atomic" as potential problem.

After manual evaluation of requirements it was found there are certain kind of words which are ambiguous by nature and this ones are being looked for ambiguity problems, this kind of words are Foreign word (tagged as FW), Adjective (JJ); Adjective Comparative (JJR); Adjective Superlative (JJS); Adverb (RB) and Adverb Superlative (RBS)

For completeness evaluation, the processing cannot be done at tag level, instead the whole sentences needs to be processed for patterns that could meet the requirement structure proposed on Section 3.5.3. This evaluation requires a more complex approach which is described on next module.

5.3 RegeX Parser

This is a grammar based chunk parser that uses a set of regular expression patterns to specify its behavior. The chunking of the text is encoded using a ChunkString, and each rule acts by modifying the chunking in the ChunkString. The rules are all implemented using regular expression matching and substitution.

A grammar contains one or more clauses in the following form:

```
NP:
{<DT|JJ>}          # chunk determiners and adjectives
}<[\.VI].*> {      # chunk any tag beginning with V, I, or .
<.*>}{<DT>         # split a chunk at a determiner
```

The patterns of a clause are executed in order. An earlier pattern may introduce a chunk boundary that prevents a later pattern from executing. Sometimes an individual pattern will match on multiple, overlapping extents of the input. As with regular expression substitution more generally, the chunker will identify the first match possible, then continue looking for matches after this one has ended.

The clauses of a grammar are also executed in order. A cascaded chunk parser is one having more than one clause. The maximum depth of a parse tree created by this chunk parser is the same as the number of clauses in the grammar [32].

With this approach the RegeX Parser based on defined grammar looks for patterns to find if requirement structure meets the required elements defined on Section 3.5.3.

If at least one element cannot be found the requirement is tagged as incomplete.

Finally all collected flags and processing results for this and previous tests are sent to next module for output formatting in human readable structure

5.4 Details Constructor

While previous processing modules do its job a series of flags are created given the processing results for each step, in this module that flags are categorized and processed to generate a human readable output string with information that may help reviewer to understand which potential problems are found by NLARE.

This modular architecture is designed to easily language extension, as same processing modules can be used and only switch Details Constructor to a different language output.

Output examples are as follows:

R1: The requirement is not atomic because it contains the word 'and'

R2: The requirement is ambiguous because it contains the word 'earlier' and 'later'

R3: The requirement is incomplete because No verifiable functionality was found

5.5 Statistics Calculator

This is final step for NLARE runtime and it provides statistics about the finished processing for easy results overview.

This information includes 3 benchmarks:

Runtime: measured time since application start until finish execution

Loading Time: time required to load all NLP data (Section 3.1).

Average Speed: average processing speed measured in evaluated requirements per second (req/s)

Finally an overview of categorized results by Total Good Requirements, Not Atomic Requirements, Ambiguous Requirements and Incomplete Requirements as well as a labeled notation to identify requirements by FR<i> where i represents the line where requirement can be found for further checking.

After this output NLARE execution ends and it is expected than human evaluation check potential problems suggested by this tool and look proper fixing to run checking cycle again.

6. EXPERIMENT DESIGN

As stated on previous section on this paper (Section 1), natural language requirements are still the most used way for requirements specification. With the lack of automatic evaluation, these requirements are evaluated by humans and therefore we set our experiment benchmark to how close is NLARE opinion about a requirement compared with human evaluation.

The experiments begin with a regular requirement elicitation and creation of a requirement specification document. Developers then are asked to read requirements and tag each of requirement with different attributes like not atomic, ambiguous or incomplete, this same requirements are later evaluated with NLARE tool to create different comparison scenarios are explained in next section.

7. TEST AND RESULTS

As it has been vastly discussed [1-4], bad requirements cause significant impact over software quality, to test NLARE capabilities we have used a project from a telemetry company which name needs to rename private.

Project overview is about creating a new hardware device for tracking vehicle position based on GPS signal with data download via WIFI, testing was done by running NLARE tool but allowing the continue of development cycle to determine if problems arise due to bad requirements and if this problems found on requirements were caught by NLARE tool at checking stage.

When a problem was found it was tracked down to find source of error after this analysis it was found that at least 69.3% of errors could be tracked to requirements problems as main source, this defects distribution is shown in the next table.

Table 1. Requirements defect distribution

Defect	Occurrence
Not Atomic	21.25 %
Ambiguous	41.25 %
Incomplete	26.25%

It is important to notice than in previous table the percentage sum which equals 88.75% does not means all that requirements were wrong as one requirement may suffer for more than one defect and therefore increase the occurrence percentage for all that defects.

After it was discovered that a requirement was root cause for defect it was fixed and tagged with the suffered defect, now our research interest is to compare human manual evaluation with NLARE to see how close it is and how many defects would be saved by proper requirement specifications, this concordance in results is shown in Figure 3.

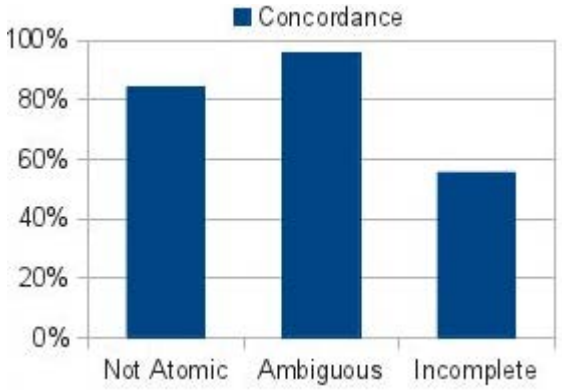

Figure 3. Concordance Human-NLARE evaluation

As it can be seen from Figure 3, NLARE tool got very good concordance with human evaluation for Atomicity and Ambiguity, where it scored 84% for atomicity and as much as 96% for ambiguity, there is still research to be done to find out if differences are entirely NLARE fault or human error.

The results for completeness as current NLARE version 1.0 has not been completely successful as it scored a relative low 56%, after differences analysis it was found they are due NLARE lack of common sense in text evaluation which usually expect more detail than provided, however further research is in mind to determine if more flexible requirement structure can be developed for better analysis or the missing information required by NLARE was indeed important even when it may look obvious if evaluated by common sense.

The next benchmark in research interest was to find out how much faster the NLARE checking could be done as compared by manual evaluation, this is important as it was mention before, big projects with lots of requirements make this evaluation a very time consuming task with lots of are for improvements.

While requirements were being evaluated by manual approach we record the average time it takes to evaluate and determine defect for requirements, this is with the assumption that humans would not get tired and could keep that average speed over N-number of requirements.

After calculations it was found reviewers got an average speed of 0.9 requirements per minute while NLARE running on an AMD P820 processor with Linux kernel 3.0.0-16 and python 2.7.2+ clocked on average 237.6 requirements per minute as can be seen in Figure 4.

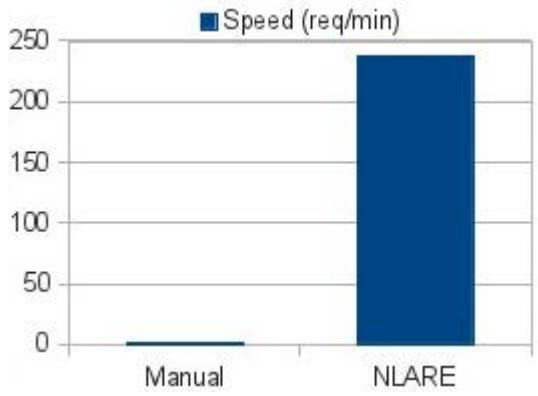

Figure 4. Speed comparison Human vs NLARE

As it can be seen from Figure 4, the speed increase over manual is very good and would help to reduce considerable amount of time in projects with a large set of requirements.

8. THREATS TO VALIDITY

As with any research there is always instance where the validity may be compromised, in this paper we have found the most common potential threat would be regarding the Ambiguity checking.

The main reason this threat could happen is because there is different kind of ambiguities, therefore NLARE tool could tag a requirement as perfectly unambiguous when it is not, however as it was clearly specified, by this 1.0 version our tool only works with Lexical ambiguity which means that Syntactic or Semantic ambiguity are not being evaluated.

It is important to notice that when NLARE tags a requirement as correct it does not mean the requirement is perfect, but at least in the performed tests it successfully pass all of them.

Another key fact is that NLARE is not intended for human substitution but instead as a tool to improve quality in requirements by helping requirement engineers to focus efforts where more are needed and reduce the error prone results that even the most experience engineer can suffer.

9. CONCLUSIONS

In this paper we have presented NLARE v1.0 which sets the beginning of a step forward the seek for a systematic and quantifiable approach in requirements quality.

Even when the tool goal would always be 100% accurate results, we think the current concordance ratio with human evaluation sets the tool in an stable release and could be used as a support tool for requirement engineers to find potential errors than even the most experienced engineers are prone to commit.

There is of course the possibility that NLARE perform some mistake in evaluation however as it is not intended for human substitution in worst case scenario it would throw a false alarm but since no requirements are being modified by NLARE, no bad impact could be performed by this tool on further steps of development.

At the time of this paper we could not find a similar free available tool for comparison as most of them are being developed by NASA and for their exclusive usage.

As with any single research there is still areas to improve, but our efforts continue in NLARE development to make a trust application for requirement specifications.

10. FUTURE WORK

By the time of this paper, NLARE 1.0 works only as a detection tool for certain quality attributes and for the case of Ambiguity only Lexical ambiguity is checked, therefore our current expectations for NLARE 2.0 involves include of more ambiguity types as well as include other quality attributes like Unitary and Consistent.

Another planned approach for future version of NLARE involve the development of an expert system where NLARE could not only detect problems but given a knowledge base propose a solution which we think will greatly impact the way requirements are evaluated and fixed as by current version requirement fix needs to be done manually and testing performed again to evaluate fixes.

On lower priority there is plans to provide a GUI to improve tool usability and generate graphic plots automatically given processing results.

11. REFERENCES

[1] Phillip A. Laplante, “What Every Engineer Should Know about Software Engineering”, CRC Press, 2007.

[2] R. Chillarege, W. L. Kao, and R. G. Condit, "Defect Type and its Impact on the Growth Curve," in the 13th IEEE International Conference on Software Engineering, Austin, Texas, May 13-17, 1991.

[3] Reyes Juárez-Ramírez, "Towards improving user interfaces: a proposal for integrating functionality and usability since early phases", IEEE, Indonesia, 2011.

[4] Alen P Nikora, Experiments in Automated Identification of Ambiguous Natural-Language Requirements, 2011

[5] D. Berry and E. Kamsties, "The sintactically dangerous all and plural in specification", IEEE Software, 2005, pp. 55-57.

[6] F. Shull and C. Seaman, “Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion,” IEEE Software. 25, 1 (Jan. 2008), 88-90. Gerard Holzmann, The SPIN Model Checker Primer and

[7] Soren Lauesen & Otto Vinte, "Preventing Requirements Defects", Sixth International Workshop on Requirements, Stockholm, 2000.

[8] OMG Unified Modeling LanguageTM (OMG UML), Superstructure version 2.2; Object Management Group, Feb. 2009.

[9] Morkos, B., Summers, J., (2009), “Elicitation and Development of Requirements Through Integrated Methods”, ASME Design Engineering Technical Conferences, San Diego, CA, Aug. 30-Sep. 2, 2009

[10] R. Fuentes, "Requirements Elicitation for Agent-based Applications", 2005

[11] L. Chung, "On Non-Functional Requirements in Software Engineering", Springer, 2009.

[12] J. H. Hayes, “Risk Reduction Through Requirements Tracing,” Proc. of 1990 Software Quality Week, San Francisco, CA, 1990.

[13] J. H. Hayes, A. Dekhtyar, J. Osbourne, “Improving Requirements Tracing via Information Retrieval,” in Proceedings of the 2003 IEEE International Conference on Requirements Engineering, Monterey, California, Sept. 2003.

[14] J. H. Hayes, A. Dekhtyar, S. Sundaram, S. Howard, “Helping Analysts Trace Requirements: An Objective Look,” in Proceedings of the IEEE Requirements Engineering Conference (RE) 2004, Kyoto, Japan, Sept. 2004, pp. 249-261.

[15] W. Wilson., L. Rosenberg, and L. Hyatt, "Automated Quality Analysis of Natural Language Requirement Specifications," in the Proceedingsof the 14th Annual Pacific Northwest Software Quality Conference Portland, 1996.

[16] R. L. Cobleigh, G. S. Avrunin, L. A. Clarke, “User Guidance for Creating Precise and Accessible Property Specifications,” in the Proceedings of the ACM SIGSOFT 14th International Symposium on Foundations of Software Engineering (FSE14), Portland, OR, pp. 208-218, Nov. 2006.

[17] University of Massachusetts, Laboratory for Advanced Software Engineering Research, PROPEL web page, http://laser.cs.umass.edu/tools/propel.shtml. Last viewed Dec 14, 2011.

[18] A. Nikora, G. Balcom, “Improving the Accuracy of Space Mission Software Anomaly Frequency Estimates,”proceedings of Third International Conference on Space Mission Challenges for Information Technology (SMC-IT 2009), Jul 2009, Pasadena, CA

[19] Requirements Assistant Webpage, http://www.requirementsassistant.nl/. Las viewed May 16, 2012.

[20] Renkema, J., "Introduction to Discourse Studies". John Benjamin Publishing Company, 2004.

[21] S. Boyd, D. Zowhi, and A. Farroukh, "Measuring the expressiveness of a constrained natural languague: an empirical study", Proc. the 13th IEEE International Conference on Requirements Engineering (RE'05), Washington, 2005, pp. 339-352

[22] D.C. Gause and G.M. Weinberg, "Exploring requirements: quality before design", Dorset House, New York, 1989.

[23] D. M. Berry, E. Kamsties, and M. M. Krieger, "From contract drafting to software specification: Linguistic sources of ambiguity", 2003.

[24] L. Goldin and D. M. Berry. Abstfinder, "A prototype natural language text abstraction finder for use in requirements elicitation", Automated Software Engineering, 1997.

[25] Brooks, F., “No Silver Bullet: Essence and Accidents of Software Engineering”, IEEE Computer, 1987, pp. 10-19.

[26] Natural Language Toolkit Official Website, http://www.nltk.org/ .Last viewed Jan 14, 2012.

[27] A spell checking library for Python website, http://packages.python.org/pyenchant/ . Last viewed Jan 14, 2012.

[28] Aby Word Processing website, http://www.abisource.com/projects/enchant/ . Last viewed Jan 14, 2012.

[29] Knowing What's What and What's Not: The Five W's (and 1 "H") of Cyberspace". Media Awareness Network. Retrieved September 12, 2008

[30] Kristina Toutanova and Christopher D. Manning. 2000. Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. In Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000), pp. 63-70.

[31] University of Pennsylvania, Penn Treebank POS website, http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html . Last viewed Jan 14, 2012.

[32] NLTK RegX Parser Documentation, http://nltk.googlecode.com/svn/trunk/doc/api/nltk.chunk.regexp.RegexpParser-class.html . Last viewed Jan 14, 2012