# Automated Detection of Language Issues Affecting Accuracy, Ambiguity and Verifiability in Software Requirements Written in Natural Language

**Allan Berrocal Rojas, Gabriela Barrantes Sliesarieva**
Escuela de Ciencias de la Computación e Informática
Universidad de Costa Rica, San José, Costa Rica
{allan.berrocal,gabriela.barrantes}@ecci.ucr.ac.cr

## Abstract

Most embedded systems for the avionics industry are considered safety critical systems; as a result, strict software development standards exist to ensure critical software is built with the highest quality possible. One of such standards, DO-178B, establishes a number of properties that software requirements must satisfy including: *accuracy*, *non-ambiguity* and *verifiability*. From a language perspective, it is possible to automate the analysis of software requirements to determine whether or not they satisfy some quality properties. This work suggests a bounded definition for three properties (*accuracy*, *non-ambiguity* and *verifiability*) considering the main characteristics that software requirements must exhibit to satisfy those objectives. A software prototype that combines natural language processing (NLP) techniques and specialized dictionaries was built to examine software requirements written in English with the goal of identifying whether or not they satisfy the desired properties. Preliminary results are presented showing how the tool effectively identifies critical issues that are normally ignored by human reviewers.

## 1 Introduction

Software requirements play a critical role in the software life cycle. It has been observed that poorly written software requirements often lead to weak and unpredictable software applications (Wilson et al., 1997). Besides, the cost of fixing errors increases exponentially throughout the different phases of software development (Galin, 2004; Leffingwell and Widrig, 2003). In other words, it is less expensive to fix an error in the software requirements phase than it is to fix the same error during the integration or verification phase.

Embedded systems for the avionics industry are developed following particularly rigorous restrictions due to strict safety and availability constraints that need to be satisfied during air or ground operations. DO-178B (RTCA, 1992) is a recognized standard for development of safety critical embedded systems. It is widely used by software certification authorities such as FAA (*Federal Aviation Association*), and it establishes some guidelines and quality objectives for each phase of a software development effort. In particular, the standard estates that software requirements must be *accurate*, *verifiable*, and *non-ambiguous*.

### 1.1 Software Quality

Milicic suggests that software quality can be understood as conformity with a given specification (Milicic et al., 2005). This definition is in total agreement with DO-178B, which requires that software is designed, built, and tested following approved standards for each phase of the development cycle.

Extrapolating the previous definition, one can argue that quality of software requirements can be understood as the degree to which software requirements also comply with a given specification. In other words, in order to produce high quality software requirements, one needs to ensure that they satisfy the criteria established in a software requirements standard.

In the case of software requirements written in natural language (NL), some of the criteria can be addressed from a linguistic perspective, observing certain types of language constructs and language usage in general that may represent violations against desired quality criteria in a standard.



*Proceedings of the NAACL HLT 2010 Young Investigators Workshop on Computational Approaches to Languages of the Americas*, pages 100–108, Los Angeles, California, June 2010. 

### 1.2 Overview of Research Goals

The overall objective of this research was to identify some of the linguistic elements that one can observe to determine whether or not software requirements written in natural language[1] comply with three specific properties established by DO-178B: *accuracy*, *verifiability* and *non-ambiguity*. Those linguistic elements can be seen as rules in an expert system, so that requirements are said to be compliant with their quality objectives when they satisfy all rules. They are said to be non-compliant with their quality objectives when rules are not satisfied.

In this research, linguistic elements were identified and independently validated by professionals in the field of software verification. Later on, a software prototype capable of examining a list of requirements was built to automatically detect when requirements do not satisfy a given rule.

The main contribution of this research is that it provides a quantitative evaluation of the target requirements. More specifically, based on the number of satisfied and non satisfied rules, the prototype scores each requirement in a 1 to 10 scale. The tool also provides additional information (qualitative analysis) to the user indicating the root of the problem when a given rule is not satisfied, as well as possible ways to fix the issue.

### 1.3 Justification

The author's experience in the field of requirements verification suggests that the task of reviewing a set of requirements for compliance with properties such as *accuracy*, *verifiability* and *non-ambiguity* is a non trivial task. This is particularly true when the reviewer lacks the proper training and tools. Some of the known difficulties for this process are:

- It requires linguistic (e.g. grammar, semantics) and technical knowledge from a reviewer.
- There is no warranty that two or more reviewers will produce the same findings for the same input (mostly due to the informal nature of NL).
- The process is error prone since reviewers become fatigued after some time.
- The process is time consuming, which directly affects budget and schedule performance.

Having a tool that partially automates the process of reviewing software requirements may represent significant improvements in the overall software life cycle process. Even when current developments in computational linguistics do not provide a complete solution for the problem at hand, a partial approach is still valuable producing numerous advantages such as:

- Linguistic and technical knowledge is input into the system in a cumulative manner, reducing dependency on highly qualified personnel.
- Results are reproducible for any given set of inputs, reducing inconsistencies while adding reliability to the results.
- Review time is significantly reduced.

## 2 Related Work

Significant work has been done in the area of software requirements analysis. Lami (Lami et al., 2004) classifies these efforts in three groups. A first group consists of preventive techniques that need to be applied during the process of writing requirements. Those techniques normally trigger checklists that are enforced by a person with no support from tools, see for instance (Firesmith, 2003). Another group consists of restrictive techniques that limit the degree of language freedom when writing requirements. One example in this group is Fuchs (Fuchs et al., 1998) who introduces ACE (Attempto Controlled English), a restricted subset of English with a restricted grammar and domain specific vocabulary. Requirements can be written in natural language with enough expressive power. They are later translated into first order predicate logic to be processed formally by a computer program.

The last group of efforts consists of analytic techniques that perform automated analysis of requirements once they have been produced. The following are two relevant projects in this group. Wilson (Wilson et al., 1997) developed a tool named ARM that performs automated analysis of a requirements document. The tool focuses on lexical analysis to detect specific keywords such as vague adverbs and

[1] This research assumes requirements are written in English.

vague adjectives that are not desired. Different from our work, ARM also checks that the document itself complies with a specific format. Then, Lami (Lami et al., 2004) described a systematic method for automated analysis of requirements detecting deficiencies such as ambiguity, inconsistencies, and incompleteness. A tool named QuARS implements the suggested methodology and appears to be a good contribution in this area [2].

## 3 Theoretical Framework

This section provides a basic explanation of some concepts that are commonly used in the field of software verification. Emphasis will be made on concepts related to the software engineering field in an attempt to set the grounds for the investigation. Other linguistic related concepts will be mentioned along the paper assuming the reader has basic understanding of them.

### 3.1 Software Life Cycle

A Software Life Cycle Model or Software Development Model consists of a group of concepts and well coordinated methodologies that guide the software development process from beginning to end (Galin, 2004). The classic software life cycle model (a.k.a. *the waterfall model*) consists of linear sequence of activities or phases that take place during a software development effort.

In the **Requirements elicitation phase**, a detailed description of what the software shall do is produced. Although there are various methods, a natural language description in the form of a list of statements is widely used to produce requirements.

A **software requirement** is a condition or characteristic that a system must possess to satisfy a contract, a standard, a formal specification or other applicable regulation (IEEE, 1990).

In simple words, a software requirement explains how the system should behave or react given a specific set of inputs and initial conditions. While not true for all software applications, in the avionics industry, all software functionalities are required to be fully deterministic. This means that the system must behave exactly the same all the time for a given set of inputs and initial conditions. This is why correctness of requirements is so critical.

The following section briefly comments on three of the properties that requirements must satisfy to meet quality objectives. Although there are many such properties, we focus on three whose detection is partially automated in this research.

### 3.2 Quality Properties for Software Requirements

To meet quality objectives, software requirement must be *accurate*, *non-ambiguous* and *verifiable*. This section provides a brief explanation of these terms in the context of software verification. Additionally, it describes the main language elements used in this research to automatically detect when software requirements do not satisfy a given property.

#### 3.2.1 Ambiguity

A word or phrase is said to be ambiguous when it has more than one possible meaning causing confusion or uncertainty. Similarly, software requirements are said to be ambiguous when they admit more than one possible interpretation. An ambiguous requirement is notably incompatible with the goal of producing deterministic software.

Berry (Berry, 2003) distinguishes six major forms of ambiguity in software requirements: lexical, syntactical, semantic, pragmatic, vagueness and language error. In this research, we focused on lexical, syntactic, vagueness, and language errors since this group covers common deficiencies that show in requirements.

One form of syntactical ambiguity occurs when requirements fail to group logical conditions (e.g. AND, OR) with appropriate punctuation marks or explicit parenthesis. In the following example, for instance, it is not clear what the conditions are for the system to enter into normal mode: "*The system shall enter Normal mode when SDI field on label 227 equals 2 or SSM in label 268 equals 3 and WOW is true or AIR is false*."

Vague adverbs usually modifying nouns (such as: *acceptable, high, low, fast, in/sufficient, normal, similar* and many others) also create ambiguous requirements like the following: "*The system shall allow the operator to adjust volume to an acceptable level*."

[2] The author has not been able to use QuARS yet.

Finally, non deterministic constructs such as *and/or, any, not limited to* also create ambiguity in requirements, such as the following case: "*The system shall display altitude and/or temperature at the bottom line of the screen*."

### 3.2.2 Accuracy

In a requirement, accuracy refers to how concise and precise a requirement is specified. Accuracy should be present not only in the content but also in the structure of a requirement.

In terms of structure, a requirement must clearly distinguish between at least two parts: condition and action. A requirement with a clear action and no condition opens a possibility to think that the specified action is permanent (which is rarely the case). On the other hand, by definition, there can not exist a requirement with no action.

For instance, the following requirement is inaccurate: "*The system shall clear the DMA shared space*," since no one knows when the action must occur.

In terms of content, a requirement must include clear and detailed information about the condition and the action that is being described. Accurate requirements also include explicit units for physical values as well as tolerances and thresholds for numerical computations.

For instance, the following requirement is inaccurate "*The system shall send ARINC label 251 every 50 ms*," but adding a tolerance value solves the issue as in "*The system shall send ARINC label 251 every 50 ms +/- 5ms*."

Non deterministic adverbs usually modifying verbs (such as: *continually, periodically, regularly* and others) also create inaccurate requirements like the following: "*The system shall periodically perform CBITE*."

Finally, there are a number of general verbs that should be avoided in requirements since they create inaccurate descriptions. Some of these verbs are: *process, monitor, support, check* among others. For instance, it is not clear to see the software action that this requirement implies: "*The system shall monitor responses from the slave processor*."

### 3.2.3 Verifiability

A requirement is said to be verifiable if it is possible to create and execute a test to demonstrate that the software behaves exactly as specified in the requirement.

Sometimes a test can not be executed primarily because of hardware or test equipment limitations. In other cases, conflicts or inconsistencies between requirements are revealed which prevent a test from being performed. However, another group of requirements become non verifiable due to language usage errors.

For instance, by definition requirements are meant to describe actions that the system shall perform. In that sense, a requirement must not describe anything that the system shall not perform. To illustrate, a requirement such as the following is *non verifiable*: "*The system shall not enter INTERACTIVE mode when WOW is false*." The reason is that a tester can not expect any specific system action during a test for this requirement.

Furthermore, requirements using the adverbs *always* and *never* are also *non-verifiable* since a test for them would require infinite time. Similarly, the term *only* must be used correctly when modifying the main action (verb) of the requirement. For example, the requirement "*The system shall only display invalid data in red color*" implies that the only action this system performs is "*display invalid data in red color*." The intended meaning is probably "*The system shall display only invalid data in red color*."

Finally, some requirements contain verbs that imply actions that a software application can not perform; instead, these are usually human-specific tasks that are incorrectly assigned to software. Some of these verbs are: *determine, ignore, consider, analyse* and others. One example of wrong usage is "*The system shall consider fault history during CBITE*."

As mentioned in section 1.2, one objective was to provide a quantitative evaluation of a set of requirements against three properties: *accuracy, ambiguity* and *verifiability*. With that goal in mind, section 4.1 introduces some of the formulations that will be used to perform the evaluation of the requirements against the selected properties.

## 4 Research Foundation

To accomplish the general objectives described in section 1.2, section 4.1 introduces a semi-formal nomenclature used to express the various situations

when a requirement either satisfies or violates any of the desired properties. This nomenclature is valuable for it allows to represent various situations in a symbolic and summarized way. Section 4.2 describes the process followed to select the criteria against which the software requirements will be evaluated for quality.

### 4.1 Proposing General Nomenclature

We will use the term *element* to refer to individual linguistics elements or rules as mentioned in section 1.2. Similarly, the term *attribute* refers to quality properties: *accuracy*, *ambiguity* and *verifiability*.

To represent the attribute *ambiguity*, we define the set $\Lambda = \{\lambda_1, \lambda_2, ..., \lambda_k\}$ with $k \in \mathbb{N}_{\geq 1}$ where each $\lambda_i$ is an element that reveals a non compliance for the attribute of *ambiguity* by a given requirement. For instance, let's assume $\lambda_1 =$ "*A requirement must not use vague or general adverbs to describe an action*," then, if we apply $\lambda_1$ to the requirement $R_1 =$ "*The system shall allow the operator to adjust volume to an acceptable level*," we conclude that $R_1$ is *ambiguous* since the adverb "*acceptable*" is vague. In that case we say that $R_1$ does not satisfy $\lambda_1$.

For *accuracy* we define $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_r\}$ with $r \in \mathbb{N}_{\geq 1}$, and for *verifiability* we define $\Upsilon = \{\upsilon_1, \upsilon_2, ..., \upsilon_s\}$ with $s \in \mathbb{N}_{\geq 1}$ in an analogous way. Summarizing, we define:

$$X_1 = \Lambda \quad , \quad X_2 = \Gamma \quad , \quad X_3 = \Upsilon$$

where $X_i = \{\epsilon_1, \epsilon_2, ..., \epsilon_n\}$ and each $\epsilon_i$ is an *element* that tells us if a requirement does not satisfy a specific attribute.

We propose the following notation to represent situations where requirements fail to satisfy an element.

- When a requirement $R_e$ meets the restriction imposed by an element $\epsilon_k$, we say that $R_e$ *satisfies* $\epsilon_k$, and we write $R_e \odot \epsilon_k$.
- When a requirement $R_e$ does not meet the restriction imposed by an element $\epsilon_k$, we say that $R_e$ *does not satisfy* $\epsilon_k$, and we write $R_e \oslash \epsilon_k$.
- When the restriction imposed by an element $\epsilon_k$ is not applicable for a requirement $R_e$, we say that $\epsilon_k$ *is not applicable* for $R_e$ and we write $R_e \oplus \epsilon_k$

Notice how the expressions $R_e \odot \epsilon_k$ , $R_e \oslash \epsilon_k$ and $R_e \oplus \epsilon_k$ can be seen as logical predicates for a binary relation. For instance, we could read the first expression as $\odot(R_e, \epsilon_k)$ or SATISFIES$(R_e, \epsilon_k)$ .

However, computing the degree in which a requirement satisfies an attribute is not a binary relation. For instance, a requirement can meet some restrictions and not others; besides, some restrictions are more critical than others.

For a more objective evaluation, a scale from 0 to 10 is proposed. Each element $\epsilon_1, \epsilon_2, ..., \epsilon_n$ in $X_i$ is assigned a value or score so that the scores for all elements in an attribute add up to 10 and $score(\epsilon_i) = p$ , $p \in \mathbb{R}^+_{10}$ (where $\mathbb{R}^+_{10} = [0, 10]$). Values are assigned depending on the criticality and type of error revealed by each element. Summarizing:

$$\sum_{j=1}^{|X_i|} score(\epsilon_j) = 10 \quad \text{where } \epsilon_j \in X_i \text{ and } X_i \in \{\Lambda, \Gamma, \Upsilon\} \tag{1}$$

Now, in order to **evaluate** a requirement $R_e$ in regards to an attribute, we define $\theta$: $(R) \rightarrow \mathbb{R}^+_{10}$:

$$\theta(R_e, X_i) = [10 - \sum_{j, R_e \oslash \epsilon_j} score(\epsilon_j)] = [\sum_{j, R_e \odot \epsilon_j} score(\epsilon_j)] \quad \text{where } \epsilon_j \in X_i \tag{2}$$

Notice how we write $\theta$ using either predicate *does not satisfy* $\oslash$ or *satisfies* $\odot$. In both cases, when an element *is not applicable* $\oplus$ to a requirement, we assume that the requirement *satisfies* such element.

To understand the meaning of $\theta$, suppose that $x = \theta(R_e, \Gamma)$ is the score a requirement $R_e$ gets when it is evaluated against a given attribute, let's say *accuracy*.

- When $x = 10$ we say that $R_e$ satisfies $\Gamma$ and we write $R_e \odot \Gamma$. $R_e$ is accurate, since it meets all the restrictions imposed by each element in $\Gamma$.
- When $x = 0$ we say that $R_e$ does not satisfy $\Gamma$ and we write $R_e \oslash^0 \Gamma$. $R_e$ is not accurate, since it does not meet any of the restrictions imposed by elements in $\Gamma$.

- When $0 < x < 10$ we say that $R_e$ does not satisfy $\Gamma$ with a degree $x$ and we write $R_e \oslash^x \Gamma$. $R_e$ meets some of the restrictions imposed by elements in $\Gamma$ but not all. In this case, the closer $x$ is to 10, the better the requirement will be[3].

Finally, to get a requirement's overall score against all three attributes, we define a function $\phi : (R) \rightarrow \mathbb{R}^+_{10}$ as follows:

$$\phi(R_k) = \frac{\sum_{i=1}^{3} \theta(R_k, X_i)}{3} \quad \text{where } X_i \in \{\Lambda, \Gamma, \Upsilon\} \tag{3}$$

$\phi$ is the arithmetic mean of the scores a requirement gets against each attribute h$X_i$ in (2). The overall score is a measure of a requirement's quality, and it could be used potentially to estimate costs in a software project.

To understand $\phi$ suppose $x = \phi(R_k)$ is the score a requirement $R_k$ gets when it is evaluated against all three attributes (*ambiguity*, *accuracy* and *verifiability*) using all elements in $\{\Lambda, \Gamma, \Upsilon\}$.

- When $x = 10$, we say that $R_k$ is *accurate*, *verifiable* and *non-ambiguous* since $R_k \odot X_i \quad \forall_{X_i} \in \{\Lambda, \Gamma, \Upsilon\}$.

- When $x = 0$ we say that $R_k$ is *inaccurate*, *non-verifiable* and *ambiguous* since $R_k \oslash X_i \quad \forall_{X_i} \in \{\Lambda, \Gamma, \Upsilon\}$.

- When $0 < x < 10$, we say that $R_k$ is either *inaccurate*, *non-verifiable* or *ambiguous* since it does not satisfy at least one element in $\{\Lambda, \Gamma, \Upsilon\}$. In this case, $\theta$ provides more information about the weakness detected in $R_k$.

The value of the suggested notation comes from the fact that we can now produce quantitative evaluations of requirements, as opposed to common qualitative evaluations. The following sections briefly describe a bottom up process we followed to select evaluation criteria for the prototype that was built.

[3]We will use either notation $\oslash$ or $\oslash^x$ to indicate that a requirement does not satisfy an attribute and the degree $x$ is not relevant.

### 4.2 Selecting Criteria for Evaluation

The process of selecting the elements for each attribute was conducted in a series of steps that are summarized below. The objective of our approach was to provide a selection of elements that satisfied three main goals. The first goal was to have representative and useful selection within the field of software verification. The second goal was that the selection could be independently validated by a group of professionals in the field. And finally, the third goal was that the selection of elements refers to weaknesses that can be automatically detected by a software.

The following is a summary of the process that was followed to select the criteria to evaluate requirements.

1. A list of elements was first suggested by the author based on relevant literature and his own experience in software verification for embedded systems. The list contained 19 elements (10 for *accuracy*, 5 for *ambiguity* and 4 for *verifiability*).

2. Five elements were filtered out as they were not candidates to be automated. Feasible candidates were those that could be automated using techniques such as parsing, tagging, regular expressions, and specialized dictionaries like WordNet (Miller, 1993) and VerbNet (Kipper, 2005). The list ended with 6 elements in *accuracy*, 4 in *ambiguity* and 4 in *verifiability*.

3. The author suggested an initial value or score for each element in the list.

4. Both the element selection and the value distribution were independently validated by a group of three professionals with demonstrable experience in software verification[4].

5. A numerical model was prepared based on the proposed approach described in section 4.1. This is already a contribution since the evaluation of the requirements could be done manually in case no tool had been created.

[4]Although these individuals are not language experts, since they have valuable experience in requirements verification, their feedback was considered a valid complement in this research.

6. A software prototype was written for a tool that is capable of examining a list of requirements applying equations 2 and 3 in section 4.1.

Section 5 briefly describes the capabilities of the prototype tool that was developed.

## 5 Automated Evaluation

This section provides a brief description of the software prototype. A more in depth description would be ideal; however, due to space limitations we will focus on two items only. First, an overview of the tool's architecture and technologies involved (section 5.1). Second, a description of the outputs this tool produces (section 5.2).

### 5.1 Building the Prototype

Our prototype tool receives the name of SRR-Director from *Software Requirements Reviewer Director*. This prototype was built using open source software and tools that are freely available for research. Our goal was to integrate several of these available resources into a single piece of software that helped us solving the problem we are studying.

Perl[5] was used as the main language for the software and Awk[6] was used as an independent tool to check some of the results while developing the tool. Input requirements normally exist in various formats such as MS Word[7], MS Excel[8], structured XML, or plain text files. We provide a tool that can be configured to read those inputs converting them into XML documents that follow a normalized structure which basically separates requirements identifiers from the actual text of the requirement.

The three main techniques used during automated inspection of the requirements were the following:

**Lexical Analysis**: this is a common and simple technique that is based on regular expressions. Perl's engine for regular expressions was particularly useful in this task. This type of analysis allows identification of key words or phrase structures that reveal specific types of weaknesses in requirements.

This technique helped identifying issues of all three types. For ambiguity it allowed to locate vague adverbs and non deterministic language constructs; for accuracy, we detected tolerance issues, non deterministic adverbs and general verbs. Finally, to check for verifiability this technique was used to capture negative requirements, infinite requirements, and wrong usage of the term *only*.

**Syntactic Analysis**: consists of parsing the requirements to transform language statements into their grammatical constituents which enables other specific analysis such as ambiguity analysis. This process was performed using a parser made available by Eugine Charniak and Brown University (Charniak et al., 2006). In this case, the CLAIR group at University of Michigan made available a Perl wrapper for the Charniak parser(CLAIR, 2009).

Studying the syntax tree produced by the parser, it was possible to identify accuracy issues such as requirements without explicit condition statements (or condition blocks). Also, studying the output of the parser along with lexical analysis of the requirement reveals cases of ambiguity when logical conditions are not stated clearly.

**Dictionaries**: two great resources were also incorporated in this research to support our analysis: WordNet (Miller, 1993) and VerbNet (Kipper, 2005). Both of this tools can also be accessed from Perl via wrappers and provide useful information about words and verbs that were used to ensure some conditions were valid while we perform the analysis of the requirements.

Dictionaries allow identification of human specific verbs and ambiguous verbs. In this case, the parser makes it possible to capture the main verb for a requirement, and further queries into dictionaries complete the task. VerbNet provides a mechanism to classify requirements according to their degree of ambiguity. This mechanism may be too stringent for flagging ambiguous verbs sometimes. There are verbs tagged as ambiguous in VerbNet, but they have a fairly well known and shared meaning in the domain of software engineering such as: *set, shutdown, turnoff, send, receive* among others.

SRR-Director runs from a command line and it is currently controlled using a number of arguments and switches. Even when this is still a prototype tool, our experiments show that the tool is very efficient capturing weaknesses in the requirements with a marginal error rate ($< 5\%$) for the rules included

[5] http://www.perl.org
[6] http://www.gnu.org/software/gawk/
[7] http://office.microsoft.com/word
[8] http://office.microsoft.com/excel

in the current version of the tool. More importantly, the tool is able to examine hundreds of requirements in a matter of minutes when the same work takes hours or even days for a human reviewer.

### 5.2 Using the Reports

In the current prototype version, the tool produces seven types of reports that provide information for three types of users:

**Quality engineers**: two reports show general information about the quality of the requirements that were analysed. Quality engineers are interested in the overall percentage of requirements compliance with the quality objectives, and they don't need details on the types of failures.

**Requirements engineers**: four reports are available for the largest audience of users who are actually interested in learning the details about the types of failures identified in the requirements. Not only are the engineers notified of the weaknesses but also they are provided with suggestions on how to fix the issues. The evaluation they receive is not only qualitative but also quantitative since they can see the score for individual requirements against each of the three properties being studied.

**Software engineers**: this is a miscellaneous report that provides performance information which may later help software engineers while tuning certain processes in the tool.

## 6 Experiments and Results

Experiments were performed using sample requirements from three distinct and real word applications in embedded systems. Test data was selected from a pool of reserved requirements that were not used during development of the tool.

Four groups of 20 requirements each were selected and given to three experienced professionals in the field of software verification. The subjects were asked to identify weaknesses in the requirements using their own criteria. They were asked to classify ill requirements as *inaccurate*, *ambiguous* or *non-verifiable* when applicable. The same groups were input to the prototype for evaluation, and results were compared.

As it was mentioned before, the tool recognizes all deficiencies described by a rule or *element* with a low error rate ($< 5\%$). We believe this is mostly due to the fact that –in this initial phase of the tool– rules are not complex, and can indeed be automated without using complex techniques.

One interesting result was that a high degree of discrepancies and disagreement between the subject reviewers was observed . On average, the three reviewers agreed only in $14\%$ of their evaluations, and only in $62\%$ of the cases there was agreement between at least two reviewers. These unexpected discrepancies certainly make it difficult to compare the tool's results with the reviewer's results to identify areas of agreement or disagreement.

A more in depth analysis of the results suggests that human beings may perform erratically when it comes to reviewing requirements that contain the types of errors we are looking for. Some of the weaknesses we want to uncover are rather subtle and, as we argued before (section 1.3), require a good level of language and technical knowledge as well as a detail oriented attitude. People are also affected by external factors such as fatigue that negatively affects the quality of their work.

## 7 Conclusions

The results of this research show that it is actually possible to automate the review process of software requirements identifying valuable sources of deficiencies that otherwise make requirements *inaccurate*, *ambiguous* or *non-verifiable*.

Besides, there are resources freely available for research that can be integrated into more specific tools to solve a variety of problems. Specialized dictionaries, stand alone tools, such as parsers, and a general purpose scripting language (Perl) were combined in order to create the tool prototype.

Finally, a simple but rather useful nomenclature to represent different scenarios that occur during requirements verification was proposed. This contribution allows us to provide a quantitative analysis of the requirements as opposed to traditional qualitative-only analyses.

## 8 Collaboration Opportunities

This section answers two specific questions to describe possible collaboration opportunities between investigators doing research on similar topics.

### 8.1 How can this work benefit other research projects?

This research was focused on three properties applicable to software requirements for aerospace systems. However, it would be ideal to apply similar techniques to examine other types of properties that are crucial in similarly critical application domains such as finance, transportation, medicine and communications.

In this work, inputs are text documents with natural language text in the form of software requirements. Those inputs are preprocessed and converted into simpler representations that basically consist of sentences. Those sentences at the end are the main input for the tool that performs the automated quality analysis.

Researchers wishing to learn more about this work are strongly encouraged to contact the author to share ideas on this topic and benefit from one another. We believe it is possible to reuse part of the approach to build similar tools to analyse requirements in languages other than English.

### 8.2 What are some resources and expertise the author lacks?

One of the main difficulties the author faced is the absence of collaboration between researchers interested in similar topics. This work has been produced mostly in isolation as part of academic research in a masters program.

Being able to share ideas with working groups either academic or industry sponsored would be a great channel to improve research scope and produce more significant results. For the nature of this research, a mixed team of linguists and software engineers would presumably improve the quality of the work.

On the one hand, linguists would provide valuable knowledge that would help identifying additional language structures that represent symptoms of weaknesses in requirements. On the other hand, software engineers would be closer to requirements engineers and could contribute with implementation details so that new rules are added to the system.

## References


Wilson, W. and Rosenberg, L. and Hyatt, L. 1997. *Automated Analysis of Requirement Specifications*. Nineteenth International Conference on Software Engineering (ICSE-97), Boston, MA.

Galin, D. 2004. *Software Quality Assurance From theory to implementation*. Pearson Education Ltd.

Leffingwell, D. and Widrig, D. 2003. *Managing Software Requirements*, 2nd Ed. Addison-Wesley.

RTCA/EUROCAE. 1992. *DO-178 Software Considerations for Airborne Systems and Equipment Certification*. RTCA, Inc., Washington, DC.

Drazen, M. and Berander, P. and Damm, L. and Eriksson, J. and Gorschek, T. and Henningsson, K. and Jonsson, P. and Kagstrom, S. and Martensson, F. and Ronkko, K. and Tomaszewski, P. . 2005. *Software quality attributes and trade-offs*, *Software Quality Models and Philosophies*. Blekinge Institute of Technology.

IEEE Standards Board. 1990. *IEEE Standard Glossary of Software Engineering Terminology*, Std 610.12-1990.

Berry, D. and Kamsties, E. and Krieger, M. . 2003. *From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity*.

Miller, G. and Beckwith, R. and Fellbaum, C. and Gross, D. and Miller, K.. 1993. *Introduction to WordNet: An Online Lexical Database*. Cognitive Science Laboratory, Princeton University.

Kipper, K.. 2005. *VerbNet: A broad-coverage, comprehensive verb lexicon*. Computer and Information Science, University of Pennsylvania.

Lami, G. and Gnesi, S. and Fabbrini, F. and Fusani, M. and Trentanni, G. . 1997. *An Automatic Tool for the Analysis of Natural Language Requirements*. C.N.R. Information Science and Technology Institute, Pisa Italy.

McClosky, D. and Charniak, E. and Johnson, M.. 2006. *Reranking and Self-Training for Parser Adaptation*. 21st International Conference on Computational Linguistics.

CLAIR official website. 2009. *URL* `http://belobog.si.umich.edu/clair/clair/downloads.html`. Visited on March 12, 2009.

Fuchs, N. and Schwertel, U. and Schwitter, D.. 1998. *Attempto Controlled English Not Just Another Logic Specification Language*. Eighth International Workshop on Logic-based Program Synthesis and Transformation LOPSTR'98, Manchester, UK.

Firesmith, D.. 2003. *Specifying Good Requirements*. Journal of Object Technology, ETH Zurich.