PARSING OF NATURAL LANGUAGE REQUIREMENTS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Sciences in Computer Science

by

Jamie L. Patterson

October 2013

COMMITTEE MEMBERSHIP

TITLE:	Parsing of Natural Language Requirements
AUTHOR:	Jamie L. Patterson
DATE SUBMITTED:	October 2013
COMMITTEE CHAIR:	Clark Turner, Ph. D. Professor of Computer Science
COMMITTEE MEMBER:	Alex Dekhtyar, Ph. D. Professor of Computer Science
COMMITTEE MEMBER:	Franz Kurfess, Ph. D. Professor of Computer Science

ABSTRACT

Parsing of Natural Language Requirements

Jamie L. Patterson

The purpose of this thesis was to automate verification of the software requirements for an implantable cardioverter defibrillator with minimal manual rework. The requirements were written in plain English with only loose stylistic constraints. While full automation proved infeasible, many significant advances were made towards solving the problem, including a framework for storing requirements, a program which translates most of the natural language requirements into the framework, and a novel approach to parts of speech analysis.

Keywords: natural language, software requirements, parts of speech analysis

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

In software development, especially in safety-critical applications, quality is of utmost importance. There are many kinds of software development issues that arise. Requirements errors are common in software development and lead to some of the most costly malfunctions. In an attempt to improve software requirements, this thesis will explore the benefits and limitations of automated verification of natural language requirements. First, existing solutions to the problem will be analyzed in a literature review. Next, a potential solution based on the most promising ideas from the literature will be developed and attempted on an existing requirements document. Finally, the results of the potential solution will be examined and suggested future work will be presented.

2. Motivation

Implantable medical devices, such as implantable cardioverter defibrillators regulate physiological activity within the human body. Incorrect operation can cause loss of consciousness, pain, or even death. Much like the doctor who performs the implant, the engineers that design and build these devices are responsible for their patients. The wellbeing of the patients provides a strong impetus for the engineers to ensure that operation is correct.

In safety-critical applications, the cost of malfunctions is not only money and inconvenience. In the worst case scenario, the consequence is loss of life. The importance of quality in medical device design and the existence of requirements errors in the current document inspired this exploration into methods for improving the requirements.

Another motivation for this thesis is the application of academic research to a real-world problem. Pure research can explore solutions to problems without being bound by the budget constraints and corporate culture. Since this project is designed for an existing company with real constraints on time and money, special considerations will have to be made.

3. Quality in Requirements

3.1 Importance of Quality Requirements

In Formal Methods: State of the Art and Future Directions, Edmund Clarke and Jeannette Wing state:

> "The process of specification is the act of writing things down precisely. The main benefit of doing so is intangible – gaining a deeper understanding of the system being specified. It is through this specification process that developers uncover design flaws, inconsistencies, ambiguities, and incompleteness." [4]

The importance of high quality requirements cannot be overstated. According to Nancy Leveson, "Accidents and major losses involving computers are usually the result of incompleteness or other flaws in the software requirements, not coding errors". [8] One industry survey of over 8000 projects stated that one third of the projects were never completed and another half were partially completed. The primary cause was identified as requirements. [37] Because of the importance of quality requirements and the problems associated with writing them, many computer scientists have questioned how the quality of requirements can be improved. This has resulted in a significant amount of literature being published concerning the correctness of requirements. [37]

3.2 Definition of Quality Requirements

To help ensure that software projects success, engineers need to produce quality documentation. [43] Arthur and Stevens define the four main quality characteristics for software documentation as accuracy, completeness, usability, and expandability. [45] Accuracy is important because

mistakes in the requirements become more costly as they propagate to the software and test design and can go through the entire software development cycle undetected. [43] Next, the requirements must be complete. An experiment was conducted where some problem solvers were given an incomplete problem representation while others were given no problem representation at all. The test subjects with no representation did better because those with the incomplete problem representation tend to treat it as a comprehensive and truthful representation. They did not consider other factors that were missing. [10] Requirements must also be clear to avoid misunderstandings from the clients, developers, and verifiers. Next, the documentation must be usable. Arthur and Stevens define usability as “the suitability of the documentation relative to the ease with which one can extract needed information”. [45] Examples of this include clarity because ambiguous requirements make the document difficult to understand and use. Also, the requirements must be accuracy, completeness, usability; otherwise verifiers cannot use the document to guarantee that the end product satisfies the original goals. Lastly, the quality of the document can be assessed by its expandability, which is the ease with which the document can be modified. While expandability is important, it largely deals with predicting the future state of the requirements and is beyond the scope of this thesis.

3.3 Root Cause of Ambiguity in Requirements

Ideally, requirements could be structured in the way that engineers could ensure accuracy, completeness, and usability. Traditionally, requirements are written using a natural language. English or other natural languages are problematic though because of their inherent ambiguities. [33] Even the simplest phrases can be understood completely differently by different people. Consider the following example:

"On a ventricular event, the software shall display the EGM data within 200 milliseconds of the event."

The phrase "within 200 milliseconds of the event" could represent a time constraint for displaying of the data or it could describe the amount of data that should be displayed. Reduction of ambiguity in the requirements helps to improve the usability of the document. If natural language is a major cause of the problems in requirements specification, one approach to improving requirements is to eliminate the natural language and use a formal language like Z or SCR instead. [14]

4. Formal Methods

4.1 Introduction to Formal Methods

Formal methods are mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. [1] The goal of formal methods is to reveal ambiguity, incompleteness, and inconsistency in a system. This would improve our three quality benchmarks (accuracy, completeness, and usability). There are different degrees of formality that can be applied to the specification. At the most basic level, there is the creation of a formal specification, which follows some defined set of rules. At a more advanced level, the specification can be written using a mathematical notation or a modeling language so that theorem proving, model checking, or other safety measures can be used on them. Strict mathematical adherence also makes machine analysis of the specification possible. [3]

Formal specification languages can be very different from each other. The Formal Methods Wiki currently list 108 different formal methods that can be considered. [41] Choosing the correct formal method for a project is one of the most important steps in the specification process. [16] For a reactor project, DeJong et. al. analyzed three different specification languages, Z Notation, Prototype Verification System (PVS), and Statecharts. [42] This work will begin with a view of these formal method approaches since the three notations were popular, similar to other formal methods, and sufficiently distinct from each other.

4.2 Z Notation

Created by Jean-Raymond Abrial in 1974, Z notation has become a standard example in literature of a formal method. One key element of the Z notation is the use of mathematical data types for the data modeling. Predicate logic is used to describe the operations that need to be performed on this data. There are special notations for state changes and operations that do not change the state of the system. [7]

Z notation has been used successfully by a large number of companies on many projects. IBM is a notable example, using Z notation for part of their Customer Information Control System, which was an online transaction processing system. IBM declared that throughout the development process there was an overall improvement in the quality of the product, a reduction in the number of errors discovered, and earlier detection of errors found in the process. IBM also estimated a 9% reduction in the total development cost of the new release. [4]

One of the most common complaints about Z notation is readability. [13] Unless someone has a strong background in mathematics, specifications written in Z are practically impossible to read. This has led some groups to write two specifications, one in plain English and another in Z, but this creates duplication of work and introduces possible mismatches between the two specifications. Readability affects the ease with which a specification can be reviewed, ultimately limiting usability. One way to handle the readability issues of notations like Z notation is to produce visualization aids. [13] These visualizations tend to be manually created but may be automatically generated under for certain kinds of specifications. State diagrams and decision trees are examples of visual aids suggested by Dulac et. al. [13]

Z Notation demonstrates the advantages and feasibility of using formal methods in real world applications. Z Notation also provides a means of writing specifications that can be automatically

verified using theorem provers. [47] Zafar even suggests compilation of specifications using Z Notation. [46]

Consider the following requirement:

Requirements Example 1 – Natural Language

On Right Atrial Noise Detection, when
the current pacing mode = DDD,
the Patient Notifier Prevent Criteria shall be met.

Conversion to Z Notion would require a schema be created. Above the line there are variable definitions and under the line is the schema body. Here is an example.

Requirements Example 1 – Z Notation

```
Switch == {OFF, ON}
PacingMode == {DDD, DDI, DVI, DOO, VDD, VVI, VOO, AAI, AOO, OOO}

----Right Atrial Noise Detection--------------------------------------
|  RANoise            : Switch
|  CurrentMode        : PacingMode
|  PNPreventCriteria  : Switch
|---------------------
|  ( RANoise == ON) ^ (CurrentMode == DDD) ==>   PNPreventCriteria = ON
-------------------------------------------------------------------------
```

Obviously, this schema is incomplete. There would need to be a way of PNPreventCriteria to do switched to OFF, but that is handled in a different requirement. Conversion of the current requirements to Z Notation would be a very lengthy process due to the stark difference between the natural language and the Z Notation's highly mathematical specification style.

4.3 Prototype Verification System

Prototype Verification System (PVS) is considerably different from Z notation. PVS is a system for writing specifications and constructing proofs, based on classical typed higher-order logic. [40] Stylistically, it looks more like pseudocode than mathematical notation. Strong typing is required to keep the higher-order logic consistent. [40]

Here is an example of the requirement being translated into PVS.

Requirements Example 1 – Prototype Verification System

```
RANoise            : TYPE = {OFF, ON}
CurrentMode        : TYPE = {DDD, DDI, DVI, DOO, VDD, VVI, VOO, AAI, AOO, OOO}
PNPreventCriteria  : TYPE = {OFF, ON}

IF RANoise == ON AND CurrentMode == DDD
THEN PNPreventCriteria := ON
ENDIF
```

Compared to PVS has the advantage of familiarity; software engineers frequently use pseudocode. It does not require as strong of a mathematical background as Z, so it more usable to a greater number of engineers.

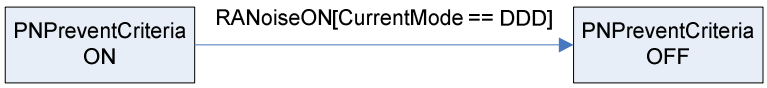
4.4 Finite State Machines

David Harel's Statecharts takes a radically different approach to formal specification, in that it uses finite state machines. [9]

This is a more lightweight use of formal methods. Baber, Parnas, and Vilkomir used a tabular representation with states on the vertical axis and inputs on horizontal axis to create a table of next states. David Harel's Statecharts introduces hierarchical nested states which help to improve the decomposition of the problem space. This greatly aids the readability, and therefore comprehension of the state diagrams. The graphical nature of Statecharts sharply contrasts the algebraic form of Z notation or the pseudocode of PVS. State diagrams in Unified Modeling Language are based on Harel's state chart notation. [26]

Here is an example of the requirement being translated into Statecharts.

Requirements Example 1 – Statecharts



The Traffic Collision Avoidance System II (TCAS II) project is an example of using state machine based formal methods, using Requirements State Machine Markup Language, which is also based on Finite State Machines. [5] The specification process discovered a number of errors in

the original TCAS that had not been previously found. The use of formal methods on this project allowed a substantial number of automated tests to be written and run against the specification to ensure its mathematical completeness and consistency. [5]

The Finite State Machine approach offers considerable contribution to the formal methods approach. First, finite state machines are easy to understand. Second, if the system is event-based, the Finite State Machines approach is very intuitive. Complications with using Finite State Machines may arise if the system is not event-driven however.

4.5 Combinations of Formal Methods

One complication with using formal methods is that not every formal method is well suited for each type of specification. [16] For example, one specification language might work well for mathematically intensive systems but not work as well for timing interactions. To properly use formal methods, an organization should use the appropriate formal method for the right application. This can be difficult to do because of the training and experience necessary to proficiently write documents using these methods. Also, complex systems may have different subsystems that have different properties. The pacing engine of an ICD is timing oriented, while the morphology features are highly computational. The varied nature of complex systems has led to a greater use of hybrid formal methods such as CSP-OZ and temporal B. [16] Of course, more complex and multifaceted formal methods increase the training and expertise demands even more.

Name	Advantages	Disadvantages
Z	State based, compact	Difficult to read, not all requirements are state based, requires strong math background, lacks timing
PVS	Easier to read	Requires strong math/proofs background
Statecharts	Easy to read, provides good visualization, easier conversion than most formal languages	Not all requirements are state based
CSP-OZ	Adds timing to Z	Difficult to read, requires strong math background

Table 1: Comparison of Formal Methods

4.6 Lightweight Formal Methods

Another approach to formal methods is to apply them in more specific instances and with simpler notation. This is called lightweight formal methods. [24] Daniel Jackson and Jeannette Wing state that formal methods will not be widely used unless the cost of specification and analysis can be greatly reduced. To accomplish this, they propose that formal methods be applied partially, instead of completely. They outline four types: partiality in language, partiality in modeling, partiality in analysis, and partiality in composition. This means that lightweight formal methods can be used on sections of a system, focused on certain attributes of that part, without full mathematical representation, with a focus on detecting errors instead of obtaining mathematical proof.

Given the time that would be needed to completely rewrite a complete requirements specification, a partial approach is probably more realistic than full translation into Z Notation, PVS, or even Statecharts. A critical part of the software system could be translated or certain elements of these specification languages could be applied to the specification. Additions to the requirements could also use partial formal methods, gradually moving the specification towards formalization.

5. Automation of Requirements Analysis

5.1 Computer Aided Software Engineering

Another aspect of the requirements writing process that should be considered is Computer Aided Software Engineering (CASE). CASE involves the use of software tools to write requirements, design software, and/or implement the software based on the design. [32] This can potentially help with the readability issue of formal methods while providing the constraints necessary to formalize a requirements specification. The software application can be used to define the inputs, outputs, events, conditions, states, and actions within a given system. The software can then provide different views based on these elements and even run various consistency checks. One example of this kind of software is IBM Rational Requirements Composer. [15] Large portions of a design can be directly derived from a formal specification. IBM Rational Rhapsody ties into Requirements Composer to create UML designs and executable prototypes based on the specification. [20]

There are costs associated with using CASE. One obvious issue is the monetary cost of the system. For example, the price of the IBM Rational tools vary based on the size of the development group, and the specific needs of the organization. Another issue is the time needed to become proficient at using the tool. Another considerable drawback of using CASE is that the organization becomes dependent on the tools to do its work. Migration from a given platform is costly. Depending on the migration, the requirements may have to been rewritten from scratch.

5.2 Automated Requirements Checking

In “Automated Consistency Checking of Requirements Specifications”, Heitmeyer, Jeffords, and Labaw outline a number of automated tests that can be performed on a requirements specification. These include tests for proper syntax, type correctness, variable completeness, initial values, reachability, disjointness, coverage, and lack of circularity. Syntax checking and type correctness are rudimentary tests that are usually run before any other tests. If the specification does not match the expected format, any other automated test will fail. Similarly, the test for completeness ensures that the variables and modes are defined while the initial value check ensures that these variables are initialized. The reachability test ensures that all conditions can potentially be met and all mode transitions in the requirements can be executed. The disjointness check ensures that each of the events, conditions, and mode transitions in the requirements are unique. If the conditions are not unique, then there are either redundancies or errors in the requirements specification. Similarly, the coverage check ensures that each variable is defined for its entire domain. In other words, there are no potential values for the variables that are not handled in the requirements. The circularity check ensures that there are no logical loops in the specification. [14]

Test Name	Description	Requires
proper syntax	Verifies that the text matches syntax rules	Syntax rules
initial values	Verifies that the type definition of each variable matches its use	Variable definition rules
variable completeness	Verifies that the variable always has a defined value	Initial values, syntax rules
type correctness	Verifies that the type definition of each variable matches its use	Syntax rules, type rules
reachability	Verifies that each requirement can be reached from the initial state	Initial values, syntax rules, type correctness
disjointness	Verifies that each state, variable, and term is defined uniquely	Initial values, syntax rules, type correctness
coverage	Verifies that each variable in a condition table is defined in the domain	Initial values, syntax rules, type correctness, reachability
circularity	Verifies that requirement loops do not exist	Initial values, syntax rules, type correctness, free from disjointness, reachability

Table 2: Automated Tests

One technique that can be employed to help improve requirements accuracy is requirements animation, where an executable version of the requirements model is generated. This executable responds to external events and user inputs and simulates outputs and state changes in system. This can help the stakeholders see issues even before official development begins. [12] The IBM Rational Suite is an example of this, where the Rhapsody design tool can directly import objects created in Requirements Composer and perform model checking. [20]

There are also algorithms that have been developed that can verify decision procedures. These algorithms will calculate the Boolean expressions and basic arithmetic in a requirement and determine whether the expression is can be satisfied within the domain. Some of these algorithms can even operate on arrays, within certain constraints. [38]

6. Natural Language Requirements

6.1 Analyzing Natural Language Requirements

Despite the apparent advantages of formal languages and CASE, many organizations choose to restrict the requirements documentation to natural language. In fact, most companies do not employ formal languages, and those that do often use the formal language for a small percentage of the overall specification, as Jackson and Wing suggested with partiality of language in the lightweight formal methods concept. If one is restricted to natural language for specification, what methods exist for ensuring the quality of the requirements documentation? In the literature, there are two major approaches: controlling natural language and processing natural language.

One high-level approach can be found that incorporates both approaches. In Lightweight Validation of Natural Language Requirements, the authors lay out an 8 steps for verification of requirements as follows:

1. Defining a style, a structure, and a language for the requirements document.
2. Selecting desirable properties to check.
3. Defining one or more models against which the properties selected in the previous steps can be checked.
4. Pre-processing the requirements document.
5. Parsing the natural language text of the requirements.
6. Building the models.
7. Checking that the models satisfy chosen properties.
8. Evaluating findings and revising the requirements specification accordingly.

The authors call steps 1-3 the setup phase since it defines the format of the document and the tests to be performed. The rest of the list is the production phase, where the verification of the

requirements documentation is performed. [17] There are many different ways of writing a specification, but if it is well-defined, an engineer should be able to write a parser than can organize the requirements into data structures and perform some level of testing on that data. There may exist problems performing this task successfully though. The specification may not be clear enough or the parser may not be refined enough. Step 8 is for evaluating the results. If the results are not acceptable, the specification, parser, models, and/or checks can be rewritten and the verifier can reiterate through the production phase. Most of the other literature deals with techniques and challenges associated with various steps of this process, but Lightweight Validation of Natural Language Requirements gave the most clear and general roadmap of the examined papers.

6.2 Types of Ambiguity in Natural Language

Much of the literature that focuses on natural language requirements attempt to deal with the issue of ambiguity in one way or another. Kamsties, Berry, and Paech suggest there are two types of ambiguity: linguistic ambiguity and requirement engineering ambiguity. [28] They define linguistic ambiguity as context-independent, whereas requirements engineering ambiguities are context dependent and therefore only discernible by someone who has knowledge in a given field. They provide an example of a context dependent requirement engineering ambiguity where the requirements check if the water level in a tank is above 100 meters for 4 seconds. The software engineers assumed that meant that the minimum water level was above 100 meters for 4 seconds, when in fact it meant that the root mean square was over 100 meters. Among the civil engineers, this measurement technique was common knowledge, but the software developers were not aware of this standard. According to the authors, requirements engineering errors tend to contribute to much more serious problems in the final product. For all practical purposes, it is impossible to find errors like these using automated techniques. Therefore, the authors suggest

that the specification process include a checklist which experts will use to weed out ambiguity in the requirements.

The research helps to define the types of ambiguity that can and cannot be checked using automated techniques. Context-dependent errors cannot be easily avoided because a person who is not a subject matter expert sees the requirement as being clear. The machine verifier would do the same without very special context sensitive programming. One technique is to be aware of words and phrases that can have ambiguous meaning. In the example above, the word “average” is an example. In cardiology, the phrase “cycle” is similar, since its meaning can vary depending on the application. One usually finds context-dependent errors by experience and therefore can avoid them by avoiding the offending phrases.

6.3 Reducing Linguistic Ambiguity using Controlled Language

Linguistic ambiguity, however, can potentially be addressed by controlling natural language to avoid known ambiguous terms. Rules can also be applied to the wording of the document. One approach is to restrict the wording of the documentation to a subset of English language and grammar (or any other natural language). This can be helpful for improving clarity, therefore reducing ambiguity. It can also help to facilitate automated analysis. One example of a controlled language is Attempto Controlled English (ACE) [27]. The most obvious and significant advantage to using an ontology like ACE, as opposed to the formal methods described in the Section 4, is readability. Unlike many other natural language approaches, ACE is a formal language and as such, a variety of computerized checks can be run against requirements written in ACE. You can also define new relationships and operations easily. One potential problem with the use of an near-natural language is that it is easy for the authors to write something that makes sense in English, but does not fit the formalized grammar. Violations of the controlled language are not

always obvious, but fortunately, there are parsers that can detect errors in syntax and semantics, based on a definitions file. Simplified Technical English is another controlled language which focuses a great deal on simplifying the sentence structure and word choice. The rules for Simplified Technical English include: restricting the length of noun phrases, restricting sentence length, avoiding gerunds, using articles before nouns whenever possible, writing sequential steps as separate sentences, and placing conditions first in sentences. [36] 16% of requirements specifications use a form of controlled natural language. [22]

6.4 Translation Techniques

Leonid Kof explores the feasibility of using natural language processing on a requirements document. [30] He analyzes a number of different techniques for building an ontology based on a requirements specification. These techniques include term extraction, term identification, clustering and taxonomy building. Term extraction involves the identification of noun phrases, verbs, etc. Term clustering defines related concepts, such as ventricular pacing and atrial pacing. Taxomony building expands on this concept by creating larger sets based on the intersections or the clusters. Each sentence has object to object and object to verb relationships that can be considered for building a data set. His paper claims that by integrating these methods, natural language processing can be done on the requirements specification, assuming a moderate amount of manual intervention is used also.

In a different paper, Leonid Kof suggests translating natural language specifications into message sequence charts. [29] They separate the requirements into subject and predicate phrases using a parts of speech tagger and then create messages from the two parts of the sentence. Once all of the requirements are translated into messages, any messages that are missing from the flow of messages are interpreted as missing requirements. One major problem with this translation

method is that the requirements might need to be rewritten. The authors describe that many of the sentences in their specification had to be changed from passive voice to active voice for the translation to occur properly.

Goldin and Berry propose a method for finding abstractions in a requirements document. [19] This method involves using automated textual analysis of a requirements document to find its key concepts. The paper focuses on how the tool can be used for requirements elicitation, but it can also be used to identify the phrases and actions in a requirements document so that they are more easily parsed into a data structure. This method is not automatic though. The original document must be written with certain properties. Pronouns must be replaced with their corresponding nouns, for example. Like many natural language processing methods, this approach involves editing of the original text.

6.5 Grammatical Parsing

One approach to analyzing the requirements documentation is to use a grammatical parser. Historically, the problem with this approach is that grammatical parsers were not always accurate, especially in documentation with a variety of technical terms. [19] There are two significant ways in which this problem presents itself. First, it may confuse noun and verbs. For example, unlike standard literature, the term “pacing” is usually a noun in cardiology.

Lately, there have been significant advancements in probabilistic context-free grammars (PCFGs). One example is The Stanford Parser from The Stanford Natural Language Processing Group. [18] They use unlexicalized PCFGs to parse sentence structure on a series of sentences from the Penn Treebank. While the authors acknowledge the advantages of using lexical data, their approach offered a number of advantages including time efficiency and space efficiency. [34]

Their approach had 86.36% accuracy when the sentence length is less than or equal to 40 words. As expected, the results rapidly deteriorate if the sentence is longer. Based on the documentation of the Stanford Parser, sentences greater than or equal to 70 words usually cannot be processed on a desktop computer. Personal experimentation confirmed this warning was accurate.

6.6 Existing Natural Language Translator

Gervasi and Zowghi propose a translator that can take natural language requirements, translate them into a more formal version, and run model checks against them. Like the Attempto Controlled English approach, the result is a formalized natural language specification. The authors do not claim the same depth of model checking available for mathematically notated formal methods like Z, but they illustrate how inconsistency checking can be performed. [21] The translated version may be incorrect due to ambiguity in the original requirements (or incorrect parsing). It would have to be verified by the clients. If the clients disapproved of the translation, another round of specification writing, translation, and testing would have to be performed. Like the Stanford Parser, Gervasi's translator also had a 70 word limit on sentences and a reduction in accuracy for sentences longer than 40 words.

6.7 Domain Ontologies

Despite advances in natural language processing, it is still difficult to process a requirements specification sufficiently by machine. Kaiya and Saeki propose a domain ontology technique where the ontology contains a thesaurus and inference rules. [23] The thesaurus contains the domain specific concepts and relationships. The ontology is visualized as a class diagram in the

paper that treats objects, functions, quality attributes, constraints, etc. as classes. The relationships in the diagram are ways in which the different classes interact with each other. These relationships include: apply, require, antonym, synonym, aggregate, cause, etc. For example, in a music player, decode is a function applied to a music file object. They claim to detect incompleteness and inconsistency in the requirements specification, measure the quality of the specification with respect to its meaning, and predict requirements changes based on semantic analysis of the change history. The second two claims are beyond the scope of this thesis, but the first claim is highly relevant. Unfortunately, their definitions of completeness and correctness are problematic. Because of these definitions, a requirements document is inconsistent if its completeness metric is 100% due to their application of contradict relationships. While their checks were of questionable utility, the use of class diagrams to represent ontologies was very interesting.

6.8 Summary Table for Related Work

Technique	Contribution	Fits within Budget (5 = cheapest)	Applicable to Group (5 = most applicable)	Applicable to Solution (5 = highest)
Formal Methods	Concept of formally checking requirements	2	3	2
Z Notation	Successful example of formal method with highly mathematical notation	1	1	1
Prototype Verification System	Successful example of formal method with pseudocode notation with automated checking	2	3	1
Statecharts	Graphical representation of formal methods based on Finite State Machines	2	3	1
Combinations of Formal Methods	Fills in the deficiency of individual formal methods by combining them	1	1	1
Lightweight Formal Methods	Reduces workload by using formal methods partially	3	4	3
CASE	Aids the writing and testing of requirements	2	2	2
Lightweight Validation of Natural Language Requirements	Provides an eight step plan for testing requirements	5	4	5
Attempto Controlled English	Example of a controlled language is qualifies as a lightweight formal specification	3	3	2
Translation Processes of Kof, Goldin and Berry	Examples of translation of natural language into different data structures	4	3	4
Stanford Parser and Gervasi's Translator	Examples of software that can determine the structure of a sentence and the parts of speech of each word.	4	3	4
Domain Ontologies	A system of identifying the relationships between terms to determine their meaning more accurately.	3	3	3

Table 3: Applicability of Techniques from Literature

7. Software Requirements Specification Analysis

The Software Requirements Specification (SRS) used for this work is compliant with IEEE Std 803. [35] The sections of the Functional Requirements are divided into functional components called features, such as Right Ventricular AutoCapture (which regulates ventricular pacing amplitude) and Rate Adaptive Pacing (which regulates the pacing rate based on physical activity). Some of these features run continuously and others are triggered on a condition or timer expiration and run for a short period of time. While there is coupling between certain features, the goal is to make each feature as independent as possible.

Within each feature, there is an overview, followed by a list of defined parameters, followed by the functional requirements for the feature. The overview is written in paragraph form and is meant to give the developers and verifiers the general idea of what the feature will do. There should be a functional requirement corresponding to all of the data in the overview, therefore strictly speaking, the overview is extraneous but is a useful educational tool for the engineers that will work on the feature. [44]

The defined parameters list the name, range, and resolution of the parameters that will be used in the feature. This is not intended to be a complete list of variables, as that would impose design into the requirements. Instead, it is ideally used to define the data elements that external instruments will load or store. Since two different parts of the larger system must interact, that interaction is clearly defined and part of the requirements themselves. Programmable parameters, the status block, and alerts block are examples of defined parameters.

The functional requirements define the behavior of the device. The majority of the functional requirements are event based. The most significant events are the cardiac events, which can be sensed (originating from the heart itself) or paced (caused by the device). These events are also

categorized by the four chambers of the heart, most commonly the right atrium or right ventricle. Other events can be triggered by external factors such as noise detection, magnet detection, or telemetry.

The event based requirements usually have the following structure:

On [event occurring],

When [set of conditions are met],

The device software shall [perform a set of actions].

There are some variations on the structure such as:

When [set of conditions are met],

The device software shall [perform a set of actions].

or

When [event occurs] and [set of conditions are met],

The device software shall [perform a set of actions].

Overall, the event based requirements can be patterned using a small set of structures. Given the order intrinsic to the requirements, it seems plausible that data can be extracted from them. If enough data could be extracted, automated testing of the requirements might be possible.

8. Original Work

8.1 Examination of Translation into a Formal Language

Given that this thesis was based on a real SRS and intended to provide real benefits to the company using it, certain constraints were taken into consideration. The most significant constraint was the time of the software engineers. While the engineers expressed interest in ideas to improve the SRS, they did not have the necessary time to perform a major rewrite of the requirements by hand or engage in lengthy training sessions. This constraint prohibited the use of the Z Notation due to the training time that would be required. The engineers did not need excessive training for PVS or Statecharts, but the time to rewrite and review the requirements would be excessive. The current SRS is written in plain English with loose stylistic rules. To adopt any of the formal specification languages from Section 4, the company would need to rewrite the SRS, which contains over a thousand pages. The management considered the value gained from a rewrite of the requirements to be less than initial cost. [48] If a formal language was to be adopted, Daniel Jackson and Jeannette Wing's approach, using a partial solution, would have to be considered.

The Finite State Machine approach used by Statecharts looked promising as a potential language to translate the requirements into. State diagrams were commonly used among the software engineers within the company, so little training would be required compared to Z Notation. As a prototype, a proposal was made to translate some of the requirements into a tabular structure with states on one axis, inputs on the other axis, and the resultant actions to be performed in the cells of the tables. This proposal was met with some excitement among developers. The idea was quickly rejected by the requirements engineers though, because it would be more difficult to maintain the requirements document from project to project, since seemingly minor changes could have a large impact on the table. [49] This issue is caused by the fact that DOORS, the tool

the requirements groups used to capture requirements software, treats every element of a table as a separate requirement and does not display the project identifiers. The only way to change elements of the table for a new project is to create a new table. Under the current system, changing a requirement for a new version of the firmware requires creating a new requirement. If a tabular requirements format like Statecharts was adopted, changing a requirement would require adding a table full of requirements, because the entire table would need to be copied. Not only is this a maintenance issue for the requirements team, but it complicates the workflow of the verification team. All tests are tied to requirement identification numbers, so a single requirement change would require them to update the requirement IDs for every test that touches that table. While the team was not philosophically opposed to the proposal, the disruption of workflow was not acceptable.

It would take less work to translate the requirements into PVS than either of the other 2 formal specification languages analyzed. Unfortunately, the idea of using pseudocode for requirements had already been rejected by the engineers. Previous attempts to use pseudocode for requirements specification had resulted in strict constraints on the design, since any deviation from the form presented in the pseudocode in the requirements was treated like a design defect by the verifiers. Given previous complications with pseudocode, no one in the department was willing to try PVS. [50]

8.2 Direction of Thesis

The complications involved with training, translation, maintenance, and adoption of formal languages limited to format of the SRS to a form of natural language. Based on the techniques described in the previous section, it seemed feasible that some of the benefits of automated requirements checking can be discovered without completely rewriting the Unity SRS. An attempt

could be made to translate the requirements from the current English language prose into data structures upon which some verification checks could be performed. This activity would have benefits whether the attempt succeeded or not. If the attempt was successful or even mostly successful, tests could be run on the content in the data structures, resulting in a measurable and significant improvement in the quality of the requirements. These tests could find issues in the requirements such as circularity and disjointness. If the attempt was not successful, it could potentially uncover problems in the SRS. The requirements that could not be translated properly will likely be candidates for rewriting, due to errors or stylistic irregularities. This would still have a positive impact on the requirements specification because it would reveal issues in the documentation.

The approach that will be used is to combine and adapt several techniques from the literature. First, the approach used in Lightweight Validation of Natural Language Requirements by Gervasi and Nuseibeh will serve as a framework for exploration of the SRS. Kof, Goldin, and Berry suggested multiple ways to translate natural language into data structures. The method used in this thesis is different, but was inspired by their work. Finally, the natural language processing techniques used by the Stanford Parser will be adapted to create symbols from the SRS in a way that reduces false positives in the translation.

8.3 Success Measurement

There are various measures of success that can be applied to this activity, each related to the different activities. First, the translation process itself can be roughly gauged by the percentage of requirements and features that can be translated in data structures correctly. While this metric is important, it does not differentiate between mistranslated requirements and requirements that could not be translated. It would be preferable to warn the user that translation was impossible

than to blindly translate the requirements incorrectly. Because of this, the second success criteria measurement for this thesis is the proper identification of requirements which can or cannot be properly translated, as that would aid in the identification of problematic requirements and reduce errors in the data structure. Since the principal goal of requirement analysis is reduction of errors, another measure that will be used is identification of errors is the SRS.

The expectations for the parser are described in the following table.

Expectation Name	Description	Reason for Choosing
Complete Translation of One Feature	At least one feature will be completely translated.	Demonstrates that whole features can be translated using this technique.
Identification of Translation Errors	The parser will correctly identify requirements which cannot be translated properly.	Demonstrates that the parser will not perform translations that the rule set is not programmed to handle.
75% Identification	The parser will be able to correctly identify 75% events, conditions and actions in the event-based requirements.	Demonstrates the percentage of requirements that can be translated without a rewrite.

Table 4: Project Expectations

8.4 Applying the 8 Steps for Verification of Requirements

In “Lightweight Validation of Natural Language Requirements”, Gervasi and Nuseibeh lay out a framework for verifying requirements. Following are the 8 steps of their framework.

1. Defining a style, a structure, and a language for the requirements document.
2. Selecting desirable properties to check.
3. Defining one or more models against which the properties selected in the previous steps can be checked.

4. Pre-processing the requirements document.
5. Parsing the natural language text of the requirements.
6. Building the models.
7. Checking that the models satisfy chosen properties.
8. Evaluating findings and revising the requirements specification accordingly.

In this section, the application of this framework to the problem of verifying the SRS will be demonstrated.

8.4.1 Defining a style, structure, and language for the document

The first step from Gervasi and Nuseibeh's framework in Lightweight Validation of Natural Language Requirements is "Defining a style, a structure, and a language for the requirements document". This step is simple because the requirements are already written. Since one of the constraints of the work is to parse the requirements document with minimal manual rework, the SRS will be used as is. Minor stylistic and structural changes will be considered over the course of the thesis work, however the specification itself is complete.

8.4.2 Selecting desirable properties to check

The second step from Gervasi and Nuseibeh's framework is "Selecting desirable properties to check". Many of the tests mentioned in Section 5.2 would be difficult to implement because of the nature of the requirements. Completeness checks would be difficult because the requirements do not expressly spell out the do-nothing states in the specification; therefore a missing requirement is usually indistinguishable from a do-nothing condition. Also, initial values are not spelled out in the requirements and data elements are not strongly typed. Because of these constraints, it

seems reasonable to focus on what can be tested. Based on the model chosen, basic model checking can be produced. Also, a partial circularity test could be performed given the event-based nature of the requirements. Full circularity tests usually require strong typing, but most conditionals in the SRS are tied to Booleans and enumerations. Other tests can be left for subsequent versions of the parser and the specification.

8.4.3 Defining models

The third step from Gervasi and Nuseibeh's framework is "Defining one or more models against which the properties selected in the previous steps can be checked". To perform circularity checks, a tree must be formed based on the events and the resultant set of actions. When an event occurs, there is a set of requirements that are potentially activated by that event. The conditions of each of those requirements are examined and if those conditions are met, the actions associated with that requirement are performed. Those actions then can create events of their own, continuing the cycle. This can be represented by a red-black tree of alternating events with conditions, and actions on each layer. There would be a special starting node, representing an input into the system.

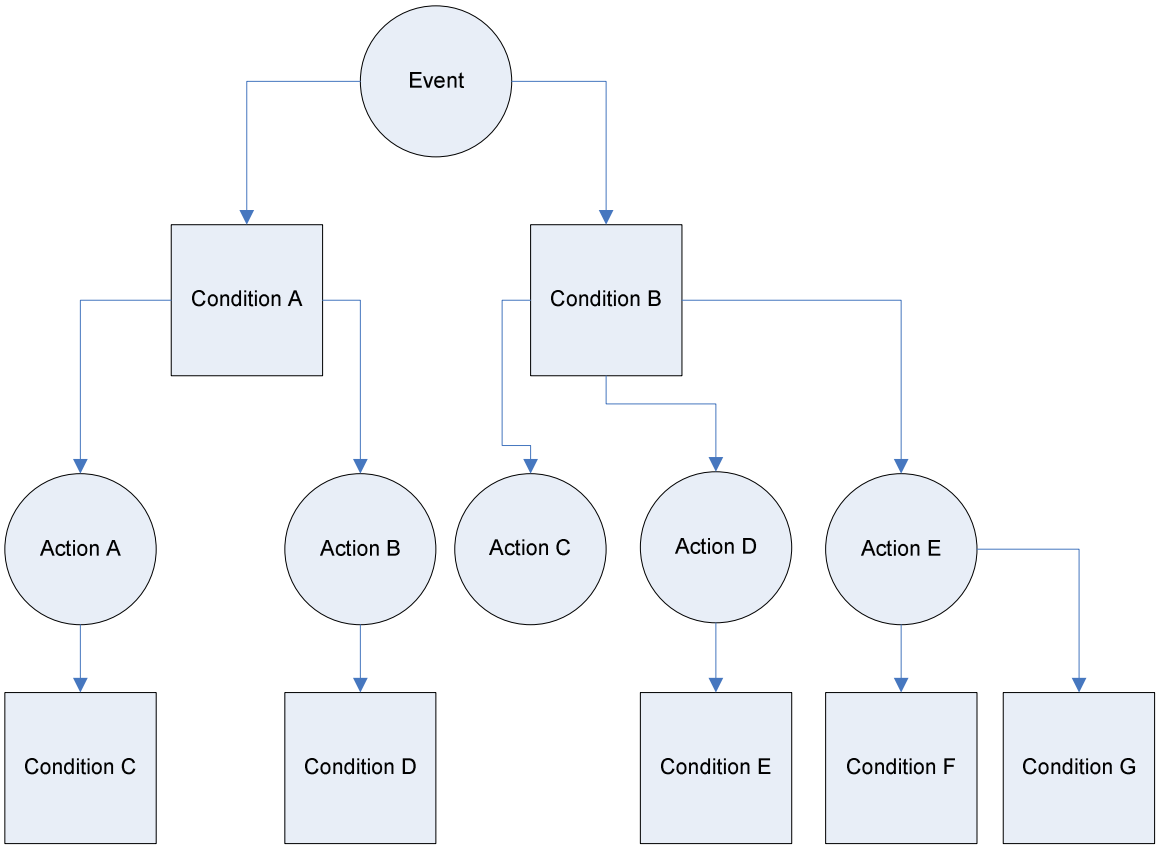


Figure 1: Red-Black Tree of Conditions and Actions Caused by an Event

To identify the potential starting nodes and terminal nodes, the inputs and outputs of the overall system can be identified. In a pacemaker or ICD, the outputs include pacing, shock (ICD only), diagnostic data, clinical alerts, telemetry, and patient notification. The inputs for the device are numerous. Some examples are intrinsic events (the patient's natural cardiac activity), the activity sensor, telemetry, magnet detection, and timer expiration. In the specified firmware, all of these inputs are handled by interrupts.

Identification of inputs and outputs can also be performed on the feature level. Given the pervasive use of the feature concept in the requirements, one approach would be to analyze each feature as an individual module with interactions with hardware and other features can be treated as inputs and outputs of that module. Generally speaking, that is the way they are implemented

as well. There are some benefits to this. First, the scope is much more manageable. Testing model compliance on a set of 100 requirements is much more feasible than testing it on tens of thousands. Secondly, the style and clarity of the requirements are highly variable throughout the SRS. Lastly, the computational time needed to perform checks is much more manageable if the system is decomposed in over 100 pieces. One drawback of this attempt is that it does not capture the entire system unless an integration step where the modules are linked together is added into the analysis.

After the inputs and outputs of the system are codified, internal variables and operations can be identified and a data structure can be defined to represent it. Once the data representation becomes clear, translation rules can be derived to put the instances of each concept and their causal relationships into a data structure. If it was impossible to codify certain concepts or requirements, that should indicate a defect in the requirements, most likely ambiguity or incompleteness.

One alternative to codifying the inputs and outputs before the analysis is to perform an analysis on the data elements in the requirements and identify the elements that are not both written to and read from. If an element is read from but not written to, it can be treated as an input. If an element is written to but not read from, it can be treated as an output. If an element is not used at all, there is an extraneous element in the requirements, which would represent an unnecessary data element requirements or a missing functional requirement.

The modeling approach taken is a hybrid between these techniques. Known system inputs are identified at the start, including pacing, shocking, telemetry command, and interrupts. Various known operations are identified as well, including assignments, mathematical operations, liens, terminations, and requests. The exact format of these data structures would be dependent on further analysis. The outputs are simply terminal operations; therefore special treatment should be unnecessary.

8.4.4 Preprocessing the SRS

The fourth step from Gervasi and Nuseibeh's framework was "Pre-processing the requirements document". This process started simply and would become more complicated through multiple iterations. The first step was to retrieve the requirements which were housed in the DOORS application. Two approaches were considered. The first approach was to use the DOORS API, which supports Visual Basic and Perl interfaces. The second approach was to export the database as a Comma Separated Values file. Either approach is valid. The CSV approach was taken for multiple reasons. First, the system would be more portable, since the work was not reliant on the DOORS Server being available or online. Second, changes could be made within impacting the real requirements. This allowed greater testability of the system. In the future, the CSV interface could be replaced with the API.

The next step in the preprocessing was to eliminate requirements that are not event-based. There were a number of different types of non-event-based requirements including headings, data definitions, constraint requirements, and other non-event based requirements. The headings were straightforward since they start with an outline number as per IEEE Std 803. The other requirements all had something in common. They did not begin with the words one would expect from an event-based requirement, such as "on", "when", "while", "if", etc. By filtering on those starting words, all of the non-event based requirements where eliminated.

There was a glitch in this system though. Requirement blocks sometimes span multiple requirement ids, as in Requirements Example 2 below.

Requirements Example 2

10001,5,"On the Patient Notifier Prevent Criteria being met, when

Patient Notifier is programmed ON, and

Patient Notifier Type X Y Alert = ON, where X ∈ {A, B} and Y ∈ {1, 2},

the device software shall:"

10002,6,"terminate Patient Notifier"

10003,6,"place a lien against Patient Notifier"

10005,6,"set Patient Notifier Type X Y Alert = OFF, where X ∈ {A, B} and Y ∈ {1, 2}"

The first comma separated value in each line represents the requirement id and the second value represents the object level. The raised object level in the last four lines indicated that they are the children of the first line (which is in return related to a series of headings indicating which feature the requirement belongs to). Although the last 4 requirements do not have an event in them, they are actions that are tied to the condition in requirement 10001 and should not be purged. This style property of the SRS necessitated a revision to the preprocessor where the event-based requirement blocks would be grouped together, preventing these requirements from being ignored.

8.4.5 Parsing the natural language text of the requirements

The fifth step from Gervasi and Nuseibeh's framework was "Parsing the natural language text of the requirements". To accomplish this, a much greater level of analysis was needed than the initial observations in the literature review. The requirements contained in the SRS did not follow strict stylistic rules, but a loose set of guidelines. Because of this, each translation rule that is created has to be thoroughly tested. There is effectively little separation of the analysis of the

requirements and the writing of the rules for the parser. Unexpected results lead to the revision of the rule or the exceptions have to be noted, so that the requirements can be rewritten. This was a very iterative process of manually analyzing the requirements, creating a rule, programming the rule, analyzing the execution results. If the results were not expected, the process starts over again. The details of this process will be covered in section 8.5 through 8.9.

8.4.6 Building the Models

The sixth step from Gervasi and Nuseibeh's framework was "Building the models". Section 8.10 will describe this process in detail.

8.4.7 Checking that the models satisfy chosen properties

The seventh step from Gervasi and Nuseibeh's framework was "Checking that the models satisfy chosen properties". Section 8.11 will describe this process.

8.4.8 Evaluating findings and revising the requirements specification

The eighth step from Gervasi and Nuseibeh's framework was "Evaluating findings and revising the requirements specification accordingly". Sections 9 and the Conclusions will cover this part of the analysis.

8.5 High Level Decomposition of Requirements

8.5.1 Introduction

The fifth step from Gervasi and Nuseibeh's framework was "Parsing the natural language text of the requirements". To accomplish this, a much greater level of analysis was needed than the initial observations in the literature review. The requirements contained in the SRS did not follow strict stylistic rules, but a loose set of guidelines. Because of this, each translation rule that is created has to be thoroughly tested. There is effectively little separation of the analysis of the requirements and the writing of the rules for the parser. Unexpected results lead to the revision of the rule or the exceptions have to be noted, so that the requirements can be rewritten. This was a very iterative process of manually analyzing the requirements, creating a rule, programming the rule, analyzing the execution results. If the results were not expected, the process starts over again.

8.5.2 Isolating Actions from the Requirement Blocks

After the preprocessor has run, the resultant requirement blocks were typically of the following structure:

> On [event occurring],
> When [set of conditions are met],
> The device software shall [perform a set of actions].

Using this structure, a search was performed on all of the potential events. The first step was to

separate the actions from the rest of the requirement block. A simple split function on the phrase "device software shall" was able to accomplish this. For each requirement, the split was attempted, and requirements without the given phrase were saved. Most of the outlying requirements contained small variants such as "software shall" and "device shall". Some features referenced the feature name instead of the device software. Criteria definitions, which will be discussed in greater detail later, also had a different format. Extra preprocessing steps were implemented to handle these variations.

8.5.3 Handling Events and Conditions

After isolating the actions, the next step was to handle the events and conditions. One of the first discoveries was that events and conditions were used interchangeably. A condition being met was treated like an event and events were sometimes in the condition set. An example of this can be found in Requirement Example 3.

> Requirements Example 3 – Noise
>
> When noise is detected,
> the Noise Detection Criteria shall be met.

Problem: Events, Conditions, and Persistence

In Requirements Example 3, it is difficult to determine whether this is an event or a condition. The initial approach was to treat events and conditions interchangeably in the translation as well, since that seemed to be the intention of the authors of the SRS. This introduced a problem with

timing though. For example, if an action is to be performed on a particular cardiac event that action should occur again if there is another cardiac event and the conditions have not changed. However, if the condition is met, the action should be performed the first time, but not necessarily every time the requirements say that the condition should be met. The difference between events and conditions is persistence.

The second approach was to treat events independently from conditions. This approach was problematic because there was no clear way to separate the event-based requirements from the condition-based requirements. If a requirement started with the word “on”, it was generally event-based. If a requirement started with the word “while” it was usually a persistent condition-based requirement. The word “when” was not clear though. At times it was used identically to “while”, but at other times was interchangeable with “on”.

Solution: Flags for Persistence

Since the requirements did not always define the level of persistence, a solution with maximum flexibility was desired. To accomplish this, conditions and events would be handled using the same structures, but those structures would include flags for persistence. It may be impossible to accurately populate those flags with the current specification and guidelines, but in the future, they could be properly updated and used in tests.

8.5.4 Isolating Individual Conditions and Actions

As mentioned previously, the first step is to isolate the action set, which is found in the text following the term “device software shall”. Consider Requirements Example 4.

Requirements Example 4

On the Patient Notifier Prevent Criteria being met, when
Patient Notifier is programmed ON, and
Patient Notifier Type A 1 Alert = ON,
the device software shall:

- terminate Patient Notifier
- place a lien against Patient Notifier
- set Patient Notifier Type A 1 Alert = OFF.

The condition set is everything before “the device software shall”, and the action set is everything after. Splitting the individual actions in the action set proved simple. In the vast majority of requirements, they are bulleted or use one action per requirement. Using this rule, it was easy to split the action block of this requirement block into three individual actions.

Splitting condition blocks was straightforward as well. The condition set could be split on the words “when” and “and” separate the individual conditions. This rule worked with most requirements, but there were notable exception cases. Requirements containing Set Notation and/or Criteria Flags required special preprocessing which will be explored in detail in Section 8.6. After the preprocessing, it was safe to use this rule for splitting condition blocks.

8.6 Literary Characteristics of the Requirements

Once the actions and conditions can be individually processed, the task of parsing the requirements text into models begins. To accomplish this, the data characteristics and the literary

characteristics of the requirements needed to be more closely analyzed. After each concept and its natural language representation was understood, translation rules can be created. The following subsections will explore some of the literary devices used in the document and examine approaches for translation and data representation.

8.6.1 Set Notation

One common element in the SRS is the use of set notation. Following is an example of set notation:

> Requirements Example 5
>
> On X Y Noise Detection, where X ∈ {Right, Left} and Y ∈ {Atrial, Ventricular}, when
>
> the current pacing mode ∈ {DDD, DDI},
>
> the Patient Notifier Prevent Criteria shall be met.

Problem: Use of set notation inconsistent

In this case, the requirement is shorthand for multiple requirements concerning various types of Noise Detection. The "element of" phrases in the top line perform term substitutions. If this was the consistent usage of the "element of" phrase, the parser could split the requirements that used the term into multiple requirements for each element of the set. Unfortunately, this usage is not consistent throughout the SRS. In the second line of the example, the "element of" operator denotes that the value of a variable is within the set.

Ideally, there would be some attribute in the requirements that would allow us to differentiate between the two meanings. In general, when a term substitution is meant, the format will be “where <term> is an element of <set>” whereas the value checks would not have a “where” at the beginning. Unfortunately, there were exceptions to this rule. There are places where both types of element of operations are used in a single requirement, making set notation parsing problematic. Without knowledge of the terms involved, it is impossible to automate the term substitutions and value checks. This creates some difficulties in the parsing since the "element of" phrase is used in 462 requirements. To continue with the parsing, the text surrounding the phrase "element of" was examined and a list was constructed of terms which were used in term substitutions. For example, the terms X and Y always led to term substitution.

Solution: Use preprocessor to split into separate requirements

Using the list of term substitutions, a customized preprocessor was created which replaced every requirement with term substitutions with a group of requirements with the replacement term explicitly stated. For example, Requirements Example 5 was split into the following four requirements:

Requirements Example 5-1

1) On Right Atrial Noise Detection, when

the current pacing mode $\in$ {DDD, DDI},

the Patient Notifier Prevent Criteria shall be met.

2) On Right Ventricular Noise Detection, when

the current pacing mode ∈ {DDD, DDI},

the Patient Notifier Prevent Criteria shall be met.

3) On Left Atrial Noise Detection, when

the current pacing mode ∈ {DDD, DDI},

the Patient Notifier Prevent Criteria shall be met.

4) On Left Ventricular Noise Detection, when

the current pacing mode ∈ {DDD, DDI},

the Patient Notifier Prevent Criteria shall be met.

This greatly simplified operations in the parser. Since all of the term substitutions were replaced, the parser could handle the ∈ operator as a comparison with multiple values.

8.6.2 Criteria Flags

In the requirements, it can become cumbersome to write out long condition lists if they are used frequently. To avoid this problem, the requirements engineers often define criteria flags. If a list of conditions are met, then the criteria is said to be met. Requirements Examples 2 and 5 demonstrate this:

Requirements Example 7 (Examples 2 & 5 with ∈ removed)

1) On X Y Noise Detection, where X ∈ {Right, Left} and Y ∈ {Atrial, Ventricular}, when

the current pacing mode ∈ {DDD, DDI},

the Patient Notifier Prevent Criteria shall be met.

2) On the Patient Notifier Prevent Criteria being met, when

Patient Notifier is programmed ON, and

Patient Notifier Type A 1 Alert = ON

the device software shall:

- terminate Patient Notifier
- place a lien against Patient Notifier
- set Patient Notifier Type A 1 Alert = OFF

3) On the Patient Notifier Prevent Criteria being met, when

Patient Notifier is programmed ON, and

Patient Notifier Type A 2 Alert = ON

the device software shall:

- terminate Patient Notifier
- place a lien against Patient Notifier
- set Patient Notifier Type A 2 Alert = OFF

4) On the Patient Notifier Prevent Criteria being met, when

Patient Notifier is programmed ON, and

Patient Notifier Type B 1 Alert = ON

the device software shall:

- terminate Patient Notifier
- place a lien against Patient Notifier
- set Patient Notifier Type B 1 Alert = OFF

5) On the Patient Notifier Prevent Criteria being met, when

Patient Notifier is programmed ON, and

Patient Notifier Type B 2 Alert = ON

the device software shall:

- terminate Patient Notifier
- place a lien against Patient Notifier
- set Patient Notifier Type B 2 Alert = OFF

Criteria flags are interesting for a number of reasons. First, there are not necessary. Theoretically, the preprocessor could store the conditions that compose criteria and replace instances of those criteria with the text of the conditions. The criteria flag requirements are not behavior requirements at all; they are definitions. It is possible to treat them as a state definition though, which simplifies the processing.

Problem: Deviation for style rules

One issue with the text of the criteria flag requirements deviate from the common use of the "shall", which is normally used to separate the conditions from action being performed. With criteria flag requirements, the action is "the [Name of Criteria] shall be met". Application of a split on the word "shall" would create an action set consisting of "be met", which does not make sense.

Solution: Standardization by preprocessing

This undesired deviation from the standard format can be fixed in preprocessing. Every place where "the [Name of Criteria] shall be met" is found, it is replaced with "the device software shall

meet the [Name of Criteria]". This standardizes the format, make it possible to use the method of splitting actions from conditions describe in the "Isolating Individual Conditions and Actions" section.

8.7 Inputs and Outputs

8.7.1 Sensed Events and Pacing Pulses

Some of the most important inputs for on ICD are sensed cardiac events, generally called "sensed events". These are the electric representation of the intrinsic heartbeat of the patient. For most pacing modes, if there is a sensed event in a given heart chamber, the ICD refrains from pacing in that chamber for that cardiac cycle. Also, many features rely on the occurrence of sensed events to trigger other actions. [39] In the requirements many of the events are based on sensed events, which are usually qualified with a set of adjectives such as "atrial", "ventricular", "brady", or "tachy". In the parsing of the sensed events, those modifiers must be saved in the data structure with the representation of the sensed events to ensure proper linkage. A domain ontology can be used to classify all of the different types of sensed events.

Pacing pulses are generally requested when the patient's heart does not pace within an expected amount of time. The software requests a pacing pulse for a given chamber of the heart with a set of characteristics including amplitude (the voltage of the pulse) and width (the amount of time for which the voltage will be applied). [39] There are similar qualifiers to pacing pulses as there are for sensed events and the data structure must save those values as well.

8.7.2 Arrhythmia and Therapy

The primary reason that an ICD is implanted is because the patient is at risk of a life-threatening arrhythmia. The ICD has a number of different steps it goes through to handle unsafe heart rhythms. The first step is the detection of the arrhythmia. This is a warning flag due to the sensed rate being too high. Depending on the configuration of the ICD, different algorithms will run to confirm how the ICD should respond to the arrhythmia. This is called diagnosis. Based on the diagnosis, the device chooses an appropriate therapy. This may entail variations in pacing, high voltage shock, or even nothing at all, depending on the diagnosis.

If high voltage shock is chosen, the firmware will request that the high voltage chip charge up. It takes time for the capacitor to build up enough charge to defibrillate the patient, which leaves some time for a change in therapy. At any time after the charge request, the software can abort the charge process and dump the excess charge. If therapy is not aborted, the software can deliver the shock.

To represent these decisions and actions in data structures the following concepts must be captured: detection, diagnosis, charge, dump, and delivery. Each must include characteristics such as the chosen therapy and the charge voltage.

8.7.3 Telemetry Commands

A common event in the requirements is the receipt of a telemetry command. For each telemetry command, certain actions are expected to be performed and a response protocol is expected to be followed. A typical example of a command response requirement is from the first part of Requirements Example 6 which states:

Requirements Example 6-1

On receipt of a Clear Diagnostics telemetry command, when

X is ongoing, where X ∈ {Ventricular Episode, Potential Ventricular Episode},

the device software shall set the Response Status to indicate a failure to complete the Clear Diagnostics telemetry command, including the reason for failure.

8.7.4 Programmable Parameters

The programmable parameters are essentially a configuration file that allows the doctor to control the high-level operation of the device. These parameters define the mode of operation of the device in the most fundamental way, including the pacing rate, the pacing mode, the pacing amplitude, the qualifiers for high voltage therapies, the EGM triggers, and the individual features modes. In the requirements, they are treated as a set of inputs which determine hundreds of attributes in the device. Fortunately, there is an external file which defines each of these parameters.

8.7.5 Diagnostics

Diagnostics are data sets that the device updates for the benefit of clinicians such as doctors and researchers. The requirements for updating diagnostics tend to be less detailed than other requirements. Some requirements simply state that the following data elements shall be maintained in a certain diagnostic. The algorithm is implied by the structure of the histogram or

diagnostic. Fortunately, this does not affect circularity tests since the diagnostics are a system output, which is a terminal node. It does however impact completeness checks, since it is difficult to verify that the data elements in the requirements are all used.

Problem: Stylistic differences in requirements with diagnostics

Diagnostics provide a challenge to the parser because the format of the diagnostics requirements does not comply with the way the parser splits action sets. Consider the following requirement block:

Requirements Example 9

On receipt of a Clear Diagnostics telemetry command, when

X is not ongoing, where X ∈ (Ventricular Episode, Potential Ventricular Episode), and

the diagnostics argument = ALL,

the device software shall request clearing of the following diagnostics:

- Short Term Average Diagnostic
- Long Term Average Diagnostic
- Episodal Diagnostics
- Exercise Compliance Diagnostic

In the SRS, each diagnostics bullet has its own requirement id. Instead of each requirement being an action to be performed when the trigger occurs, each of the children requirements was a data element in the structure of the diagnostics.

Solution: Parsing rules

The parser had to be updated to look for keywords such as “following diagnostics”. When these keywords are used, a diagnostic store is occurring; therefore the bulleted elements should be treated like diagnostic data elements instead of actions to be performed.

8.8 Communication between Features

8.8.1 Requests

Sometimes there are shared resources in the system that multiple features are trying to control. In these cases, each feature has to make a request for the given resource. The controller of the shared resource will then have some sort of arbitration to determine which feature's request is accepted. Grammatically, requests to another part of the software and requests to the hardware are treated identically. For the sake of simplicity, the same data structure will be used for hardware and software requests. Like many other parts of the system, this gets in the way of completeness tests because requests to the hardware are indistinguishable for requests to software with missing arbitration requirements. This issue is left for future work.

8.8.2 Liens

The firmware glossary defines the term, lien, as follows:

> A conditional suspension metaphor that enables one feature to temporarily suspend another feature to prevent unwanted interactions between the two features. The feature

liened upon determines which of its actions will be suspended during the lien. A lien placed by a feature can only be removed by that feature. If the lien placer goes out of scope, the lien goes with it.

8.8.3 Terminations

The firmware glossary defines the term, terminate, as follows:

> Terminate is an action that is performed on an algorithm. When an algorithm is terminated it exits all of its ongoing response.

It is understood from this that any request the algorithm is making is canceled. This says nothing about the internal state of the algorithm. The algorithm is not stopped with unless there is a lien associated with the termination, therefore the terminated feature is free to start making requests again shortly after the termination resolves.

8.9 Parts of Speech Handling

One method that was used to aid the categorization of the requirements was parts of speech handling. This method attempts to determine the noun, verbs, adjectives, etc. in a sentence. As mentioned in the literature review, there are a number of methods for performing this action with varying levels of success. Many of these techniques use probabilistic techniques to determine the sentence structure and the best techniques have 86% accuracy.

The approach taken in this thesis is different because it is preferable to have unclassified sentences than to have wrongly classified sentence. That requirement should be examined instead of being blindly classified, since this is a safety-critical environment. In order to avoid false categorization, in this project, only the least error prone techniques were used.

The main goal of this parts of speech engine is to identify noun phrases, whether they are in the subject or the predicate, since this would allow simpler categorization of actions and would help to determine whether conjunctions were separating noun phrases or parts of a compound sentence. This essentially works as a syntactical check, because if the sentence structure cannot be derived, it is almost impossible to derive the meaning of the sentence.

The first step in setting up the engine was to select the dictionary. The dictionary selected for the parser was Kevin Atkinson's Parts of Speech Dictionary, which was a combination of the WordNet database and the Moby (tm) Parts of Speech II database. Kevin Atkinson's Parts of Speech Dictionary had the advantage of including determiners, unlike most other similar dictionaries. It also seemed to be the more complete and accurate than the other dictionaries examined. For optimization purposes, a condensed version of the dictionary was derived which only contained words that were actually in the requirements document.

A correlated set was derived as well, words in the requirements document that were not in the dictionary. Many of these terms were acronyms or industry jargon, but this list did contain a number of misspelled words that could be corrected. This improved the quality of the document and made it easier to create a supplemental dictionary of the industry terms which could be combined with Kevin Atkinson's Parts of Speech Dictionary to create a full set of words.

The supplemental word list could also be used to override the defined parts of speech in the main dictionary. For example, the word “command” is frequently used in the SRS. In common English, the word “command” is used as a noun or a verb. However, in the SRS, it is always a noun.

Customizing the supplemental dictionary significantly increased the accuracy of the parts of speech handling.

The parts of speech engine goes through the following stages to determine the noun phrases within the document.

1) Known phrases from the glossary identify programmable parameters, which are classified as noun phrases.
2) Criteria flags are identified and classified as noun phrases.
3) Capitalized words that are not at the beginning of a sentence can safely be considered part of a noun phrase.
4) Numbers are classified as being part of a noun phrase.
5) Any words in the dictionary that are only nouns are classified as part of the same noun phrase.
6) Any noun directly followed by a noun is combined into the same noun phrase.
7) Any string of adjectives directly followed by a noun is combined into a single noun phrase.
8) Any word directly following an article is classified as part of a noun phrase.

The parts of speech engine converted the text of the noun phrase into a single string where to spaces where converted into underscore characters with an “!NP_” prefix. For example “ventricular pace pulse” was converted to “!NP_ventricular_pace_pulse”. The implementation of the parts of speech engine greatly simplified the parsing process since it condensed large sets of words into a single phrase that could be more easily operated on. This caused the parser to be much more accurate. Before the parts of speech engine was implemented, the detection rate for conditions was 22% and after it’s addition, the detection grew to 35%. (Extra filters further improved the detection rate later.)

8.10 Model for Parsed Data

Once the different concepts in the SRS were addressed, a model could be created for the data as stated by Gervasi. Since the main test being considered was the circularity check, it was necessary that actions could be linked to the triggers based on that action. For example, an assignment of a data element would be linked to a requirement that triggers on the update of that data element. Therefore, the first element to would be needed was an operation type (e. g. lien, termination, assignment). Next, the feature or data element being operated on would be necessary. For many operations, a source is needed as well. Each operation has its own data set that would need to be defined.

One approach to solve this problem would be to create different structures for each operation. While this approach defines the data in a precise and clear manner, each operation would need a different handler in the linker. A simpler solution would be to use a linked list whose head is the operation and whose nodes contains an attribute name, such as source or destination, and then the value of that element. The linker could traverse through the list and link the matching triggers and actions. Special programming could be used if the linking required special rules, but the default would be to match a set to elements. Because the linker was written in Perl, arrays were chosen instead of a linked list for speed of processing. Perl is optimized to deal with arrays and does not require that the length of an array to be set, which the primary advantage of linked lists. In the arrays, the first element is the operation code, followed by alternating data name and data value pairs.

For each action and trigger, a regular expression could be compared to the preprocessed text. If the regular expression matched the text, that action or trigger would be processed accordingly. For example, the regular expression /^\s*terminate\s+any\s+ongoing\s+(\!NP_[\w\-\/]+)\W*$/ could be used to process termination of a feature, which has already been processed as a noun

phrase. One important aspect of the regular expression is that it expects only those 4 words. If there is extra wording, the expression does not match the text and the other regular expressions are attempted.

In the first version of the parser, a different routine was used to parse every match. Since that method was cumbersome and difficult to maintain, a different approach is now used. Instead of directly writing all of the regular expressions and routine, an array is maintained per translation rule. The first element of the array is a string that contains the expected text with a code for each noun phrase expected. The other elements of the array represent the data structure which results from the translation. For example, one of the rules for assignment translation is ["set !NN to !NN", $ASSIGNMENT, “destination, "!A", “source”, "!B"]. The first element is translated to the regular expression
/^\s*set\s+(\!NP_[\w\-\/]+)\s+to\s+!NN\W*$/. If the action matches this regular expression the data structure for the action is the array ($ASSIGNMENT, “destination, "!A", “source”, "!B"), where !A is replaced with the first noun phrase and !B is replaced with the second noun phrase.

The translator also looks for defined prepositional phrases and translates them before the main translator runs. It works similarly to the main translator but for the action or trigger it removes the prepositional phrases it finds and saves its results in an array that will be appended to the data structure. For example, to process the trigger “On setting of the RVAC Threshold to FAIL after a logical channel closed RVAC Search during the current daily period”, the following steps would be taken:

Step	Text	Result Array
Original Text	On update of the RVAC Threshold to FAIL after an RVAC Search during the current daily period	empty
Parsing	On update of !NP_RVAC_Threshold to !NP_FAIL after !NP_RVAC_Search during !NP_current_daily_period	empty
Prepositional rule: ["after !NN", "after", "!A"]	On update of !NP_RVAC_Threshold to !NP_FAIL during !NP_current_daily_period	[“after”, “!NP_RVAC_Search”]
Prepositional rule: ["during !NN", "during", "!A"]	On update of !NP_RVAC_Threshold to !NP_FAIL	[“after”, “!NP_RVAC_Search”, "during", "!NP_current_daily_period"]
Main rule: [“on !NP_update of !NN to !NN”, $ASSIGNMENT, “destination”, “!A”, “source”, “!B”],	empty	[$ASSIGNMENT, “destination”, "!NP_RVAC_Threshold", “source”, “!NP_FAIL”, “after”, “!NP_RVAC_Search”, "during", "!NP_current_daily_period"]

Table 5: Parsing Example

8.11 Parser Results

Each of the stages of the preprocessor and parser described in the previous sections were implemented successfully. The filters were running as expected and more filters could be added with minimal effort. 60% of the events/conditions could be correctly parsed and 50% of the actions could be correctly parsed. New filters at this stage seem to have minimal impact on the overall performance; the addition of more filters is yielding diminishing returns. The performance of the parser is not adequate enough to justify the creation of a linker yet. Running a circularity check on a tree which is missing almost half of its nodes will not provide useful information.

In the Success Measurement section, the following success criteria were laid out as follows:

Expectation Name	Description	Reason for Choosing
Complete Translation of One Feature	At least one feature will be completely translated.	Demonstrates that whole features can be translated using this technique.
Identification of Translation Errors	The parser will correctly identify requirements which cannot be translated properly.	Demonstrates that the parser will not perform translations that the rule set is not programmed to handle.
75% Identification	The parser will be able to correctly identify 75% events, conditions and actions in the event-based requirements.	Demonstrates the percentage of requirements that can be translated without a rewrite.

Table 6: Project Expectation Review

There was one feature, Therapy Enable Disable, for which every requirements was successfully translated into data structures. In four other features, either all of the actions or all of the events and conditions were successfully parsed. This expectation was successfully met.

As stated before, the parser identifies all of the requirements that it cannot translate; therefore the second bullet point was successfully met.

The parser could only understand 75% of the events, conditions and actions in the event-based requirements. Much of this was due to stylistic issues. There are a number of issues that make automatic testing of requirements difficult that are not due to any error in the writing of the requirements. The initial assumption that 3/4 of the events, conditions and actions in the event-based requirements could be parsed correctly was naïve. This expectation was not met.

Therefore the results are as follows:

Expectation Name	Description	Results
Complete Translation of One Feature	At least one feature will be completely translated.	Success – One feature was successfully translated in its entirety.
Identification of Translation Errors	The parser will correctly identify requirements which cannot be translated properly.	Success – The parser can successfully differentiate between requirements that meet the rulesets and requirements that do not.
75% Identification	The parser will be able to correctly identify 75% events, conditions and actions in the event-based requirements.	Failure – The parser could only translate 50% of the requirements.

Table 7: Project Results

9. Contributions

There are 3 major contributions provided by this thesis.

1) This thesis provides an application of Gervasi and Nuseibeh's framework from Lightweight Validation of Natural Language Requirements to a real-world, safety-critical requirement specification.
2) This thesis explores a novel approach to using PCFGs in a way that reduces miscategorization.
3) This thesis produces a product that can be used to help validate the requirements specification it was created for and future specifications based on it.

The most significant contribution in this thesis is the application of Gervasi and Nuseibeh's framework from Lightweight Validation of Natural Language Requirements to a real-world, safety-critical requirement specification. This project demonstrated the utility and versatility of the framework while demonstrating actual constraints imposed by budget and corporate culture.

The second most significant contribution was the adaptation of PCFGs in a way which reduces errors. PCFG tools have an inaccuracy percentage which is not acceptable in safety-critical applications. However, a partial use of PCFG can produce a solution with a smaller percentage of misrepresented cases at the cost of more unknown cases, which is preferable in this environment.

The third contribution is the actual product. This produces a real benefit for the engineers using this specification and will help future specification based on it.

10. Conclusion

Jeannette Wing states that “a method is formal if it has a sound mathematical basis”. [3] On this basis, the current incarnation of the SRS cannot be translated into a formal grammar. Despite the use of basic arithmetic, Boolean logic, and predicate logic, the language used in the SRS is simply not rigid enough to translate.

Since over half of the requirements can currently be represented in data structures, this makes any future translation of the requirements into a Computer Aided Software Engineering system a much smaller task. It would also aid in translation of the requirements into any other formal specification. In the future additions to this SRS, new requirements will be evaluated using this parser to help to ensure the accuracy, completeness, and usability of the specification.

The system of translation developed for this work is a contribution as well. By using a simple array of arrays, translation rules can be easily added to accommodate future changes in the SRS. Entirely new concepts can be supported in the data structure by adding a handful of rules. The concepts used in this thesis can be applied to other technologies easily. By customizing the lexicon, and producing a regular expression rule set list, this parser could be applied to requirements documentation in other fields as well.

In conclusion, from an SRS improvement perspective, this project translated half of its requirements into data structures, and provided a means to quickly expand on the work already done. From a general requirements analysis perspective, advances made in this project include: a framework for storing requirements, a program which translates most of the natural language requirements into the framework, and a novel approach to parts of speech analysis.

11. Future Work

There are a number of tasks that could be performed to continue the work started in this thesis. Additional filters could be added to the rule set to improve the output. This approach has diminishing returns, but could yield a more robust parser. Next, improvements could be made to the grammatical processing of the parser. Definition of the parts of speech of key words used in the thesis could yield better results in the parser. Lastly, at the current stage, only the format of the requirements can be checked. Once a sufficient number of the requirements have been translated, additional checks, such as circularity and disjointness could be performed.

The current system can recognize prepositions like “during” and “after”, however it cannot translate these concepts into a real timing analysis. The focus of this thesis has been the causal relationships between event and actions, but the timing issues related to the requirements have been largely ignored. Timing analysis is a possible area for future work.

The parser is not currently robust enough to handle very large sentences, such as complex if statements. There are request handlers in the requirements that can receive up to ten requests at a time and use multi-branch conditionals to handle the arbitration. This is not an issue with the SRS, as the requirements are well formed, but a limitation of the parser’s coding. The parser has difficulty with complex arithmetic analysis for similar reasons.

The parser needs support for requirements that are not event-based. The majority is the requirements in the current SRS are event-based, but many of them can be simplified by removing unnecessary conditionals.

Once 100% (or near 100%) translation is reached, there are many possibilities. One promising example is that the requirements could be translated into a formal language or a Computer Aided

Software Engineering system, allowing the benefits of all of the tests mentioned in the thesis. The work needed by the engineers themselves would be minimized because the translator would handle most of the conversion automatically.

References

[1] R. W. Butler. What is Formal Methods? http://shemesh.larc.nasa.gov/fm/fm-what.html

[2] S. Harte. Discussion. Jun. 2009.

[3] J. Wing. A Specifier's Introduction to Formal Methods.

[4] E. Clarke, M. Edmund. and J. Wing. Formal Methods: State of the Art and Future.

[5] M. Heimdahl and N. Leveson, Completeness and consistency in hierarchical state-based requirements. IEEE Transactions on Software Engineering. June 6, 1996

[6] R. Baber, D. Parnas, and S. Vilkomir. Disciplined Methods of Software Specification: A Case Study. Proceedings of the International Conference on Information Technology Coding and Computing. 2005.

[7] J. Spivey. The Z Notation: a reference manual, 2001.

[8] N. Leveson, Safeware: System Safety and Computers. Addison Wesley, 1995.

[9] D. Harel. Statecharts: A Visual Formalism for Complex Systems, 1987.

[10] N. Leveson. An Approach to Building. Human-Centered Specifications, 2000.

[11] R. Charette. Why Software Fails. IEEE Spectrum, 2005.

[12] H. Van, A. van Lamsweerde, P. Massonet, C. Ponsard. Goal-Oriented Requirements Animation.

[13] N. Dulac, et. al., On the Use of Visualization in Formal Requirements Specification.

[14] C. Heitmeyer, R. Jeffords, and B. Labaw, Automated Consistency Checking of Requirements Specifications.

[15] IBM. Rational Requirements Composer. http://www-01.ibm.com/software/awdtools/rrc/

[16] J. Bowen and M. Hinchey. Ten Commandments of Formal Methods … Ten Years Later. Computer, 39(1). 2006.

[17] V. Gervasi and B. Nuseibeh. Lightweight Validation of Natural Language Requirements. Software – Practice & Experience. 32(2) 2002.

[18] The Stanford Natural Language Processing Group. The Stanford Parser: A statistical parser. http://nlp.stanford.edu/software/lex-parser.shtml

[19] L. Goldin and D. Berry. Abstfinder, A Prototype Natural Language Text Abstraction Finder for Use in Requirements Elicitation. Automated Software Engineering. 4(4). 1997

[20] IBM. Medical Best Practices Seminar. March 18, 2010.

[21] V. Gervasi and D. Zowghi. Reasoning About Inconsistencies in Natural Language Requirements. ACM Transactions of Software Engineering and Methodology. 14(3). 2005.

[22] L. Mich, M. Franch, and P. Novi Inverardi. Market Research for Requirements Analysis Using Linguistic Tools. Requirements Engineering Journal. 9(1). 2004.

[23] H. Kaiya and M. Saeki. Based Requirements Analysis: Lightweight Semantic Processing Approach. 5th International Conference on Quality Software. 2005.

[24] D. Jackson and J. Wing. Lightweight Formal Methods. ACM Computing Surveys. 1996.

[25] J. Coughlin and R. Macredie. Effective Communication in Requirements Elicitation: A Comparison of Methodologies. Requirements Engineering. 2002.

[26] J. Offutt and A. Abdurazik. Generating Tests from UML Specification. Lecture Notes in Computer Science. 1999.

[27] N. Fuchs, K. Kaljurand, and G. Schneider. Attempto Controlled English Meets the Challenges of Knowledge Representation, Reasoning, Interoperability, and User Interfaces. Proceedings of FLAIRS '06. 2006.

[28] E. Kamsties, D. Berry, and B. Paech. Detecting Ambiguities in Requirements Documents Using Inspections. Proceedings of the First Workshop on Inspection in Software Engineering. 2001.

[29] L. Kof. Scenarios: Identifying Missing Objects and Actions by Means of Computational Linguistics. Requirements Engineering Conference, 2007. 2007.

[30] L. Kof. Natural Language Processing: Mature Enough for Requirements Documents Analysis? Tenth International Conference on Applications of Natural Language to Information Systems. 2005.

[31] N. Leveson. Completeness in Formal Specification Language Design for Process-Control Systems. Proceedings of the third workshop on Formal Methods in Software Practice. 2000.

[32] T. E. Bell, D. C. Bixler, and M. E. Dyer. An Extendable Approach to Computer-Aided Software Requirements Engineering. IEEE Transactions on Software Engineering. 3(1). 1977.

[33] D. Berry and E. Kamsties. Ambiguity in Requirements Specification. Perspectives on Software

Requirements. 2004.

[34] D. Klein and C. Manning. Accurate Unlexicalized Parsing. Proceedings of the 41st Meeting of the Association for Computational Linguistics. 2003

[35] IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 803-1998. IEEE-SA Standards Board. 1998.

[36] Simplified Technical English. Specification ASD-STE100. ASD Simplified Technical English Maintenance Group. 2010.

[37] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. Proceedings of the 22nd International Conference of Software Engineering. 2000.

[38] C. Heitmeyer. A Panacea or Academic Poppycock: Formal Methods Revisited. High-Assurance Systems Engineering. 2005.

[39] H. Moses, B. Miller, K. Moulton, and J. Schneider, A Practical Guide to Cardiac Pacing. 5th Edition. 2000.

[40] S. Owre, J. Rushby, and N. Shakhar. PVS: A Prototype Verification System. CADE. 1992.

[41] Formal Methods Wiki. http://formalmethods.wikia.com/wiki/Formal_methods

[42] C. DeJong, M. Gibble, J. Knight, and L. Nakano. Formal Specification: A Systematic Evaluation. Technical Report CS-71-13, Department of Computer Science, Uinversity of Viginia, Charlottesville. 1997.

[43] Guidelines for Successful Acquisition and Management of Software-Intensive Systems: Weapon Systems, Command and Control Systems, Management Information Systems – Condensed Version 4.0. Software Technology Support Center. U. S. Air Force. 2003.

[44] D. Litvack, E. Azzam, S. Harte, J. Patterson. Review of the Firmware Requirements Guidelines Document. Discussion. April 2010.

[45] J. Arthur and T. Stevens, Document Quality Indicators: A Framework for Assessing Document Adequacy, Virginia Polytechnic Institute, State University, 1990.

[46] N. Zafar, Automatic Construction of Formal Tree Syntax based on Regular Expressions. Proceedings of the WCE 2012 Vol II, 2012.

[47] L. Freitas, Proving Theorems with Z/Eves. University of Kent. 2004.

[48] D. Litvack and J. Patterson. Discussion. Sept. 2010.

[49] J. Patterson, E. Azzam, and S. Yang. Development Meetings. Oct. 2010.

[50] J. Patterson and D. Litvack. Discussion. July 2009.