8-2013

COMPUTATIONAL REPRESENTATION OF LINGUISTIC SEMANTICS FOR REQUIREMENT ANALYSIS IN ENGINEERING DESIGN

Alex Lash
Clemson University, alash@g.clemson.edu

COMPUTATIONAL REPRESENTATION OF LINGUISTIC SEMANTICS FOR REQUIREMENT ANALYSIS IN ENGINEERING DESIGN

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Mechanical Engineering

---

by
Alex Vincent Lash
August 2013

Accepted by:
Dr. Gregory M. Mocko, Committee Chair
Dr. Joshua D. Summers
Dr. Georges Fadel

ABSTRACT

The objective of this research is to use computational linguistics to identify semantic implicit relationships between text-based relationships. Specifically, natural language processing is used to implement linguistic semantics in requirement analyzers. Linguistic semantics is defined as the meaning of words beyond their string form, part of speech, and syntactic function. Many existing design tools use part of speech tagging and sentence parsing as the foundation of their requirement analysis but ultimately use string algorithms to evaluate requirements. These string algorithms cannot capture the implicit knowledge in requirements. This research compares five methods of requirement analysis. A manual analysis provides the benchmark against which the subsequent analyzers are judged. A syntactic analysis is implemented and compared to the manual method to gain insight into the capabilities of current methods. The other three analyzers implement semantic tools for requirement analysis through semantic ontologies and latent semantic analyses. The results from the semantic analyzers are compared to the results of the other two analyzers to judge the capabilities of semantics in requirement analysis. The findings show that semantics can be identified with at least 74% accuracy. Further, the agreement between the semantic results and the manual results are more related than the syntax results and the manual results. While the implementation of semantics into requirement analysis does not completely agree with manual findings, the semantic analyses improve upon syntactic and string matching analyses used in current research.

DEDICATION

For my Mom and Dad.

ACKNOWLEDGEMENTS

Dr. Gregory Mocko

Dr. Joshua Summers

Dr. Georges Fadel

Dr. Mackay Salley

Dr. Anne Catlla

Dr. David Sykes

Ms. Kaye Autrey

Dr. Melissa Thibodeau

Ms. Barbara Booth

Andrew Klas

William Lefever

Bill Sharp

Alex Taylor

Cameron Ferguson

Nathan Milam

Ryan Linton

John Dubose

Geoff Goff

Lukeus Wingo

TABLE OF CONTENTS

Page

Table of Contents (Continued)

LIST OF TABLES

List of Tables (Continued)

Table Page

LIST OF FIGURES

Figure Page

List of Figures (Continued)

Figure Page

List of Figures (Continued)

Figure Page

List of Figures (Continued)

Figure Page

CHAPTER ONE
OVERVIEW OF REQUIREMENT ANALYSIS RESEARCH

Chapter Objectives:

- Establish the motivating research problem.
- Identify and describe research objectives.
- Provide a comprehensive overview of semantic requirement analysis research.
- Provide an outline of the thesis.

The purpose of this research is to use computational linguistics to identify implicit relationships between text-based requirements. Specifically, these relationships are identified based on requirement semantics. Relationships between requirements aid engineering designers by providing a graphical representation of requirements that allows multiple relationships between requirements. For instance, requirement relationships are used to predict change propagation when an initial requirement is modified or deleted. This prediction can save time and money by minimizing the need to iterate over the design process. Current manual methods find intelligent relationships between requirements, but become tedious and error prone when scaled up. Also, existing automated methods use string matching and syntax to identify relationships. These relationships are not in agreement with those found manually.

Through the implementation of a semantic analysis into current automated methods, this research seeks to build upon syntactic methods to bring requirement analysis results in closer agreement with manual results. Semantics afford the scalability of automated methods while finding relationships that more closely replicate manual methods. To accomplish this goal, three semantic requirement analyzers are created that derive the semantic relationships between the requirements. The results from the semantic analyzers are compared to:

- Existing syntactic and string matching requirement analysis methods.
- Requirements models based on human expertise (manual).

The manual models provide a benchmark against which the analyzers are evaluated. A requirement analyzer using only string matching and syntax analysis is also created. This analyzer is representative of the automated requirement analysis method used in current research. As with the semantic analyzers, this analyzer is also compared to the manual models. The performance of the syntactic analyzer serves as the baseline against which the abilities of the semantic analyzers are evaluated.

In particular, this research aims to meet three research objectives (RO). These objectives are shown in Table 1-1.

Table 1-1: List of research objectives.

RO	Description
1	Apply linguistic semantics to requirement statements to improve computational understanding of requirement statements.
2	Identify and form semantic relationships between requirements.
3	Compare semantic analysis methods against syntax and manual methods to show the value of semantics to requirement analysis.

The following sections detail these research objectives and how each objective is tested. Also, the sections show how each research objective (RO) relates to the overarching research purpose and motivation.

1.1 RO 1: Supplementing Requirements with Semantics

Linguistic semantics is defined as the meaning of terms beyond their string form, part of speech, and syntactic function. Linguistic semantics can be thought of as the implicit knowledge in a text. Many existing design tools do not use linguistic semantics to analyze requirements and use part of speech tagging and sentence parsing alone as the foundation of their requirement analysis. These tools use string operations to evaluate the requirements. Syntax and string matching methods do not replicate manual results. This conclusion is drawn by finding the level of agreement between the manual results and result of a method using syntax and string matching methods alone. The level of agreement is found using Cohen's Kappa, which determined a poor agreement of 0.27. By incorporating linguistic semantics, implicit knowledge can be identified. Consequently, the results of the semantic analyzers more closely represent manual results with a maximum kappa value of 0.48. These results are discussed in further depth in the test cases performed in Chapter Four.

Linguistic semantics are applied using natural language processing (NLP). NLP provides a set of tools to apply semantics in computational methods. Two semantic tools are used in the semantic requirement analyzers. These semantic tools are latent semantic analysis (LSA) and semantic ontologies. Two of the analyzers employ only LSA. The third tool implements both LSA and a semantic ontology. The relationships discovered by the semantic analyzers are evaluated relative to the manual approach and syntactic/string matching approach.

Semantics are supplied to the requirements in two ways. LSA find latent relationships between requirement statements. The other method is performed by mapping the objects of the requirement statements to a semantic ontology. While LSA supplies semantics, the LSA model does not explicitly define two semantically related terms or documents. The semantic ontology relies on external expertise and relationships between words such as those captured in traditional dictionaries and thesauruses. Mapping to a semantic ontology defines the terms and the terms are related based upon defined semantics such as synonymy. As a result, the semantic ontology allows for an accuracy analysis of the semantic mapping. This accuracy analysis judges the ability of an analyzer to automatically supply a term with meaning (i.e. define a term). The results of this accuracy analysis show the ability of computational methods to add meaning to requirement terms and statements. In turn, the mapped semantic terms are used to relate requirements based on the semantic similarity in the ontology.

1.2 RO 2: Forming Semantic Relationships

In this research, semantics are used to form relationships between requirements. Two of the semantic requirement analyzers form relationships based upon cosine similarity from the requirement vectors created by LSA. The other semantic requirement analyzer forms relationships by calculating the ontology path length based on the mapped semantic terms.

The requirement analyzers model these relationships in a design structure matrix (DSM). The DSMs are $n \times n$ matrices where n is the number of requirements. Each cell is a value indicating the relationship between the requirements. The relationships are based on the semantics supplied by the analyzers (RO 1). The relationships are bidirectional forming symmetric matrices. Relationships between requirements have applications as formal models in requirement repositories and have been demonstrated in requirement analyzers such as requirement change propagation detection. Also, relationships between requirements capture conceptual knowledge of the requirements that is often overlooked by traditional hierarchal requirement list formatting. For instance, a traditional structure is usually subdivided into headings such as geometry and safety. Yet some requirements pertain to multiple headings, but a traditional requirements list cannot capture multiple relationships. Forming relationships between requirements using a DSM provides a web/graph structure that can capture this information.

1.3 RO 3: Value of Linguistic Semantics to Design

To assess the value of semantics to requirement analysis, the developed requirement analyzers are used to evaluate a requirements document of a BMW accelerator pedal module. In addition to the three semantic analyzers and the syntactic/string-matching analyzer, the requirements document is manually analyzed. The manual analysis serves as the benchmark against which the requirement analyzers are judged. It should be noted that the time efficiency of the analyzers is not evaluated directly by this research.

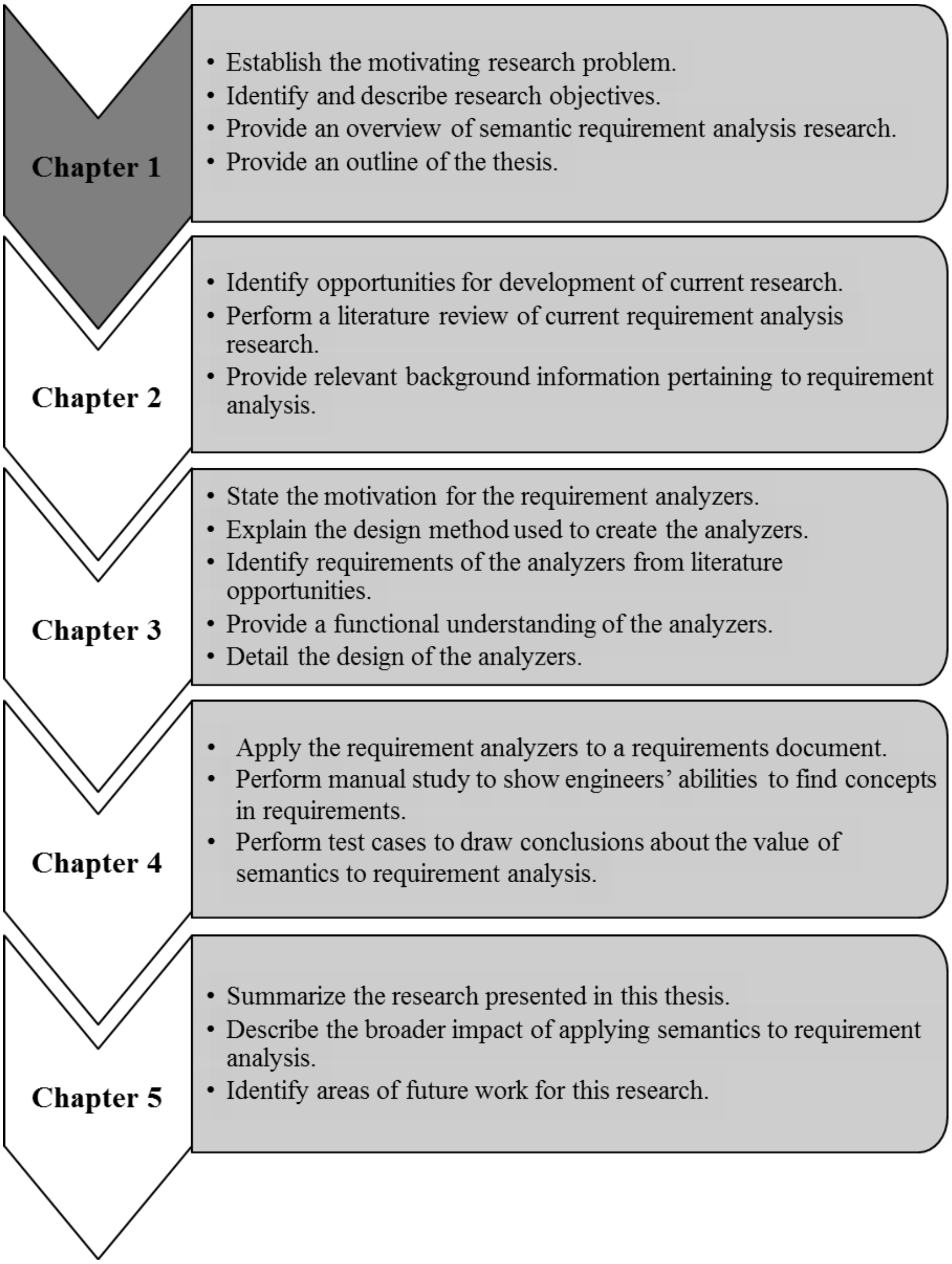
Six test cases are then performed to evaluate the results of each tool. The results of the manual analysis are first compared to one another to measure the level of agreement between engineers when relating requirements. The manual results are also compared to each requirement analyzer. The level of agreement between the manual and syntactic/string-matching analyzer represents the abilities of existing requirement analysis methods. The semantic analyzers aim to improve upon the results obtained by the syntactic/string-matching analyzer. Also, the ability to map requirement terms to a semantic ontology is measured.

1.4 Overview of Thesis

The first two research objectives relate to the computation interpretation of requirements. The first objective interprets the semantics or implicit knowledge in the requirements. The second objective gains conceptual insight about the requirements by finding relationships between requirements. Each of the three analyzers meets these objectives in a unique approach. The third research objective seeks to show the value of semantics to requirement analysis by evaluating the analyzers against each other and established methods that use syntax and string matching alone. The remainder of the thesis details the research from literature review through conclusions of the semantic requirement analyzers. Specifically, the thesis:

- Surveys current literature and research in requirements analysis.
- Identifies opportunities in current literature that will be addressed by this research.
- Defines the problem and derive requirements for the requirement analyzers based on the identified opportunities in literature.
- Conceptualizes, embodies, and details the analyzers to meet the derived requirements.
- Implements the analyzers on a test case requirements document to evaluate solutions.
- Answers research objectives and draw conclusions including future work.

Figure 1-1 provides a visual outline of the thesis showing the deliverables of each chapter as they relate to the objective of this research.



Chapter 1

- Establish the motivating research problem.
- Identify and describe research objectives.
- Provide an overview of semantic requirement analysis research.
- Provide an outline of the thesis.

Chapter 2

- Identify opportunities for development of current research.
- Perform a literature review of current requirement analysis research.
- Provide relevant background information pertaining to requirement analysis.

Chapter 3

- State the motivation for the requirement analyzers.
- Explain the design method used to create the analyzers.
- Identify requirements of the analyzers from literature opportunities.
- Provide a functional understanding of the analyzers.
- Detail the design of the analyzers.

Chapter 4

- Apply the requirement analyzers to a requirements document.
- Perform manual study to show engineers' abilities to find concepts in requirements.
- Perform test cases to draw conclusions about the value of semantics to requirement analysis.

Chapter 5

- Summarize the research presented in this thesis.
- Describe the broader impact of applying semantics to requirement analysis.
- Identify areas of future work for this research.

Figure 1-1: Overview of thesis chapters showing completed chapters (grey chevron).

CHAPTER TWO
LITERATURE REVIEW OF RESEARCH IN NATURAL LANGUAGE REQUIREMENT ANALYSIS

Chapter Objectives:

- Perform a literature review of current requirement analysis research.
- Provide relevant background information pertaining to requirement analysis.
- Identify opportunities and gaps for development of current research.

A variety of analysis methods are used to interpret requirements. The goal of each analyzer is to provide the user/designer with valuable information about the requirements document. This chapter discusses the foremost methods for analyzing requirements:

- Established syntactic method for analyzing requirements.
- Established semantic NLP tools and their roles in requirement analysis.
- Established methods for gaining valuable conceptual information about requirements.

These methods are better understood through the linguistic approach to requirement analysis. Therefore, this method is detailed in this chapter. Before understanding the methods for analyzing requirements, it is important to understand the importance of requirement analysis.

2.1 Importance of Requirement Analysis

Understanding the importance of requirements in the design process is needed to understanding the importance and benefits of requirement analysis. Though many different design methodologies exist, requirements are consistently established as the foundation of the design process [1, 2, 3]. Requirements are realized from customer wants and refined over the course of the product development. Therefore, requirements serve as the connection between customer needs and the realized product. The requirements document also serves as a guide to judge the success of the product. Figure 2-1 shows the systematic design process and the role of requirements during each phase.

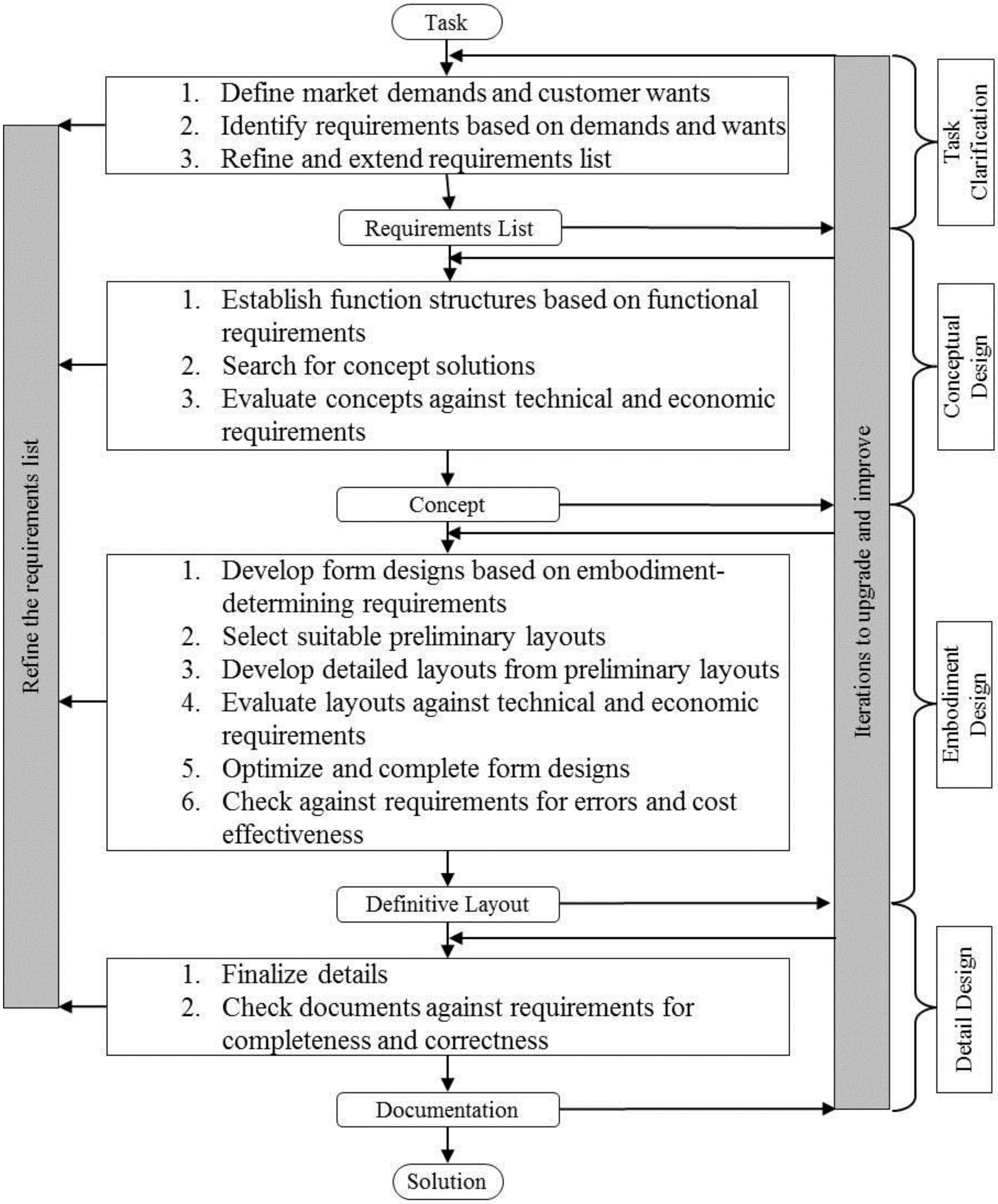

Task
1. Define market demands and customer wants
2. Identify requirements based on demands and wants
3. Refine and extend requirements list
Requirements List
1. Establish function structures based on functional requirements
2. Search for concept solutions
3. Evaluate concepts against technical and economic requirements
Concept
1. Develop form designs based on embodiment-determining requirements
2. Select suitable preliminary layouts
3. Develop detailed layouts from preliminary layouts
4. Evaluate layouts against technical and economic requirements
5. Optimize and complete form designs
6. Check against requirements for errors and cost effectiveness
Definitive Layout
1. Finalize details
2. Check documents against requirements for completeness and correctness
Documentation
Solution
Refine the requirements list
Iterations to upgrade and improve
Task Clarification
Conceptual Design
Embodiment Design
Detail Design

Figure 2-1: Adapted systematic design process detailing the use of requirements throughout the process [1].

Figure 2-1 shows that not only are requirements used to assess the final product, but also to assess the concepts and principles developed within the process. For instance, during concept evaluation and selection, the requirements are used as a metric to select the concept(s) to embody. It is important then that the requirements document is of high quality to ensure that the use of the document is both efficient and valuable. Also, Figure 2-1 shows that the requirements list is refined over the course of the design process. Requirements are added, deleted, and modified throughout the design process. Consequently, the requirements must be analyzed often, and therefore, an efficient requirement analysis method is beneficial.

2.2 Linguistic Approach to NL Requirement Analysis

Most requirements are written in natural language (NL). NL is chosen because it is ergonomic for both efficient formulation and efficient communication, especially among non-technical stakeholders. Customer wants for products are often expressed in NL and therefore, requirements are often first derived in NL.

NLP tools address a given text such as a requirements document in levels. [4, 5]. These categories are based on the linguistic level of interpretation. The three levels of interpretation are shown in Table 2-1. The levels can affect one another. For instance, ambiguity on the syntactic level can affect the understanding at the semantic and conceptual levels. More importantly, a better understanding at the semantic level can improve the conceptual understanding.

Table 2-1: Levels of interpretation for understanding natural language text.

Level of Interpretation	Description	Computational Methods
Syntactic	Part of speech and sentence structure	Part of Speech Tagging Sentence Parsing
Semantic	Word or sentence meaning; Requires context	Latent Semantic Analysis Semantic Ontologies
Conceptual	Requirements meaning; Requires domain knowledge	Formal Modeling Quality Metrics

The syntactic level of interpretation is well-established in requirement analysis. This level provides the base upon which a conceptual understanding of requirements can be formed. The semantic level can provide meaning to words beyond their syntax thus furthering the understanding of requirements. Like the syntactic level, the conceptual level is also well-established in requirement analysis. At the conceptual level, domain knowledge is applied. Using this knowledge, useful information can be supplied to the user/designer. This research seeks to enhance the conceptual level of interpretation by improving upon the established syntactic methods through semantics.

2.3 Established Syntactic NLP Methods for Requirement Analysis

Syntax is defined as the understanding of the structure of text. Text can be a word, statement, paragraph, or an entire document depending on the analysis. For requirement analysis, this text is most often words and sentences. For a word, syntax involves its part of speech—its structural purpose in a statement. For a statement, syntax involves its parse tree—the structure of a statement. Syntactic meaning is applied to requirements via three NLP methods.

1. Tokenizing – separating a text into its respective tokens (e.g. words, numbers)

2. Tagging – applying part of speech tags to each token
3. Parsing – creating a tree structure of a statement

It is common practice to refer to the combination of all three steps as parsing and the combination of the first two steps as tagging. Tagging and parsing are well-established NLP methods for requirement analysis. Many requirement analysis tools use tagging and parsing as the foundation of their requirement analysis. Tagging applies part of speech tags to each token. Parsing creates a tree structure by applying part of speech tags as well as phrase tags. Figure 2-2 shows a sample parse tree of a requirement.

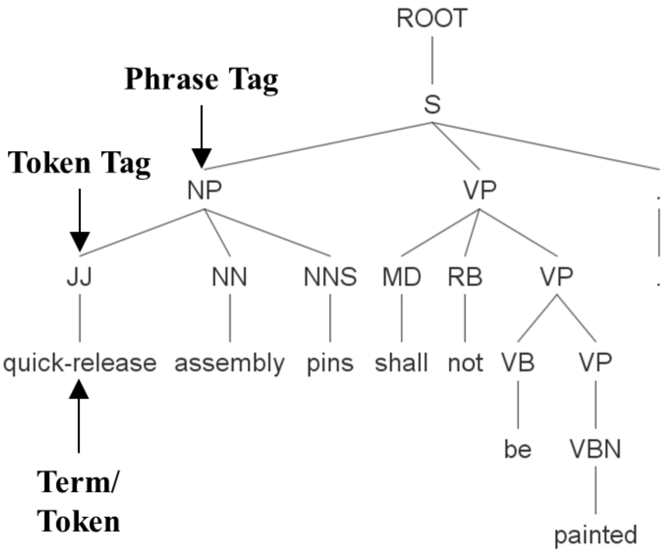
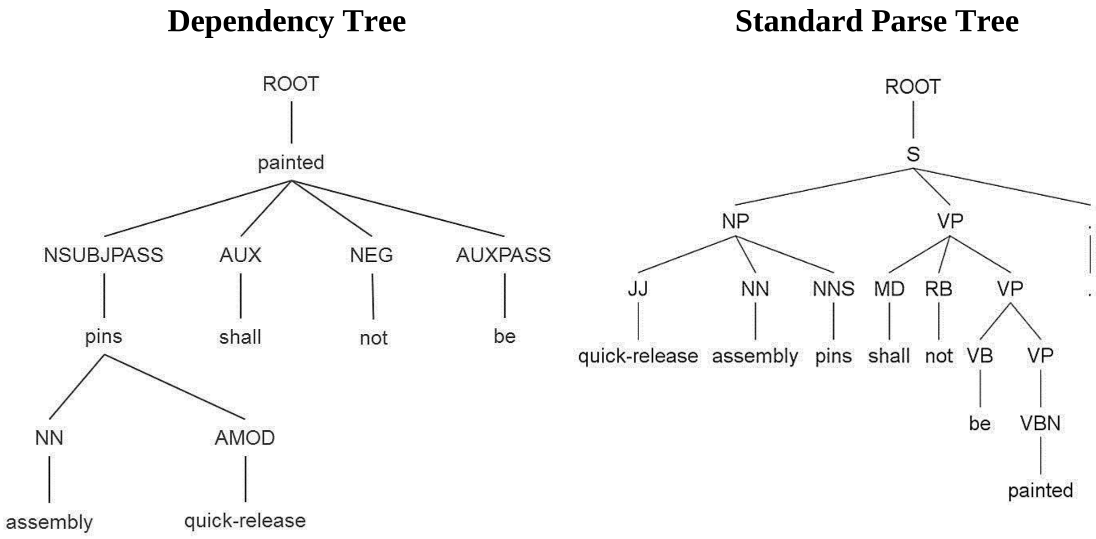


Figure 2-2: Sample parse tree showing general structure, key elements, and an example of an adjectival noun [6].

Dependency trees are parse trees that provide the syntactic function of each word in a statement as opposed to only the part of speech or phrase tag. As seen in Figure 2-3, the requirement statement:

Quick-release assembly pins shall not be painted.

is parsed in both formats [6]. From the dependency tree, it can easily be found that the word *pins* is the subject by the tag NSUBJPASS, which stands for noun subject passive. On the right, the standard parse tree provides similar information except the finding of the subject can only be found through the finding of the plural noun (NNS) *pins* under the first noun phrase (NP). In this way, the dependency tree provides more direct, valuable information as compared to standard parse tree.


Dependency Tree

Standard Parse Tree

Figure 2-3: Dependency tree (left) versus a standard parse tree (right) of a requirements statement.

Being able to identify specific syntactic elements in a requirement such as the subject and direct object is important to requirement analysis. Research has shown that a link between syntactic elements and requirement concepts can be obtained [7, 8]. This method is discussed further in Section 2.5.2. This research seeks to build on the established syntactic NLP methods for requirement analysis. The requirement analyzers designed for this research implement parsing to provide a foundation for the semantic and conceptual requirement analysis methods.

2.4 Semantic NLP Tools

Once the syntactic methods have been performed in a requirement analysis, the semantic and conceptual interpretations of requirements are performed. Semantic analysis is not used often in requirement analysis and only syntax and string matching methods are used to gain conceptual insight [9, 10, 11, 12, 13]. However, string matching and syntax alone may not be an effective way to gain conceptual insight about requirements. Therefore, semantics may provide a method for gaining further insight about requirements in order to enhance conceptual understanding. Two accepted semantic tools are used in the requirement analyzers. These tools are latent semantic analysis and semantic ontologies.

2.4.1 Latent Semantic Analysis

One tool for identifying semantics in text is latent semantic analysis (LSA). LSA utilizes singular value decomposition (SVD) to reduce the noise in a set of texts to reveal latent semantics. This method provides a way to relate texts based upon the implicit relationships within a given domain. In terms of a requirement analysis, the domain is typically the requirements list as a whole, where each requirement is considered a document. Figure 2-4 shows the basic workflow of a LSA with sample objects at each phase.

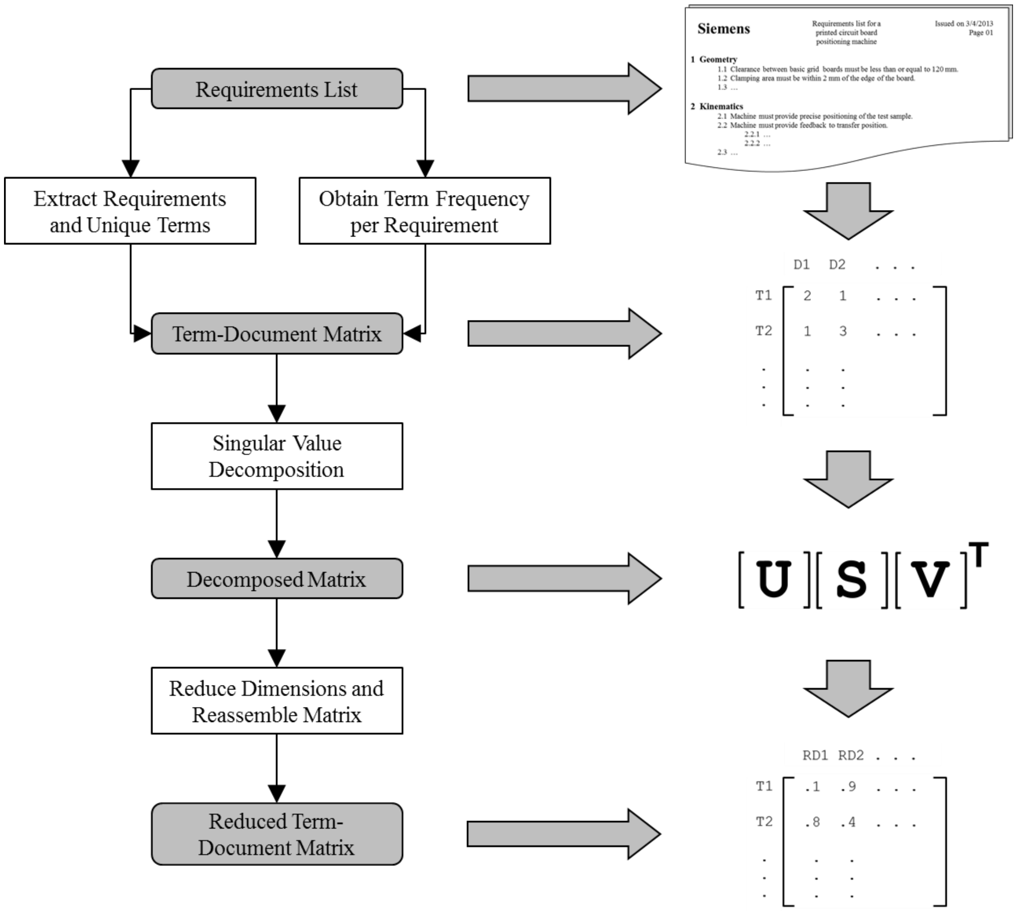

Requirements List
Extract Requirements and Unique Terms
Obtain Term Frequency per Requirement
Term-Document Matrix
Singular Value Decomposition
Decomposed Matrix
Reduce Dimensions and Reassemble Matrix
Reduced Term-Document Matrix
Siemens
Requirements list for a printed circuit board positioning machine
Issued on 3/4/2013
Page 01
1 Geometry
1.1 Clearance between basic grid boards must be less than or equal to 120 mm.
1.2 Clamping area must be within 2 mm of the edge of the board.
1.3 …
2 Kinematics
2.1 Machine must provide precise positioning of the test sample.
2.2 Machine must provide feedback to transfer position.
2.2.1 …
2.2.2 …
2.3 …
D1 D2 . . .
T1 2 1 . . .
T2 1 3 . . .
[U][S][V]T
RD1 RD2 . . .
T1 .1 .9 . . .
T2 .8 .4 . . .

Figure 2-4: LSA workflow (left) with sample objects from each function (right).

First, all the terms in the requirements are extracted. An ***n***×***m*** term-document matrix is created where ***n*** is all the unique terms in the requirements list and ***m*** is all of the individual requirements. The matrix is populated with the frequencies of each term per requirement. Next, the matrix is decomposed through singular value decomposition (SVD). While decomposed, the rank can be reduced while preserving the relationships of the term-document matrix. The decomposed matrix is then reassembled and the resulting matrix provides a term-document matrix where the rank—related to the implicit topics in the requirements list—is reduced. As a result, the latent semantics or implicit relationships between the requirements can be revealed. The row vectors can be related via cosine similarity (or other vector comparison method) to find similarity between terms. Also, the column vectors can be compared in a similar fashion to find the similarity between requirements.

If LSA is being run on raw NL text, it may be valuable to normalize the original term-document matrix before performing SVD. This normalization can minimize the effects of common words that have no semantic meaning. Normalization can include methods such as:

- Term frequency – inverse document frequency (TF-IDF)
- Stopword removal
- Log – entropy normalization

Through these methods, words with higher semantic meaning are given more weight before performing SVD, dimension reduction, and reassembly.

This research uses LSA in the requirement analyzers to supply semantics to the requirement statements. The reduced term-document matrix provides a way to find relationships between terms and requirements. While LSA can find semantic relationships, it cannot identify the semantics. For instance, LSA can tell if a term is semantically related to another term but cannot directly identify the meaning of the term. To directly apply meaning to terms and requirements, LSA must be supplemented with another semantic tool.

2.4.2 Semantic Ontologies

Another NLP tool that can add linguistic semantics to a requirements analysis is a semantic ontology. In general, an ontology is a structured corpus. For instance, a regular corpus such as a stopword list is a list of words, while an ontological corpus can have relationships between items in the corpora. The structure of the ontology provides expert knowledge outside the domain of the requirements list. One example of such a corpus is WordNet [14]. WordNet provides relationships between words in its database through semantic relations such as a synonym. Other semantic relationships can also be found. Specifically, a semantic ontology can provide relationships between words beyond string matching algorithms. A sample list of semantic relations between words is provided in Table 2-2.

Table 2-2: Semantic relations between words [14].

Semantic Relation	Description
Synonym	A word has the same meaning as another word
Meronym	A word is a member of another word
Hyponym	A word is more specific instance of another word
Hypernym	A word is a more general instance of another word

The relationships in a corpus are defined by experts in the field of linguistics. The benefit of having these relationships is the enhanced ability to query the corpus. In a regular corpus, only string matching operations can be performed, while with an ontological corpus any number of query operations can be performed depending on the structure of the ontology. For example, the relationship between the component *automobile* and *bumper* can be identified where a non-semantic, string matching algorithm would not address. Figure 2-5 shows a sample of possible relationships that can be found.

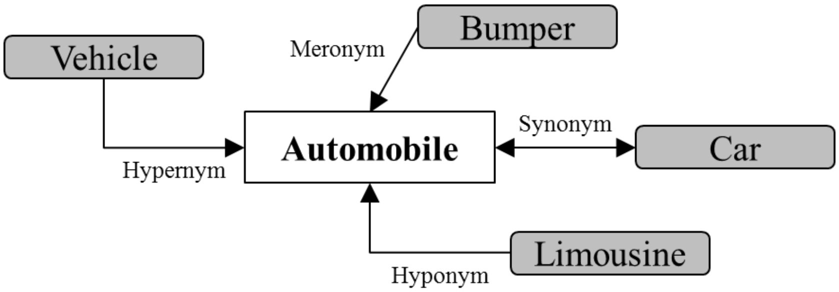


Figure 2-5: Semantic relationships between automobile and related terms.

In WordNet, it is possible to retrieve a similarity value between two words in the ontology by traversing the web of relationships in the ontology. Because an ontological corpus was used, these values have defined semantic relations and have context from experts in the field of linguistics.

In this research, one of the requirement analyzers is supplemented with a semantic ontology. This analyzer shows that a mapping to a semantic ontology can be performed. Using this mapping, the requirement analyzer is used to relate the components in the requirements. This result is then compared to the other analyzers and the established syntactic method.

2.5 Established Method for Gaining Conceptual Insight into Requirements

To provide valuable information to designers, a requirement analysis must gain a conceptual interpretation of the requirements. In particular, the requirement analyzers in this research are interested in finding the relationships between requirements. The relationships between requirements have been shown to provide valuable information to designers. Chen shows that relating requirements to one another via components can provide a model to aid the development of a product lifecycle model [9]. Morkos has shown that relationships between requirements can aid in change propagation prediction [12, 13]. Further, tools like IBM's DOORS™ tool and NASA's ARM tool [10] have identified the value of traceability to requirements. Traceability is the ability to find the source and relations between requirements. Two established methods for conceptual interpretation are formal models and quality metrics. These tools are discussed in light of this research.

2.5.1 Quality Metrics

Quality metrics judge the value of a requirement by checking a requirement for certain characteristics. Researchers have identified their own respective metrics to conform to their requirement analyzer, but a set of core metrics can be identified [15, 10, 11]. These metrics are listed in Table 2-3. This table shows the quality metrics and a description of each.

Table 2-3: List of core quality metrics [15].

Quality Metric	Description
Unambiguity	Requirement has same meaning to all readers
Conciseness	Requirement consists of only necessary details
Testability	Requirement has a method to check if it is met
Traceability	Requirement source can be traced and all links to other requirements are made
Consistency	Requirement has no overlap in content, terms in requirements are consistent
Correctness	Requirement does not contradict other requirements, standards, or physical laws
Completeness	All possible quantifications of a requirement have been made

The purpose of each metric is to provide a method to calculate the quality of a requirement and/or requirements document. However, many of these metrics are checked by using syntax and string matching algorithms. For instance, NASA's ARM tool checks for traceability of a requirement by detecting if a requirement is started with a string of numbers separated and terminated with periods [10]. Traceability can also be managed by a requirement tool such as DOORS™. However, DOORS™ requires the relationships between requirements to be identified manually. This manual identification can be tedious and is not scalable.

2.5.2 Formal Modeling

A formal model allows for domain knowledge to be applied to requirements. This application of domain knowledge allows the computer to perform conceptual interpretations. Formal models are often used to relate the conceptual elements in requirements—such as components—to one another. These relationships create an intelligent web of conceptual understanding that the requirement analyzer can process once the proper syntactic elements have been input into the formal model. Formal models can take on various forms. Some are ontologies that can analyze requirements using predicate logic [16]. Other formal models can identify ambiguity on a conceptual level [5]. Still other formal models attempt to create dependency repositories [9, 17, 18].

It has been shown that syntactic elements can be mapped to certain requirement concepts [19, 9, 7, 8]. For instance, the existence of a modal in a statement conveys the necessity of a requirement. Figure 2-6 shows how certain syntactic elements can be mapped to requirement concepts. The requirement concepts are the terms that add semantic and conceptual meaning to the requirement statement.

Syntactic Element		Requirement Element
Subject	→	Component
Modal	→	Necessity
Action Verb	→	Function
Object	→	Component
Adjunct	→	Condition

Figure 2-6: Adapted model of syntactic elements mapped to elements in a requirements statement [7, 8].

Of specific importance to this research is the identification of subjects and objects as they are semantically significant in requirements. Three of the requirement analyzers in this research create relationships between requirements based upon subjects and objects identified in the requirements. Subjects and objects have been used in established research in requirement analysis. For instance, Chen implements a component-centric formal model to aid in developing a product lifecycle model [9]. McLellan uses components to identify requirements critical to mass reduction [20, 21]. Also, research by Morkos in requirement change propagation relies on subjects and objects to create relationships between requirements [12, 13].

An issue with many formal models is that they forgo a semantic analysis. In other words, using the syntactic interpretation (parsing), the syntactic data is used to map directly to a formal model [5, 17, 18, 19, 9, 16]. Without a semantic understanding, a requirement analysis may not be as valuable. The formal model may be incomplete because it cannot identify semantic relationships between words such as synonymy and meronymy. Considering the design of an automobile, the component *suspension* is evidently related to the component *suspension* in another requirement. However, *suspension* is also related to the component *spring*. Considering change propagation through requirements, it is likely that a change to a *spring* requirement could change a *suspension* requirement. The research areas above do not use a semantic analysis and therefore cannot find relationships such as the example above. In particular, this research explores the value of a formal model supplemented with semantics to requirement analysis.

2.6 Research Opportunities

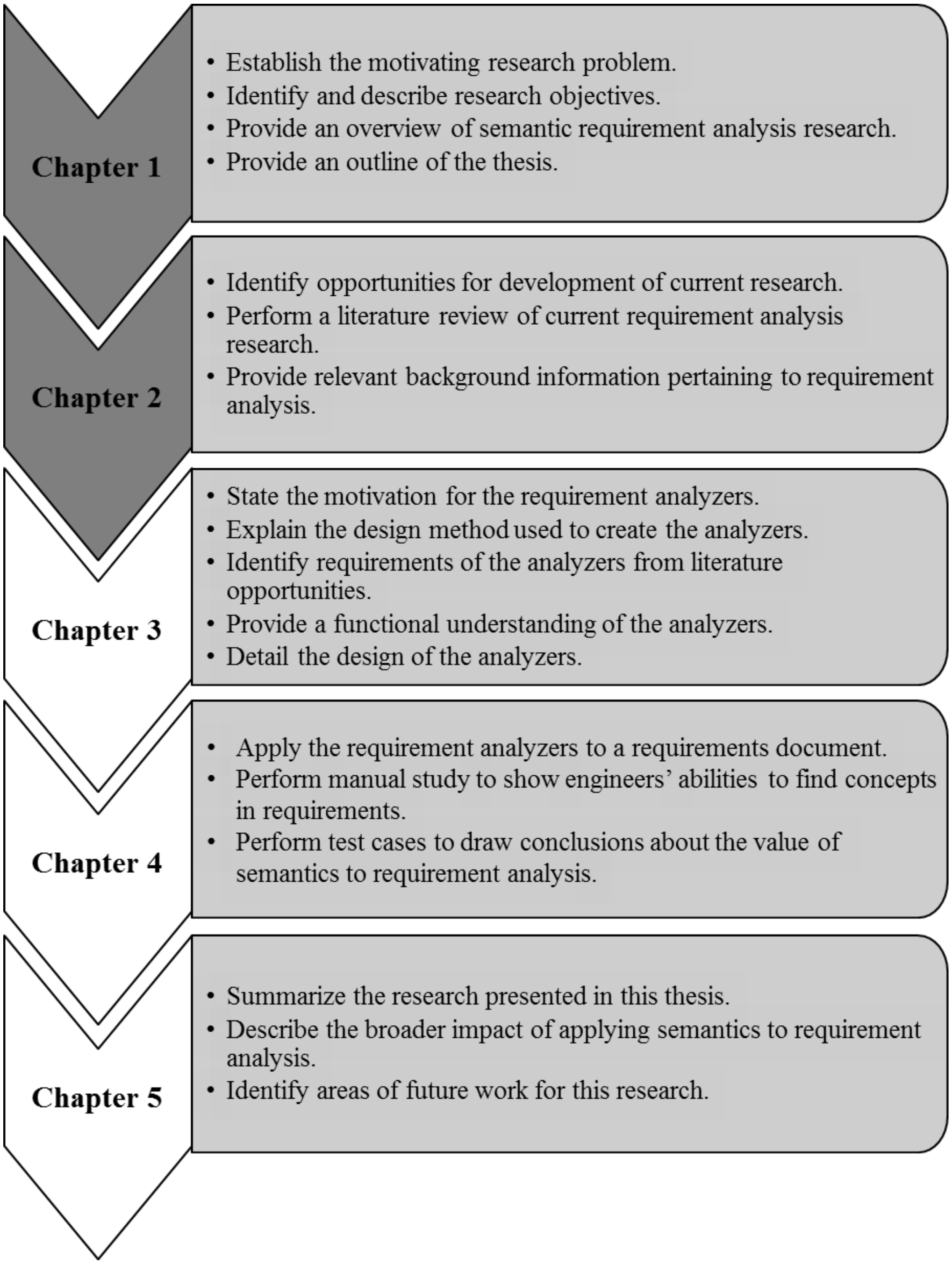
The literature review conducted in this chapter has provided an overview of relevant research in requirement analysis. From each topic reviewed, opportunities for further research have been identified.

Table 2-4: Topics reviewed and opportunities for research identified.

Section	Topic	Opportunity
2.1	Importance of Requirement Analysis	Requirement analysis can efficiently improve the quality of a requirements document and in turn, improve the design process.
2.2	Approach to NL Requirement Analysis	Linguistic approach provides a structured method for understanding NL requirement analysis.
2.3	Established NLP Methods for Requirement Analysis	Syntactic interpretation provides a base for extending requirement analysis to include semantics.
2.4	Semantic NLP Tools	Semantic NLP tools provide a way to enhance requirements with semantics.
2.5	Established Requirement Analysis Methods	Methods for requirement analysis use only syntax and string matching to conceptualize requirements.

2.7 Chapter Conclusions

In this chapter, a literature review of current research is performed and relevant background information pertaining to requirement analysis is identified. Further, opportunities have been identified for development of current research. In Chapter 3, these opportunities are translated into requirements of the requirement analyzers. Figure 2-7 shows the completed chapters (grey chevrons) and upcoming chapters (white chevrons) along with their respective deliverables.



Chapter 1

- Establish the motivating research problem.
- Identify and describe research objectives.
- Provide an overview of semantic requirement analysis research.
- Provide an outline of the thesis.

Chapter 2

- Identify opportunities for development of current research.
- Perform a literature review of current requirement analysis research.
- Provide relevant background information pertaining to requirement analysis.

Chapter 3

- State the motivation for the requirement analyzers.
- Explain the design method used to create the analyzers.
- Identify requirements of the analyzers from literature opportunities.
- Provide a functional understanding of the analyzers.
- Detail the design of the analyzers.

Chapter 4

- Apply the requirement analyzers to a requirements document.
- Perform manual study to show engineers' abilities to find concepts in requirements.
- Perform test cases to draw conclusions about the value of semantics to requirement analysis.

Chapter 5

- Summarize the research presented in this thesis.
- Describe the broader impact of applying semantics to requirement analysis.
- Identify areas of future work for this research.

Figure 2-7: Overview of thesis chapters showing chapters one and two completed (grey chevrons).

CHAPTER THREE
DESIGN OF THE REQUIREMENT ANALYZERS

Chapter Objectives:

1. State the motivation for the requirement analyzers.
2. Explain the design method used to create the analyzers.
3. Identify the requirements of the analyzers from literature opportunities.
4. Provide a functional understanding of the analyzers.
5. Detail the design of the analyzers.

This chapter details the development of the requirement analyzers. Three semantic requirement analyzers are designed. Further, one syntactic requirement analyzer is designed that represents methods used in existing research. All of the analyzers share the same syntactic interpretation, but each semantic analyzer interprets the semantics in the requirements in a different fashion.

Table 3-1: List of the requirement analyzers created including the Syntactic Analyzer (1) and three semantic analyzers (2-4).

ID	Analyzer	NLP Tools
1	Syntax	Tagging, Parsing
2	LSA	Tagging, Parsing, LSA
3	Component LSA	Tagging, Parsing, LSA
4	Semantic	Tagging, Parsing, LSA, Semantic Ontology

The analyzers are designed in three phases. First, task clarification converts the identified opportunities from Chapter Two to requirements of the analyzers. Next, conceptual design provides a functional model for the analyzers and the function algorithms. Then, the functions are programmed and integrated during embodiment and detail design. The analyzers are improved throughout the design process and final testing is performed before the solutions can be presented.

3.1 Requirements for Analyzers

The task clarification phase uses the identified opportunities in literature to create requirements for the requirement analyzers. Table 3-2 shows the requirements that address the opportunities identified from literature and which analyzers it applies to.

Table 3-2: List of requirements showing which analyzers the requirement applies to and the respective opportunity it addresses.

ID	Analyzers	Requirement	Opportunity
1	1, 2, 3, 4	Analyzer shall be automated.	Requirement analysis can efficiently improve the quality of a requirements document and in turn, improve the design process.
2	1, 2, 3, 4	Analyzer shall be developed based upon linguistic approach.	Linguistic approach provides a structured method for understanding NL requirement analysis.
3	1, 2, 3, 4	Analyzer shall use established syntactic approach to requirement analysis.	Syntactic interpretation provides a base for extending requirement analysis to include semantics.
4	2, 3, 4	Analyzer shall implement semantic tools.	Semantic NLP tools provide a way to enhance requirements with semantics.
5	1	Analyzer shall implement only syntactic tools to provide a baseline analysis.	Methods for requirement analysis use only syntax and string matching to conceptualize requirements.

3.2 Algorithm Development

Once the requirements have been elaborated, the functional models for each analyzer can be derived from the analyzer requirements, and the function algorithms for each analyzer are formulated. This conceptual design section presents the functional models and function algorithms for each analyzer individually with one exception. The syntactic analysis method is the same across all analyzers. Therefore, this design is presented only once.

3.2.1 Syntactic Analysis

Section 2.3 has shown that dependency trees provide an automated method to identify syntax in requirements. The syntactic analysis inputs raw NL requirements and outputs these dependency trees. The functional model of these requirements is presented in Figure 3-1. This analysis serves as the foundation for all of the requirement analyzers.

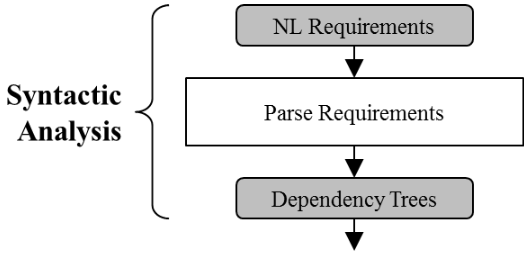


Figure 3-1: Functional model of the syntactic analysis of all requirement analyzers.

In Figure 3-1, each white box represents a function and each grey box represents an input and/or output of each function. The dependency trees are created via the Stanford Parser [6]. As shown in Figure 3-2 parsing the requirements requires three sub-functions.

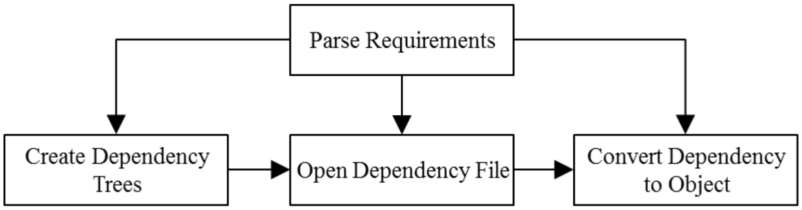


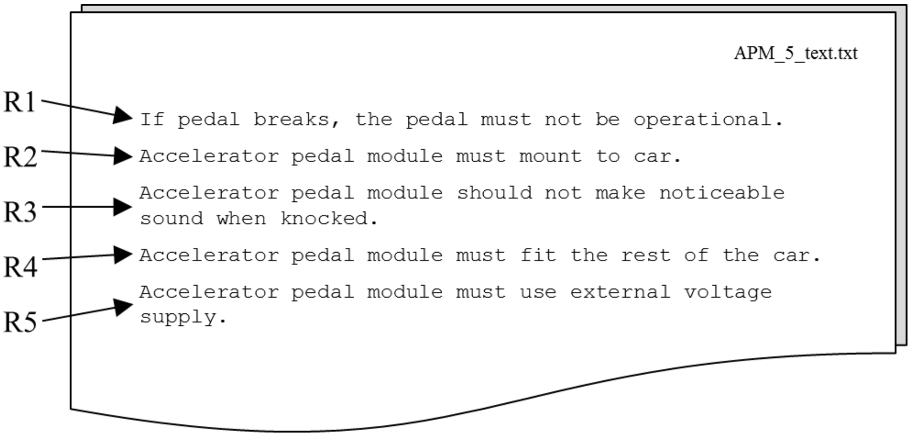
Figure 3-2: Functional decomposition of requirement parsing.

The algorithm to parse the requirements is shown in Table 3-3:

Table 3-3: Algorithm for parsing requirements.

Function	Parse Requirements		
Input	NL requirements	**Type**	.txt file
Parse text file into dependency trees. *Open dependency tree text file.* *Convert dependency tree text file to objects.*			
Output	Dependency trees	**Type**	List of objects

A sample set of five requirements for a BMW accelerator pedal module have been applied to this function. As shown in Figure 3-3, the input is a list of the NL requirements in a text file.



R1
R2
R3
R4
R5

APM_5_text.txt

```
If pedal breaks, the pedal must not be operational.
Accelerator pedal module must mount to car.
Accelerator pedal module should not make noticeable
sound when knocked.
Accelerator pedal module must fit the rest of the car.
Accelerator pedal module must use external voltage
supply.
```

Figure 3-3: Text file of sample set of requirements for the accelerator pedal module.

The output is a list of dependency objects within the program. A sample dependency tree for the second requirement in Figure 3-3 is shown in Figure 3-4.

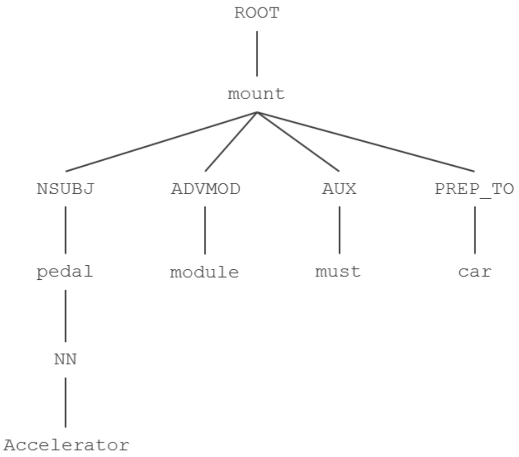


Figure 3-4: Sample dependency parse of a NL requirement.

3.2.2 Syntax Analyzer

The Syntax Analyzer does not incorporate semantics, but uses syntax and string matching alone to create a conceptual model of the requirements. Figure 3-5 shows the functional model of the Syntax Analyzer. The syntactic analysis has already been detailed in Section 3.2.1 and thus is not shown in the model.

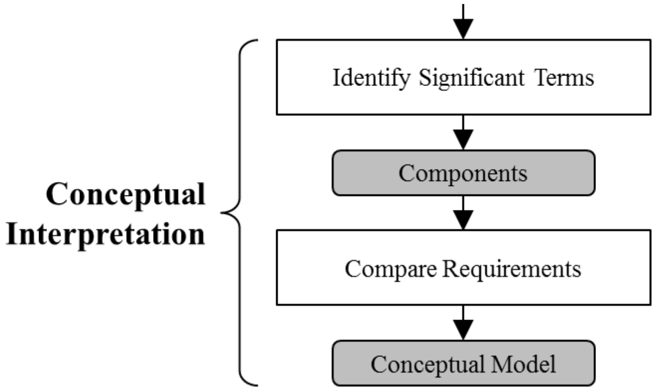


Figure 3-5: Functional model of the Syntax Analyzer with implied syntactic analysis.

The functional model provides the structure for the Syntax Analyzer. Now, each function must be conceptually defined. The algorithms for functional models provided in Figure 3-5 are shown along with inputs/outputs and any sub-functions. These functions are applied to a sample set of requirements to aid the understanding of functions.

3.2.2.1 Term Identification

Once the requirements have been parsed and dependency tree objects have been created, the significant terms can be identified. Section 2.5.2 showed that the syntactic elements of a requirement can be mapped to requirement elements. To find the syntactic elements the dependency trees must be traversed. As shown in Figure 3-6, identifying the significant terms has two sub-functions.

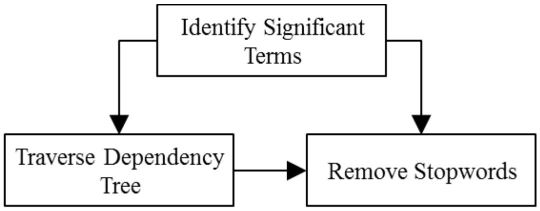


Figure 3-6: Functional decomposition of significant term identification.

The algorithm to identify significant terms is shown in Table 3-4.

Table 3-4: Algorithm for identifying significant terms.

Function	Identify Significant Terms		
Input	Dependency trees	**Type**	List of objects
For each dependency tree: *Call Traverse Dependency Tree function.* *Remove duplicates from key terms* *Append key terms list to a list of list.* *Call Remove Stopwords Function.*			
Output	Key terms by requirement	**Type**	List of list of strings

The key terms are selected by identifying the subjects (NSUBJ) and direct objects (DOBJ) of the requirements through the dependency tags. Figure 3-7 shows a visualization of this selection process performed on the dependency tree from the previous example.

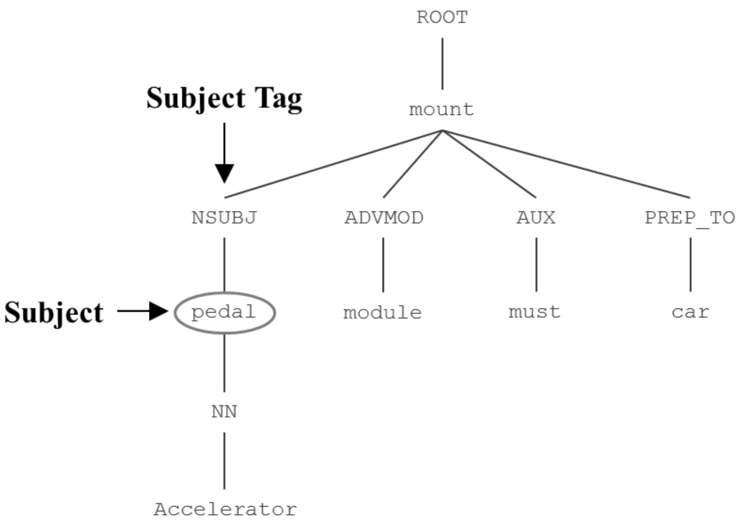


Figure 3-7: Visualization of selection process for identifying significant words from dependency trees.

For all of the sample requirements the identified key terms are:

1. *pedal*/NSUBJ
2. *pedal*/NSUBJ
3. *pedal*/NSUBJ, *sound*/DOBJ
4. *pedal*/NSUBJ, *rest*/DOBJ
5. *pedal*/NSUBJ, *supply*/DOBJ

Once the syntax has been used to identify key terms, a semantic analysis can be performed on these key terms to give semantic meaning to the requirements.

3.2.2.2 Conceptual Modeling

The conceptual model relates the requirements based on the identified strings. This analyzer calculates a binary value on whether or not the strings match between requirements. Using the values derived between requirements, a design structure matrix

(DSM) is used to model the requirements. The conceptual analysis is performed via a comparison and modeling of the requirements as shown in Figure 3-27.

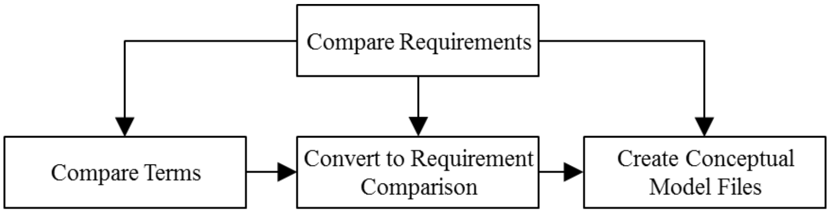

Compare Requirements
Compare Terms
Convert to Requirement Comparison
Create Conceptual Model Files

Figure 3-8: Functional decomposition of requirement comparison in Syntax Analyzer.

The algorithm to perform conceptual modeling is given in Table 3-16.

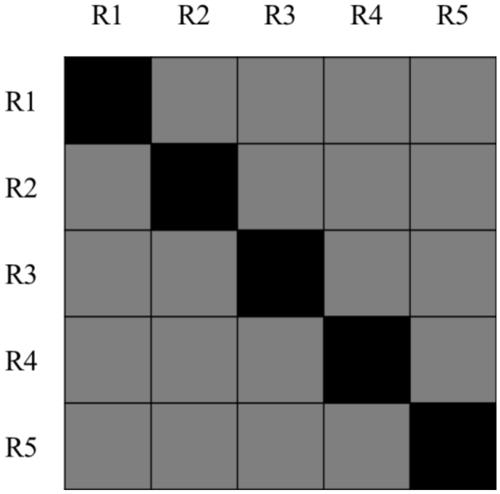
Table 3-5: Algorithm for comparing requirements.

Function	Compare Requirements		
Input	Terms by requirement	**Type**	List of list of objects

```
For each pair of requirements:
    For each pair of terms across requirements:
        If terms match:
            Add 1 to requirement similarity value.
    Append requirement similarity value to a requirement DSM.
Create conceptual model files from requirement DSM.
```

Output	DSM conceptual model	**Type**	.csv file

Once the terms for each requirement are input, the terms across each requirement pair are compared to see if the terms are a string match. An example of the output DSM can be seen in Figure 3-28.



	R1	R2	R3	R4	R5
R1	■	▨	▨	▨	▨
R2	▨	■	▨	▨	▨
R3	▨	▨	■	▨	▨
R4	▨	▨	▨	■	▨
R5	▨	▨	▨	▨	■

Figure 3-9: DSM of sample requirements showing relationships between requirements (grey boxes) found via the Syntax Analyzer.

The DSM is symmetric, so the upper and lower triangular matrices show the same information. Along the diagonal, the requirements are compared to themselves and are evidently related. The sample DSM shows that all requirements are related because every requirement has the word *pedal* identified. The purpose of this example is to detail how the analysis is performed. Chapter Four applies the requirement analyzers to a complete requirements document.

3.2.3 LSA Analyzer

This analyzer incorporates LSA to create relationships between requirements. Figure 3-10 shows the functional model of the LSA Analyzer. As with the Syntax Analyzer functional model, the syntactic analysis is not explicitly shown but is how the dependency trees are created.

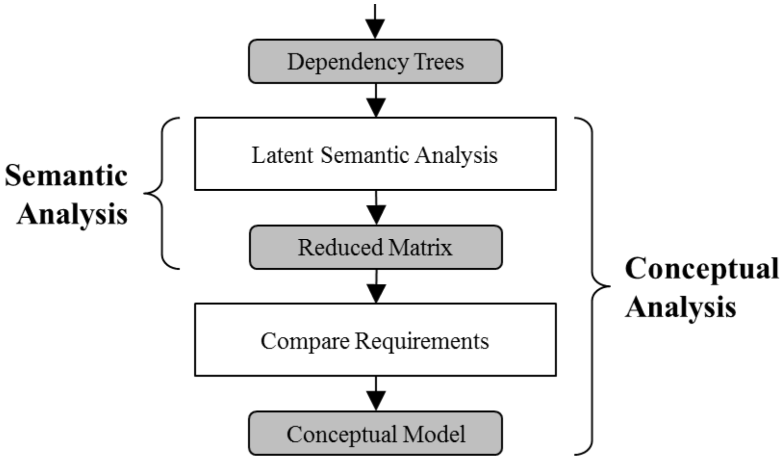


Figure 3-10: Functional model of the LSA Analyzer with implied syntactic analysis.

From the functional model, the two functions in Figure 3-10 are defined. To aid understanding, the five sample accelerator pedal module requirements are applied to this analyzer.

3.2.3.1 Latent Semantic Analysis

The tokens from the dependency trees are used as input for the LSA. Running LSA provides a way to computationally measure similarity between terms in the requirements. The function for this algorithm is provided in Table 3-13.

Table 3-6: Algorithm for running LSA in the LSA Analyzer.

Function	Run Latent Semantic Analysis		
Input	All terms by requirement	**Type**	List of list strings
Create term-document matrix using term frequency. *Normalize term-document matrix using log-entropy model.* *Decompose term-document matrix.* *Rank-reduce decomposed matrices.* *Reconstructed term-document matrix with reduced matrices.*			
Output	Reduced term-document matrix	**Type**	List of lists of floats

The input for running LSA is all the terms in each requirement. In other words, each term from each dependency tree is fed into the function to create the term-document matrix. Figure 3-23 shows the process for inputting a sample accelerator pedal module requirement into the LSA function.

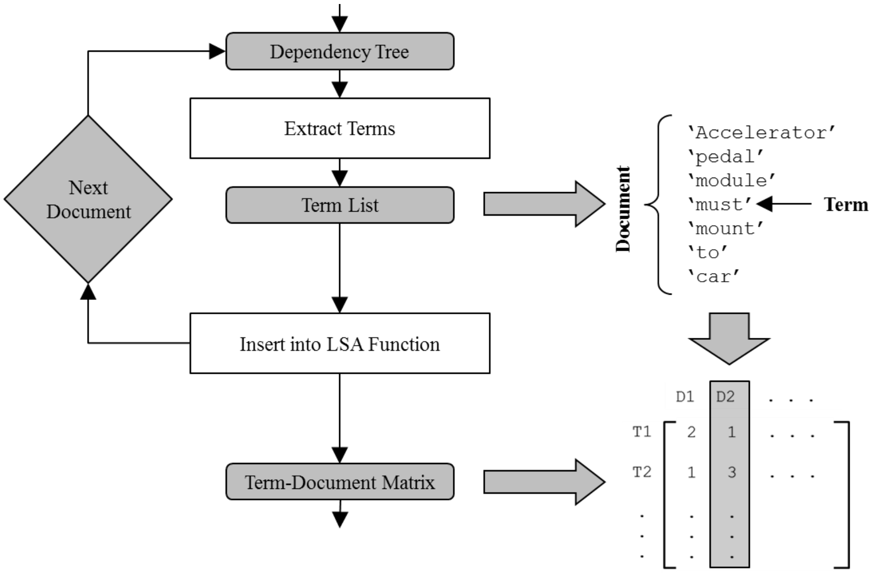


Figure 3-11: Workflow of inputting requirements into LSA (left) showing an example requirement (right).

Once the terms are inserted into the term-document matrix, the LSA is performed via singular value decomposition (SVD). The output is a rank-reduced term-document matrix. The rank lowering decreases the complexity of the matrix to bring forth the latent relationships. Returning to the five accelerator pedal module requirements example, Figure 3-12 shows a sample portion of the outputted rank-reduced term-document matrix.

$$\begin{array}{r|ccccc|}
 & \text{R1} & \text{R2} & \text{R3} & \text{R4} & \text{R5} \\
\text{accelerator} & .01 & .14 & .19 & .18 & .23 \\
\text{supply} & -.07 & .09 & .01 & .10 & .16 \\
\text{module} & .01 & .14 & .19 & .18 & .23 \\
\text{voltage} & -.07 & .09 & .01 & .10 & .16 \\
\text{rest} & .08 & .07 & -.02 & .13 & .10 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{array}$$

Figure 3-12: Sample of the reduced term-document matrix performed on five accelerator pedal module requirements.

The reduced term-document matrix relates the requirements and the terms in the requirements. Each value represents the semantic importance of the respective term to the respective requirement. For instance, Figure 3-12 shows that the terms *accelerator* and *module* have the highest displayed importance to requirement three (*R3*), while *rest* has the least displayed importance. With the latent semantics identified through the reduced term-document matrix, the conceptual model can be made by comparing column vectors.

3.2.3.2 Conceptual Modeling

Using the reduced-term-document matrix, semantic relationships between requirements are formed. Figure 3-13 shows the sub-functions for comparing requirements to create the conceptual model.

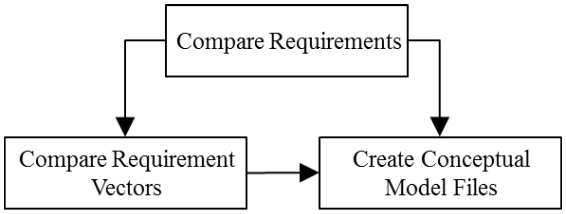


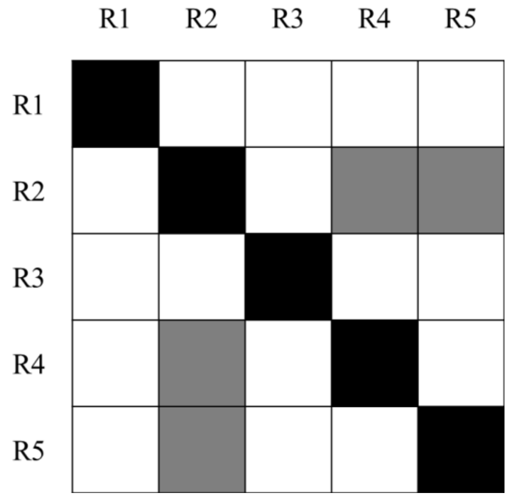
Figure 3-13: Functional decomposition for comparing requirements in LSA Analyzer.

The algorithm to compare requirements in the LSA Analyzer is shown in Table 3-7.

Table 3-7: Algorithm for comparing requirements to create conceptual model in LSA Analyzer.

Function	Compare Requirements		
Input	Reduced term-document matrix	**Type**	List of list of floats
For each pair of column vectors: *Compute cosine similarity requirement between requirements.* *Append requirement similarity value to a requirement DSM.* *Create conceptual model files from requirement DSM.*			
Output	DSM conceptual model	**Type**	.csv file

Similar to the conceptual model of the Syntax Analyzer created in Section 3.2.2.2, the LSA Analyzer outputs a DSM of the relationships between requirements. This conceptual model is shown in Figure 3-14.



	R1	R2	R3	R4	R5
R1	■				
R2		■		▨	▨
R3			■		
R4		▨		■	
R5		▨			■

Figure 3-14: DSM of sample requirements showing relationships between requirements (grey boxes) found via the LSA Analyzer.

The relationships found by the LSA Analyzer are different than those found by the Syntax Analyzer. Chapter Four draws comparisons between these analyzers as well as the other two analyzers.

3.2.4 Component LSA Analyzer

Similar to the LSA Analyzer, the Component LSA Analyzer also uses a latent analysis to interpret semantics in the requirements. The Component LSA Analyzer on components only. This component-centric analysis is consistent with the other requirement analyzers discussed previously—Syntax Analyzer (Section 3.2.2) and Semantic Analyzer (Section 3.2.5). Figure 3-15 shows the functional model of the Component LSA Analyzer except the syntactic analysis, which is detailed in Section 3.2.1.

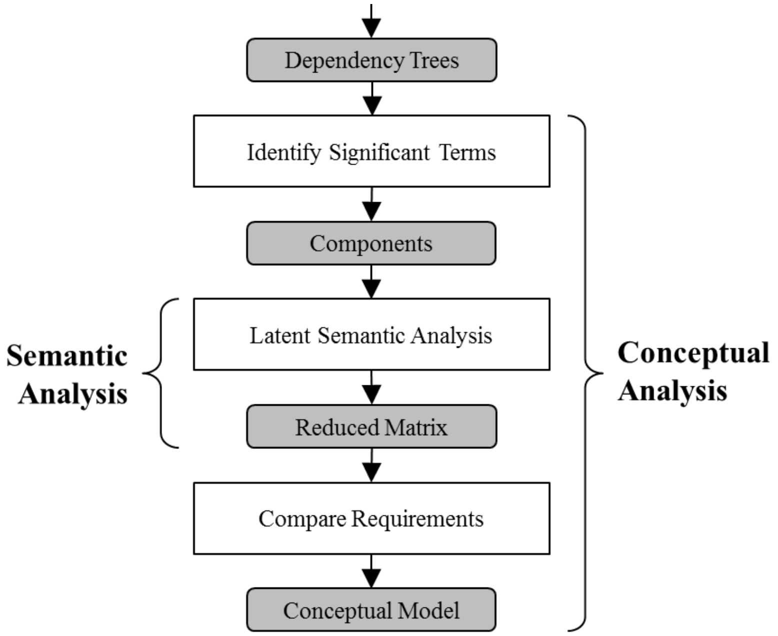


Figure 3-15: Functional model of the component LSA Analyzer with implied syntactic analysis.

From the functional model, each function is decomposed and algorithms are detailed. As with the previous analyzers, the sample accelerator pedal module requirements are analyzed to aid understanding.

3.2.4.1 Term Identification

Identifying significant terms is identical to the algorithm performed for the Syntax Analyzer. This functional decomposition and algorithm is provided in Section 3.2.2.1. The inputs and outputs of the function are reiterated in Table 3-8.

Table 3-8: Inputs and outputs of the term identification function.

Function	Identify Significant Terms		
Input	Dependency trees	**Type**	List of objects
Output	Key terms by requirement	**Type**	List of list of strings

3.2.4.2 Latent Semantic Analysis

The significant terms from the dependency trees are used as input for the LSA. The LSA function for the Component LSA Analyzer is different from the LSA function of the LSA Analyzer described in Section 3.2.3.1. The function for this algorithm is provided in Table 3-9.

Table 3-9: Algorithm for running LSA in the component LSA analyzer.

Function	Run Latent Semantic Analysis		
Input	Significant terms by requirement	**Type**	List of list strings
Create term-document matrix using term frequency. *Decompose term-document matrix.* *Rank-reduce decomposed matrices.* *Reconstructed term-document matrix with reduced matrices.*			
Output	Reduced term-document matrix	**Type**	List of lists of floats

Since the input terms are all of significance to the requirement, there is no need to normalize the term-document matrix. Figure 3-16 shows the process for inputting a sample accelerator pedal module requirement into the LSA function.

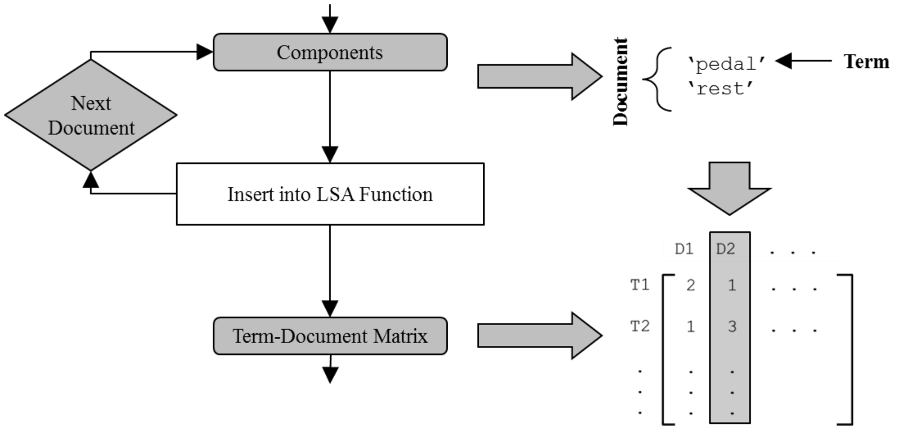


Figure 3-16: Workflow of inputting requirements into LSA (left) showing an example requirement (right).

Once the terms are inserted into the term-document matrix, the LSA is performed via singular value decomposition (SVD). Returning to the five accelerator pedal module requirements example, Figure 3-17 shows the outputted rank-reduced term-document matrix.

$$
\begin{array}{r|rrrrr|}
 & \text{R1} & \text{R2} & \text{R3} & \text{R4} & \text{R5} \\
\text{sound} & .19 & .19 & .90 & -.10 & -.10 \\
\text{pedal} & .88 & .88 & 1.07 & 1.07 & 1.07 \\
\text{rest} & .19 & .19 & -.10 & .40 & .40 \\
\text{supply} & .19 & .19 & -.10 & .40 & .40
\end{array}
$$

Figure 3-17: Reduced term-document matrix performed on five accelerator pedal module requirements.

With the latent semantics identified through the reduced term-document matrix, the conceptual model can be made by comparing column vectors.

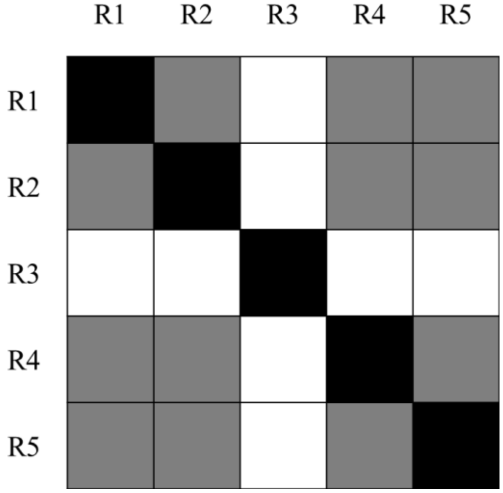
3.2.4.3 Conceptual Modeling

The function for creating the conceptual model of the requirements in the component LSA Analyzer is identical to the function in the LSA Analyzer. This function is detailed in Section 3.2.3.2. The function uses the reduced-term-document matrix to create semantic relationships between requirements. The inputs and outputs from comparing requirements in the component LSA Analyzer are shown in Table 3-10.

Table 3-10: Inputs and outputs of the compare requirements function.

Function	Compare Requirements		
Input	Reduced term-document matrix	**Type**	List of list of floats
Output	DSM conceptual model	**Type**	.csv file

Similar to the conceptual model of the previous analyzers, the component LSA Analyzer outputs a DSM of the relationships between requirements. This conceptual model is shown in Figure 3-18.



	R1	R2	R3	R4	R5
R1	■	▨	□	▨	▨
R2	▨	■	□	▨	▨
R3	□	□	■	□	□
R4	▨	▨	□	■	▨
R5	▨	▨	□	▨	■

Figure 3-18: DSM of sample requirements showing relationships between requirements (grey boxes) found via the component LSA Analyzer.

3.2.5 Semantic Analyzer

The Semantic Analyzer incorporates both LSA and a semantic ontology. Figure 3-19 shows the functional model of this analyzer. As with the previous models, the syntactic analysis has already been detailed in Section 3.2.1 and thus is not shown in the model.

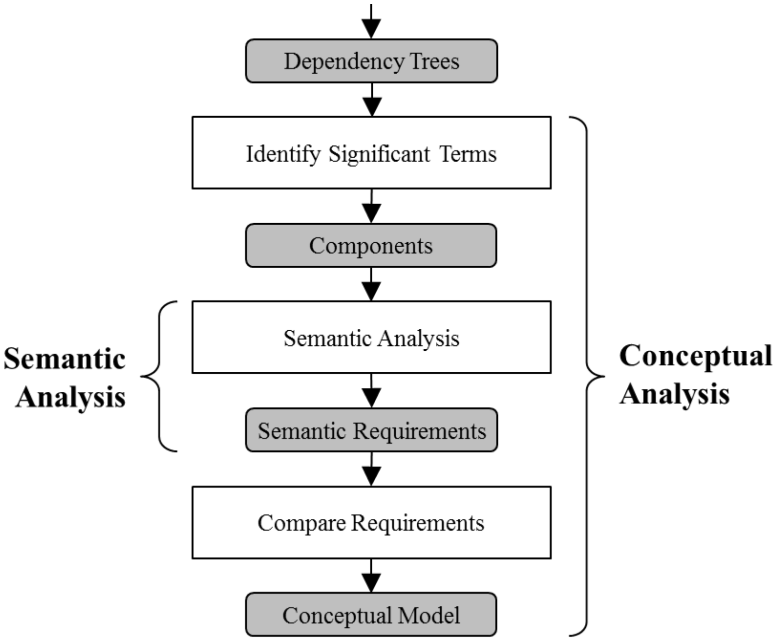


Figure 3-19: Functional model of the Semantic Analyzer with implied syntactic analysis.

First, requirements are parsed and dependency trees are created for each requirement. Next, significant terms are identified. Using the identified significant terms, the semantic analysis is performed. The semantic analysis applies semantic meaning to the significant terms. The sub-functions of the semantic analysis are shown in Figure 3-20.

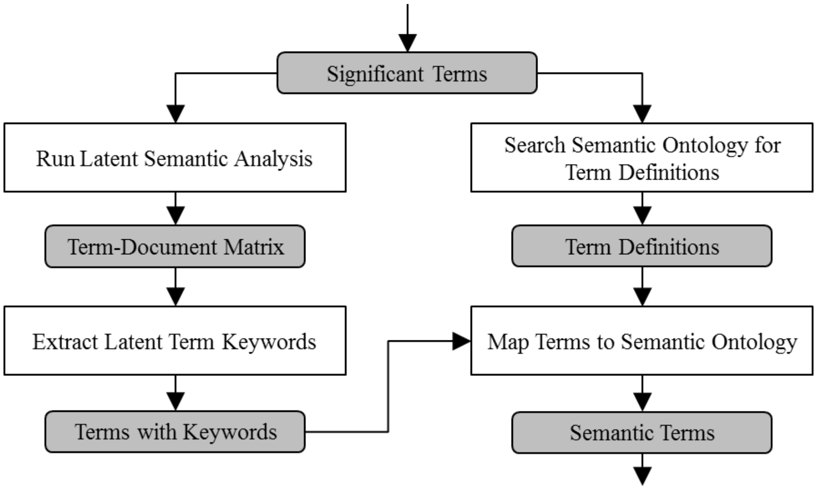

Significant Terms
Run Latent Semantic Analysis
Search Semantic Ontology for Term Definitions
Term-Document Matrix
Term Definitions
Extract Latent Term Keywords
Map Terms to Semantic Ontology
Terms with Keywords
Semantic Terms

Figure 3-20: Semantic analysis sub-model showing sub-functions.

The algorithms for functional models provided in and Figure 3-20 are shown along with inputs/outputs and any sub-functions. These functions are applied to a sample set of requirements to aid the understanding of functions.

3.2.5.1 Term Identification

Identifying significant terms is identical to the algorithm performed for the Syntax Analyzer and component LSA Analyzer. This functional decomposition and algorithm is provided in Section 3.2.2.1. The inputs and outputs of the function are reiterated in Table 3-11.

Table 3-11: Inputs and outputs of the term identification function.

Function	Identify Significant Terms		
Input	Dependency trees	**Type**	List of objects
Output	Key terms by requirement	**Type**	List of list of strings

3.2.5.2 Semantic Analysis

Figure 3-20 shows that the semantic analysis is broken down into four main sub-functions. These sub-functions are provided in a functional model in Figure 3-21.

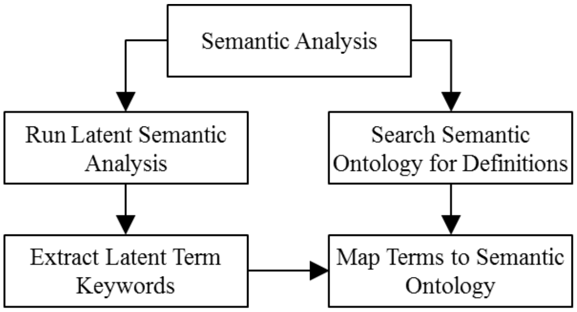

Semantic Analysis
Run Latent Semantic Analysis
Search Semantic Ontology for Definitions
Extract Latent Term Keywords
Map Terms to Semantic Ontology

Figure 3-21: Functional decomposition of the semantic analysis.

The semantic analysis supplies the terms identified as significant terms with semantic meaning thereby providing the requirement with semantics. This action is performed by the function *Search Semantic Ontology for Term Definitions*. In this function, each term is found in the semantic ontology and all semantically related words are captured for each possible definition. A particular meaning of a term within the semantic ontology is called a synset. A synset encapsulates all syntactic and semantic information about a particular term. The algorithm for this function is given in Table 3-12.

Table 3-12: Algorithm for searching the semantic ontology for term definitions.

Function	Search Semantic Ontology for Term Definitions		
Input	Key terms	**Type**	List of strings

```
For term in key terms:
    Find all possible synsets.
    For synset in possible synsets:
        Find all keywords of a synset.
        Append keywords to a list.
    Append keywords list to a list.
    Place list of lists into a dictionary with term as key.
```

Output	Dictionary of possible definitions	**Type**	Dictionary of lists of lists of strings

The keywords for each synset are derived from the lemmas (synonyms), definitions, and/or example sentences. Continuing the example from Section 3.2.5.1, the input for this function is each significant term from the requirements document. These inputs are:

- *pedal*
- *sound*
- *rest*
- *supply*

After the function is performed, the output is a dictionary of possible synsets for a word. For instance, the possible synsets for the term *sound* are shown in Figure 3-22.

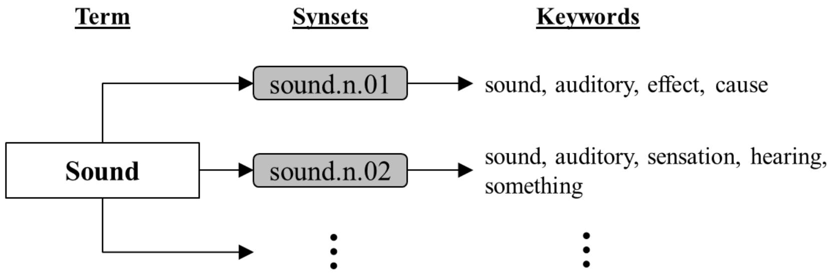

Term
Synsets
Keywords
Sound
sound.n.01
sound.n.02
sound, auditory, effect, cause
sound, auditory, sensation, hearing, something

Figure 3-22: Sample synsets for the term *sound* showing keywords found for each synset.

Next, a specific meaning (synset) must be identified from all of the possible meanings of a word. Context from the requirements document aids in this process to help ensure the correct synset is chosen. In this semantic analysis, context is provided via the functions *Run Latent Semantic Analysis* and *Extract Latent Term Keywords*. Running LSA provides a way to computationally measure semantic similarity between terms in the requirements. The function for this algorithm is provided in Table 3-13.

Table 3-13: Algorithm for running LSA.

Function	Run Latent Semantic Analysis		
Input	All terms by requirement	**Type**	List of list strings
Create term-document matrix using term frequency. *Normalize term-document matrix using log-entropy model.* *Decompose term-document matrix.* *Rank reduce decomposed matrices.* *Reconstructed term-document matrix with reduced matrices.*			
Output	Reduced term-document matrix	**Type**	List of lists of floats

The input for running LSA is all the terms in each requirement. In other words, each requirement is broken into terms and fed into the function to create the term-document matrix. Figure 3-23 shows the process for inputting the requirement into the LSA function. A sample requirement is shown on the right.

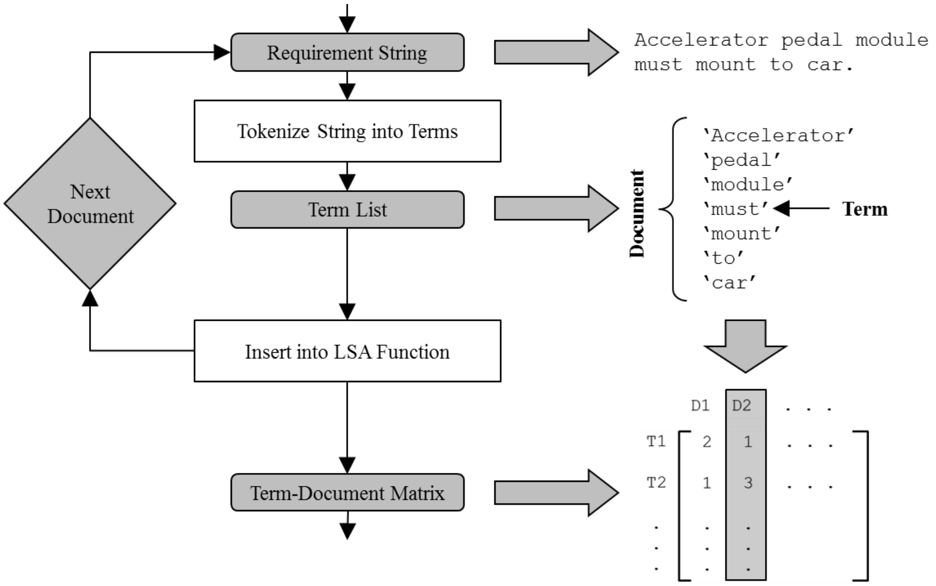


Figure 3-23: Workflow of inputting requirements into LSA (left) showing an example requirement (right).

The output is a rank-reduced term-document matrix. Similar to the original matrix, the reduced matrix limits the rank to decrease noise. By decreasing this variance, the latent relationships can be found.

Once LSA is complete, the terms within the term-document matrix identified as most related to the significant terms are then extracted. In this way, context for each significant term is applied. The algorithm for this function is shown in Table 3-14.

Table 3-14: Algorithm for extracting latent term keywords.

Function	Extract Latent Term Keywords		
Input	Reduced term-document matrix	**Type**	List of list of floats

```
For each pair of rows in reduced term-document matrix:
    Compute cosine similarity of row vectors.
    Append similarity value into design structure matrix.
For each value in design structure matrix:
    If value greater than similarity cutoff value:
        Relate corresponding terms to one another.
Create dictionary of related terms to a given term.
Modify dictionary to only contain significant terms as keys.
```

Output	Dictionary of latently related terms	**Type**	Dictionary of lists of strings

The output of the function is a dictionary of latently related terms for each significant term in the requirements. Each significant term has a key and a value in the dictionary. The example in Figure 3-24 shows a sample of the dictionary for the significant term *sound*.

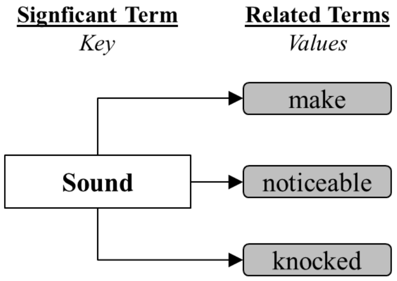


Figure 3-24: Example of terms latently related to the significant term *sound*.

Once context has been applied and possible definitions are known, the meaning for an unknown term can be found by mapping to a term to the semantic ontology. The function *Map Terms to Semantic Ontology* uses the found context to apply meaning to the terms (i.e. define the terms). In the semantic analysis, context is applied by relating the terms extracted from the requirements document to the terms captured for each possible definition. The definition that the context terms are most related to is selected as the definition of the word in the semantic ontology. The algorithm for this mapping is shown in Table 3-15.

Table 3-15: Algorithm for mapping terms to the semantic ontology.

Function	Map Terms to Semantic Ontology		
Inputs	Dictionary of latently related terms	**Types**	Dictionary of lists of strings
	Dictionary of possible definitions		Dictionary of lists of lists of strings

```
For term in latently related terms:
    Look up possible definitions.
    For each possible definition:
        Compare to latently related terms.
    For each comparison:
        If the comparison is the most similar:
            Append synset to a dictionary with term as key.
Replace terms by requirement with semantic terms.
```

Output	Semantic terms by requirement	**Type**	List of list of objects

An example mapping for the term *sound* is provided in Figure 3-25.

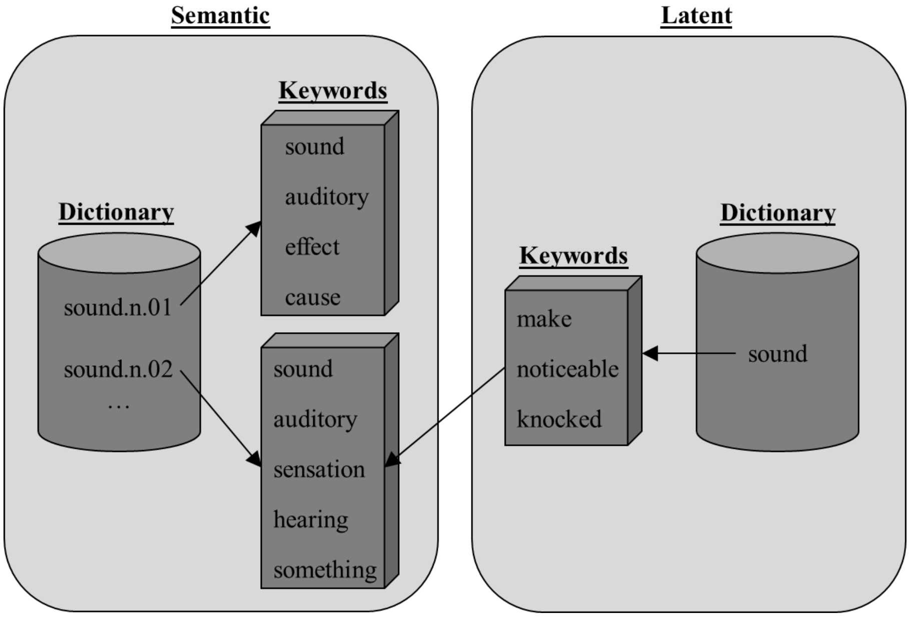


Figure 3-25: Example mapping of significant term *sound* (right) to the semantic ontology (left) by comparing keywords.

The keywords from the latent analysis are related to the keywords from the semantic ontology. In the example in Figure 3-25, *sound* is mapped to the synset *sound.n.02* because the relationship between the latent and semantic keywords is stronger. The output for all requirements is the significant semantic terms for each requirement. This full output is shown in Figure 3-26.

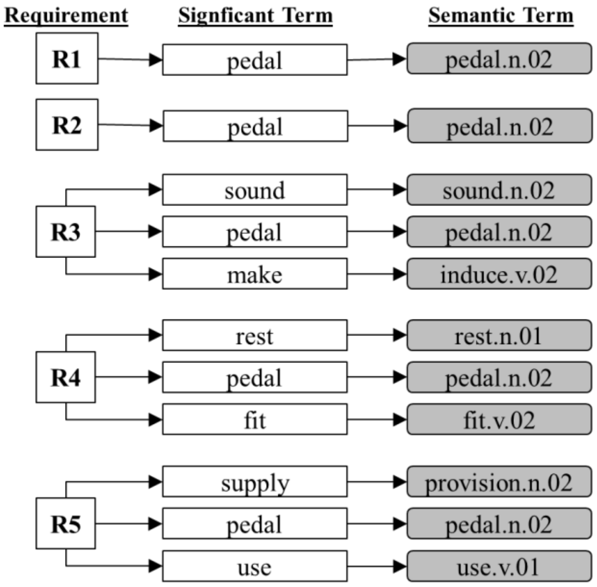


Figure 3-26: Output of semantic analysis showing significant terms (middle) mapped to the synsets in the semantic ontology (right).

Once semantics have been applied to the requirements, conceptual modeling can be performed.

3.2.5.3 Conceptual Modeling

This analyzer calculates a similarity value based on semantics between any two requirements. Using the similarity metric derived between requirements, a DSM is used to model the requirements. The conceptual analysis is performed via a comparison and modeling of the requirements as shown in Figure 3-27.

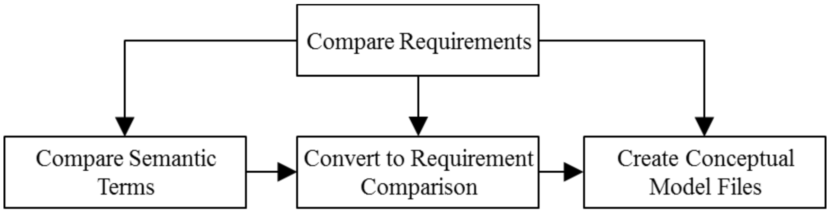

Compare Requirements
Compare Semantic Terms
Convert to Requirement Comparison
Create Conceptual Model Files

Figure 3-27: Functional decomposition of requirement comparison.

The algorithm to perform conceptual modeling is given in Table 3-16.

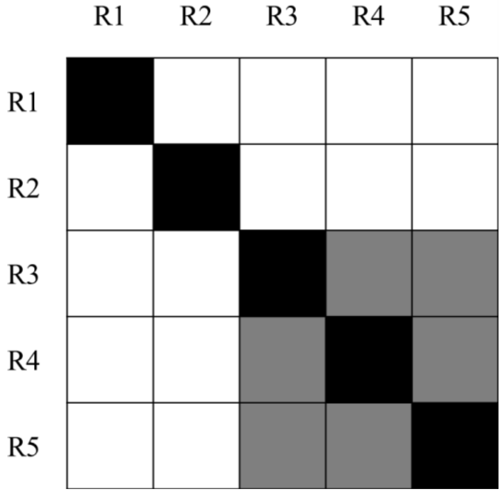
Table 3-16: Algorithm for comparing requirements.

Function	Compare Requirements		
Input	Semantic Terms by Requirement	**Type**	List of list of objects

```
For each pair of requirements:
    For each pair of semantic terms across requirements:
        Traverse semantic ontology to find similarity.
        Add similarity value to requirement similarity value.
    Append requirement similarity value to a requirement DSM.
Create conceptual model files from requirement DSM.
```

Output	DSM conceptual model	**Type**	.csv files

Once the semantic terms for each requirement are input, the terms across each requirement pair are compared to find the semantic similarity between requirements. An example of the output DSM can be seen in Figure 3-28.



	R1	R2	R3	R4	R5
R1	■				
R2		■			
R3			■	▨	▨
R4			▨	■	▨
R5			▨	▨	■

Figure 3-28: Sample DSM showing relationships between requirements (grey boxes) for the Semantic Analyzer.

3.3 Code Implementation

Embodiment of the design entails programming the functions from the algorithms and integrating the functions together. The computational tools and programming language are implemented. To integrate the functions, the analyzers must be tested and iterations of the function programs must be performed when integration issues arise.

Using the algorithms from Section 3.2, the functions are coded. The code for all functions is provided in Appendix A. The requirement analyzers are programmed in Python [22]. The computational tools used to support the requirement analyzers are the Stanford Parser, Natural Language Toolkit (NLTK), and WordNet [6, 14, 23]. Table 3-17 provides an overview of these tools and where they are used in the program.

Table 3-17: Description of computational tools used in the requirement analyzers [6, 14, 23].

Computational Tool	Description	Functions Where Applied
Stanford Parser	Parser that provides dependency graphs for syntactic understanding.	Parse Requirements
NLTK	Set of NLP methods that provide access to corpora and semantic tools.	Semantic Analysis
WordNet	Semantic ontology that is mapped to for semantic understanding.	Semantic Analysis, Compare Requirements

Embodied functional models with functions, classes, and methods identified are provided. Each function in the grey boxes corresponds with the function coding in the Appendix. The embodied function model for the Syntax Analyzer is shown in Figure 3-29.

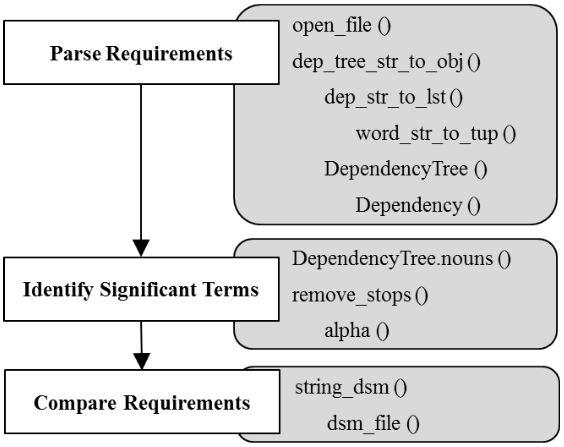


Figure 3-29: Embodied function model of the Syntax Analyzer showing functions, classes, and methods.

The embodied function model for the LSA Analyzer is shown in Figure 3-30.

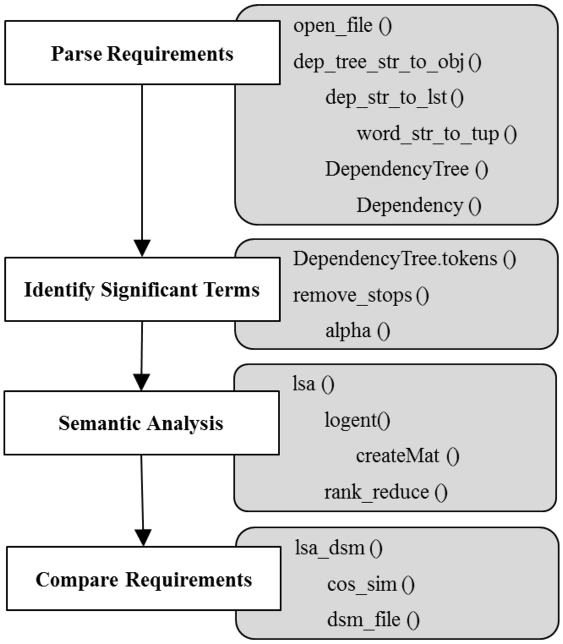


Figure 3-30: Embodied function model of the LSA Analyzer showing functions, classes, and methods.

The embodied function mode for the component LSA Analyzer is shown in Figure 3-31.

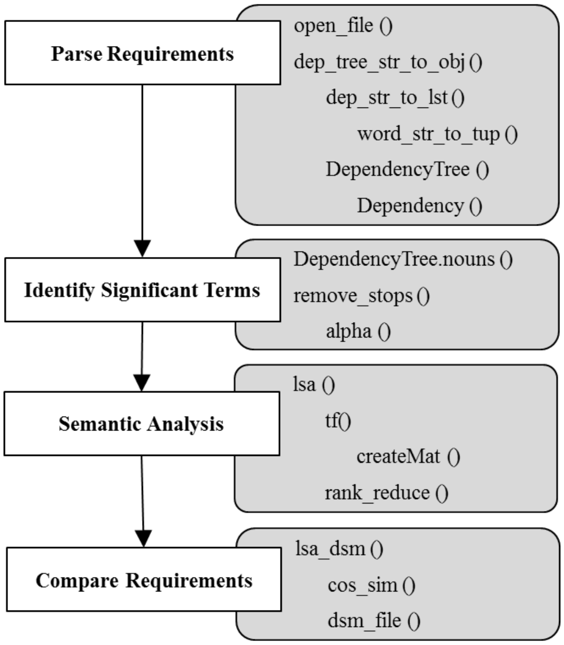


Figure 3-31: Embodied function model of the component LSA Analyzer showing functions, classes, and methods.

The embodied function model for the Semantic Analyzer is shown in Figure 3-32.

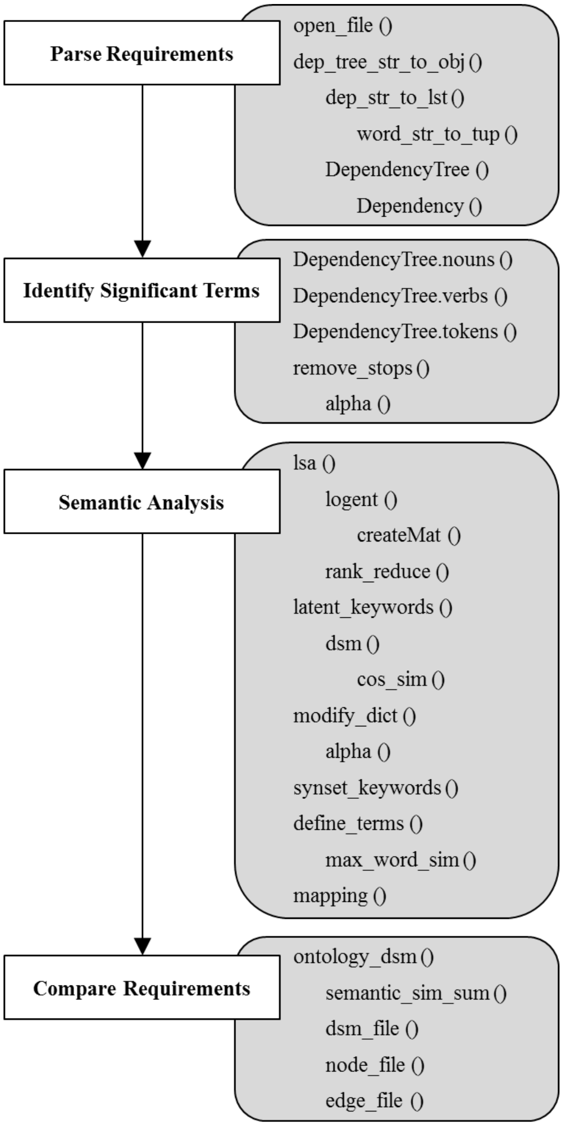


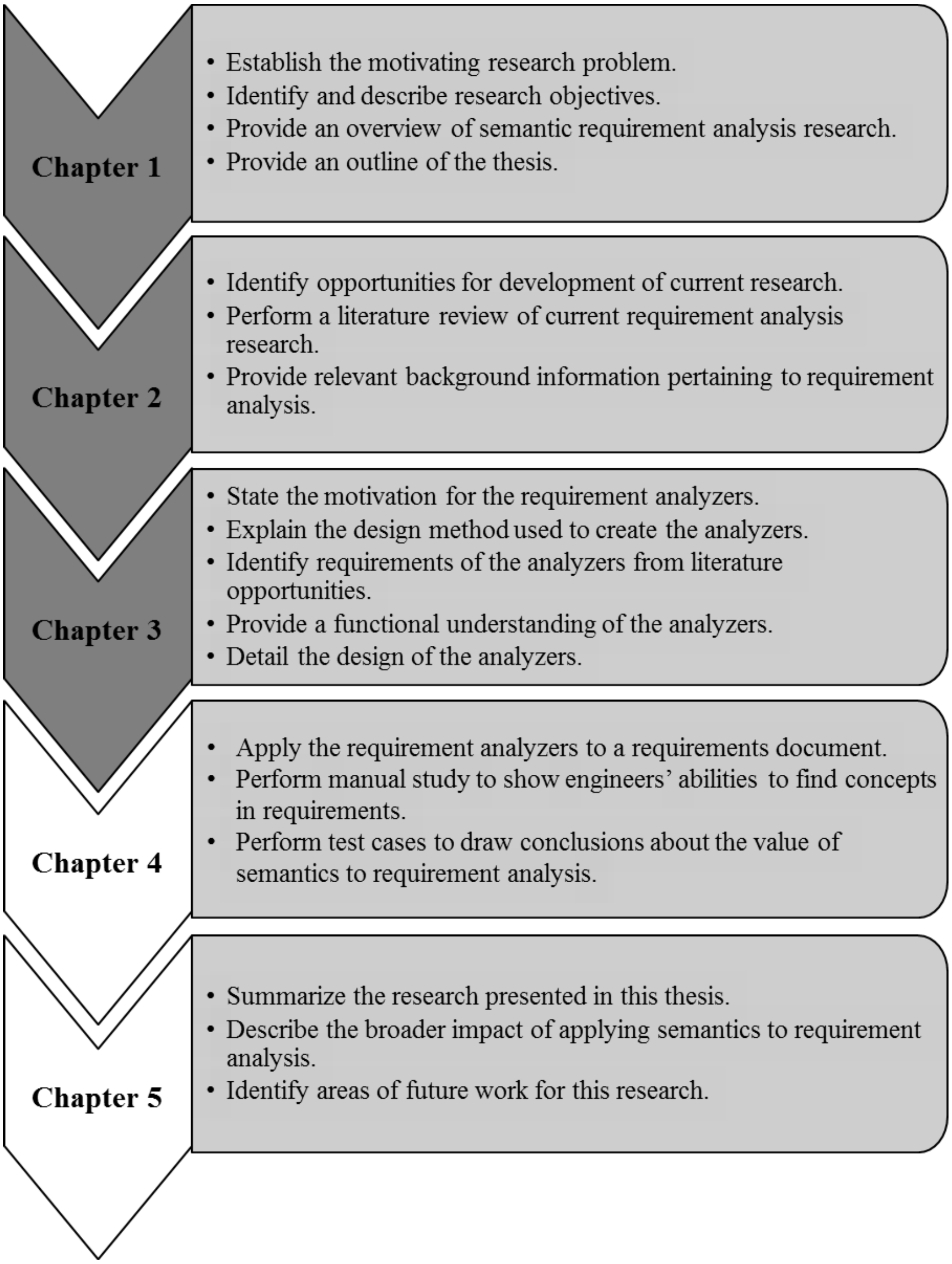
Figure 3-32: Embodied function model of the Semantic Analyzer showing functions, classes, and methods.

3.4 Chapter Conclusions

In this chapter, the design of each requirement analyzer was detailed. The four requirement analyzers designed are:

- Syntax Analyzer
- LSA Analyzer
- Component LSA Analyzer
- Semantic Analyzer

The Syntax Analyzer implements only parsing and string matching. This analyzer is representative of existing methods used for requirement analysis. The LSA Analyzer extends the Syntax Analyzer by using LSA to find latent relationships between the requirements. The Component LSA Analyzer performs similar to the LSA Analyzer except only on identified components in the text instead of the entire requirement statement. The Semantic Analyzer extends the Component LSA Analyzer by using a semantic ontology in unison with LSA. To create these analyzers, the identified opportunities from research are converted to analyzer requirements. Using the derived requirements, the analyzers are conceptualized via a functional model and subsequently detailed. Figure 3-33 shows the completed chapters (grey chevrons) and upcoming chapters (white chevrons) along with their respective deliverables.



Chapter 1

- Establish the motivating research problem.
- Identify and describe research objectives.
- Provide an overview of semantic requirement analysis research.
- Provide an outline of the thesis.

Chapter 2

- Identify opportunities for development of current research.
- Perform a literature review of current requirement analysis research.
- Provide relevant background information pertaining to requirement analysis.

Chapter 3

- State the motivation for the requirement analyzers.
- Explain the design method used to create the analyzers.
- Identify requirements of the analyzers from literature opportunities.
- Provide a functional understanding of the analyzers.
- Detail the design of the analyzers.

Chapter 4

- Apply the requirement analyzers to a requirements document.
- Perform manual study to show engineers' abilities to find concepts in requirements.
- Perform test cases to draw conclusions about the value of semantics to requirement analysis.

Chapter 5

- Summarize the research presented in this thesis.
- Describe the broader impact of applying semantics to requirement analysis.
- Identify areas of future work for this research.

Figure 3-33: Overview of thesis chapters showing chapters 1-3 completed (grey chevrons).

CHAPTER FOUR
TEST CASES AND INTERPRETATION OF REQUIRMENT ANALYSIS RESULTS

Chapter Objectives:

1. Apply the requirement analyzers to a requirements document.
2. Provide overview of requirements document evaluated by requirement analyzers.
3. Perform manual study to show engineers' abilities to find concepts in requirements.
4. Perform test cases to draw conclusions about the value of semantics to requirement analysis.

In this chapter, the semantic requirement analyzers designed in Chapter Three are applied to a BMW accelerator pedal module requirements document. The results are compared to manual findings in a series of test cases. These test cases are introduced in Table 4-1.

Table 4-1: List of test cases to validate research.

ID	Test Case	Description
1	Manual Comparison	Compare manually-obtained relationships.
2	Semantic Mapping	Compare semantic ontology mapping and LSA to manually identified semantics.
3	Manual to Syntax	Compare syntax and string matching algorithms to manual relationships.
4	Manual to Component LSA	Compare LSA performed on the components to manual relationships.
5	Manual to Semantics	Compare semantic ontology mapping and LSA to manual relationships.
6	Manual to LSA	Compare LSA performed on entire requirements to manual relationships.

The first test case relates manual findings to one another. The goal of this test case is to determine the level to which engineers conceptually agree on requirement relationships. To gather these findings, subjects are asked to relate requirements to one another individually. These results are then compared to one another and a manual collaborative study. The second test case assesses a requirement analyzer's ability to interpret requirements. Test cases 3-6 evaluate the ability of semantics to extend conceptual understanding of requirements beyond those of syntax and string matching algorithms that are often found in requirement analyses.

Before the test cases are introduced, an overview of the accelerator pedal module is provided. Also, the manual study performed in this research is detailed. This study relates the accelerator pedal module requirements to one another based on the expertise of engineers. The manual study provides the standard against which the conceptual models of the requirement analyzers are judged. Afterwards, the test cases are presented and the results of each are discussed.

4.1 BMW Accelerator Pedal Module Overview

A requirements document for a BMW accelerator pedal module is analyzed using the semantic requirement analyzers. The accelerator pedal module provides both the pedal and pedal mechanism that accelerate the vehicle. Figure 4-1 shows a model of the accelerator pedal module.

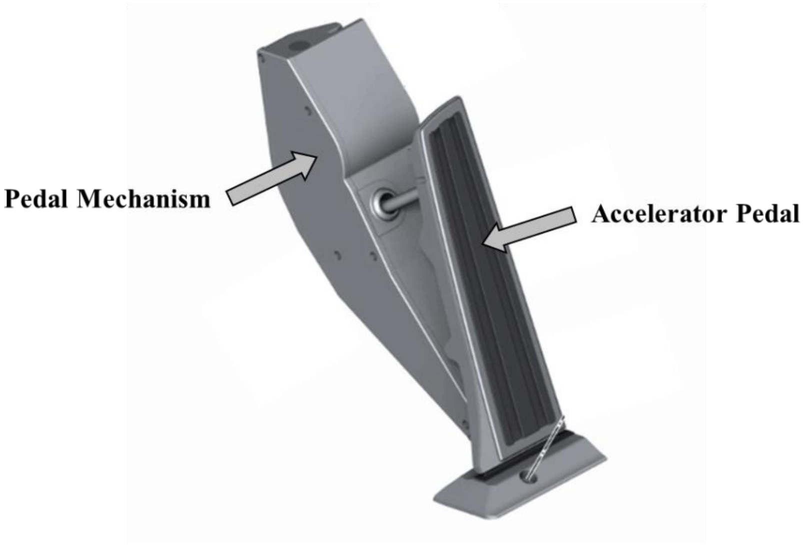
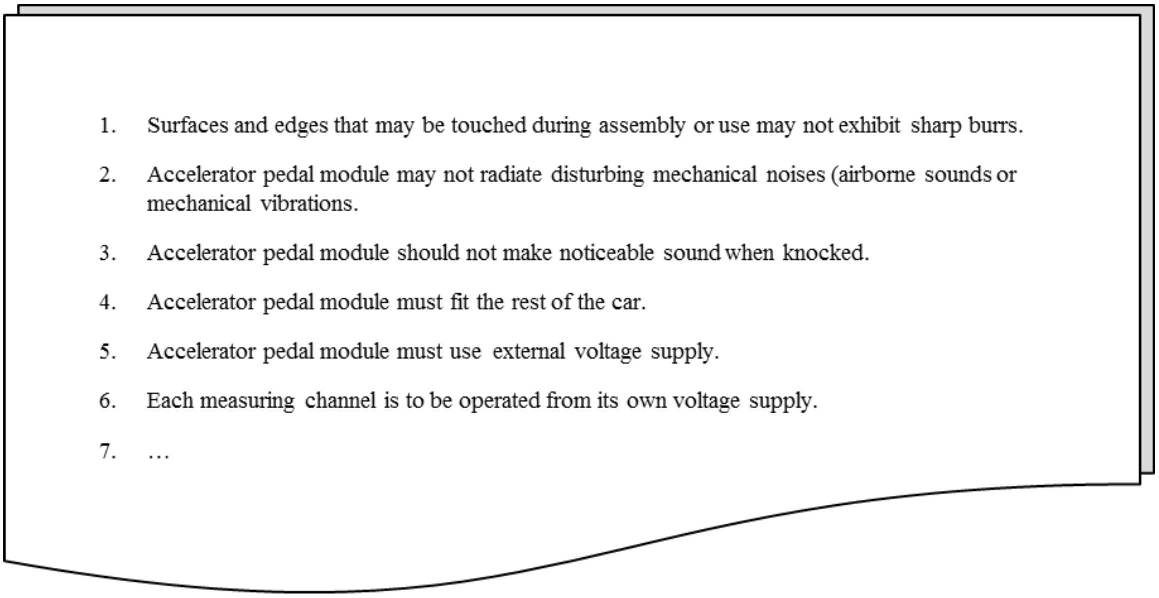


Figure 4-1: Solid model of the BMW accelerator pedal module [24].

While the pedal module has already been designed and produced, the requirements document used for analysis is an early design phase revision of the document. This document was chosen because it is an industry requirements document that is feasible for manual analysis. The requirements document consists of 24 natural language requirements. This requirements document is text-based and has no tables or figures. A sample of the formatted requirements document is shown in Figure 4-2. The full requirements text is provided in the Appendix B.



1. Surfaces and edges that may be touched during assembly or use may not exhibit sharp burrs.
2. Accelerator pedal module may not radiate disturbing mechanical noises (airborne sounds or mechanical vibrations.
3. Accelerator pedal module should not make noticeable sound when knocked.
4. Accelerator pedal module must fit the rest of the car.
5. Accelerator pedal module must use external voltage supply.
6. Each measuring channel is to be operated from its own voltage supply.
7. …

Figure 4-2: Image of the BMW requirements document showing a sample of the requirements and document formatting.

4.2 Manual Requirement Relation Study

Value in this research is based on whether or not the conceptual models found by the analyzers map to that of manually derived conceptual models. In other words, if the requirement analyzers do not derive the relationships between requirements that engineers expect, the requirement analyzers are not useful. The study involved three engineers. The background of each engineer is provided in Table 4-2.

Table 4-2: Education level of each engineer used in the manual study.

ID	Engineering Design	Requirements	Linguistics
Engineer 1	Undergraduate	Graduate	Graduate
Engineer 2	Graduate	Graduate	Undergraduate
Engineer 3	Graduate	Graduate	Undergraduate

The engineers were provided a brief overview of the accelerator pedal module and a problem statement:

Draw the relationships between requirements on the board provided.

The problem statement was designed to create the relationships between requirements that engineers would find beneficial to the design process for any reason they deemed fit. Then, the engineers collaboratively pairwise related the 24 accelerator pedal module requirements to one another on the whiteboard provided. The whiteboard had the 24 requirements around the edges of the board and a marker was used to draw lines between related requirements. A sample of these relationships on five requirements is shown in Figure 4-3.

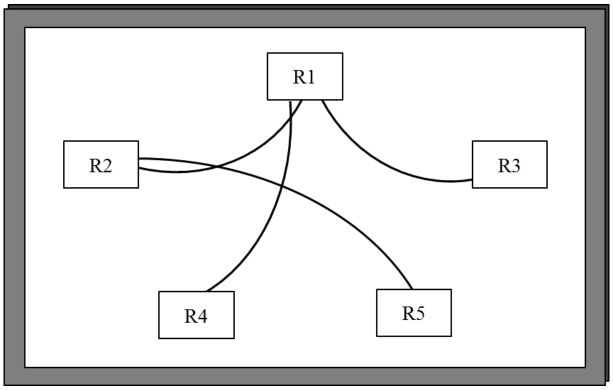
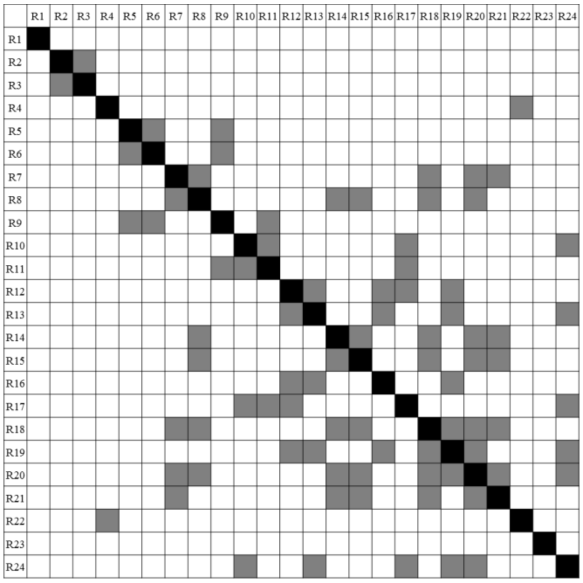


Figure 4-3: Sample of pairwise relating requirements on a whiteboard. Each box (R1-R5) contains the requirement statement text.

The relations where created on a binary scale:

- Line drawn between two requirements: Yes, the requirements are related.
- No line drawn between two requirements: No, the requirements are not related.

The results from the whiteboard were converted to a DSM showing the relations. The DSM is a conceptual model of the manually derived relationships between requirements.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■																							
R2		■	▨																					
R3		▨	■																					
R4				■																		▨		
R5					■	▨			▨															
R6					▨	■			▨															
R7							■	▨										▨		▨	▨			
R8							▨	■						▨	▨			▨		▨				
R9					▨	▨			■		▨													
R10										■	▨						▨							▨
R11									▨	▨	■						▨							
R12												■	▨			▨	▨		▨					
R13												▨	■			▨			▨					▨
R14								▨						■	▨			▨		▨	▨			
R15								▨						▨	■			▨		▨	▨			
R16												▨	▨			■			▨					
R17										▨	▨	▨					■							▨
R18							▨	▨						▨	▨			■	▨	▨	▨			
R19												▨	▨			▨		▨	■	▨				▨
R20							▨	▨						▨	▨			▨	▨	■	▨			▨
R21							▨							▨	▨			▨		▨	■			
R22				▨																		■		
R23																							■	
R24										▨			▨				▨		▨	▨				■

Figure 4-4: DSM of accelerator pedal module requirements showing relationships between requirements (grey boxes) found via manual study.

This final DSM is compared to the requirement analyzers in Test Case 1 and Test Cases 3 – 8 to evaluate the ability of the analyzers to form relationships between requirements.

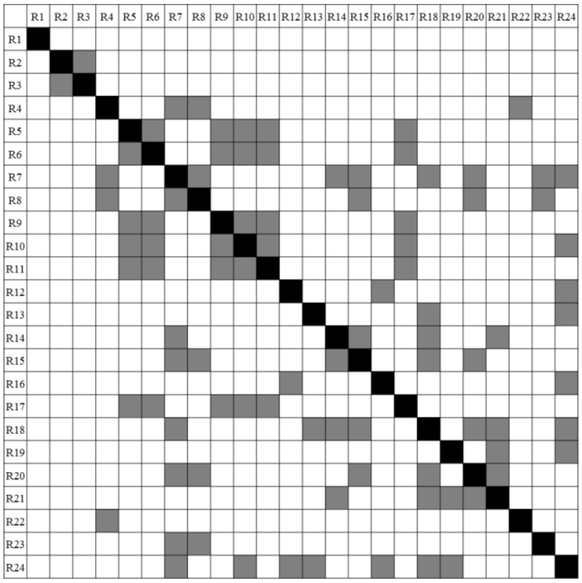
4.3 Test Case 1: Manual Comparison

In addition to the collaborative conceptual model created by the study in Section 4.2, three individual manual models were created. Using the same process described in Section 4.2, three engineers were asked to pairwise relate the 24 accelerator pedal module requirements to one another on a whiteboard. This study differs in that each individual engineer separately drew relationships, resulting in three individually obtained sets of results.

Table 4-3: Education level of engineers used in Test Case 1 to compare to results found in manual study.

ID	Engineering Design	Requirements	Linguistics
Engineer 1	Undergraduate	Graduate	Graduate
Engineer 2	Graduate	Graduate	Undergraduate
Engineer 3	Graduate	Graduate	Undergraduate

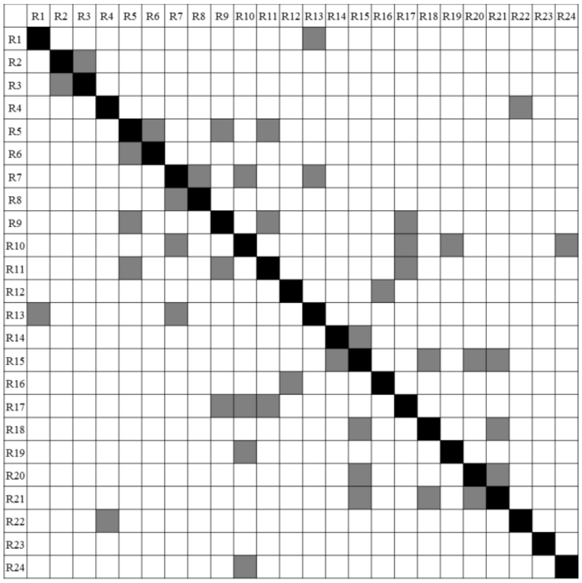
Each result from the whiteboard was converted to a DSM for a total of three DSMs—one per engineer. The DSM for Engineer 1 is shown in Figure 4-5.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■																							
R2		■	▨																					
R3		▨	■																					
R4				■			▨	▨														▨		
R5					■	▨			▨	▨	▨						▨							
R6					▨	■			▨	▨	▨						▨							
R7				▨			■	▨						▨	▨			▨		▨			▨	▨
R8				▨			▨	■							▨					▨			▨	
R9					▨	▨			■	▨	▨						▨							
R10					▨	▨			▨	■	▨						▨							▨
R11					▨	▨			▨	▨	■						▨							
R12												■				▨								▨
R13													■					▨						▨
R14							▨							■	▨			▨			▨			
R15							▨	▨						▨	■			▨		▨				
R16												▨				■								▨
R17					▨	▨			▨	▨	▨						■							
R18							▨						▨	▨	▨			■		▨	▨			▨
R19																			■		▨			▨
R20							▨	▨							▨			▨		■	▨			
R21														▨				▨	▨	▨	■			
R22				▨																		■		
R23							▨	▨															■	
R24							▨			▨		▨	▨			▨		▨	▨					■

Figure 4-5: DSM of accelerator pedal module requirements showing relationships between requirements (grey boxes) from Engineer 1.

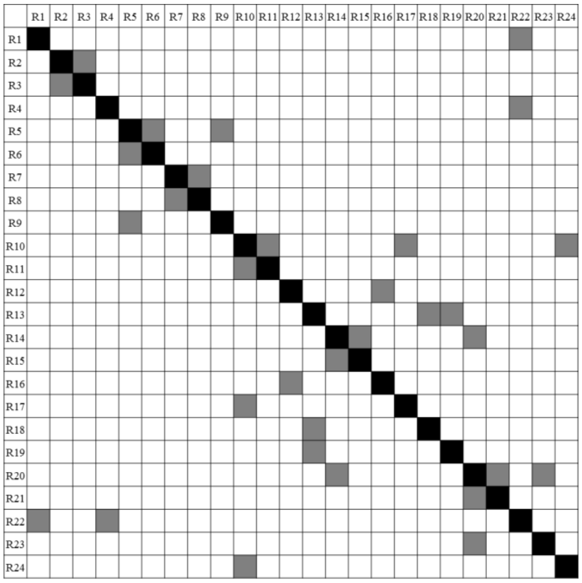
The DSM for Engineer 2 is shown in Figure 4-6.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■												▨											
R2		■	▨																					
R3		▨	■																					
R4				■																		▨		
R5					■	▨			▨		▨													
R6					▨	■																		
R7							■	▨		▨			▨											
R8							▨	■																
R9					▨				■		▨						▨							
R10							▨			■							▨		▨					▨
R11					▨				▨		■						▨							
R12												■				▨								
R13	▨						▨						■											
R14														■	▨									
R15														▨	■			▨		▨	▨			
R16												▨				■								
R17									▨	▨	▨						■							
R18															▨			■			▨			
R19										▨									■					
R20															▨					■	▨			
R21															▨			▨		▨	■			
R22				▨																		■		
R23																							■	
R24										▨														■

Figure 4-6: DSM of accelerator pedal module requirements showing relationships between requirements (grey boxes) from Engineer 2.

The DSM for Engineer 3 is shown in Figure 4-7.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■																					▨		
R2		■	▨																					
R3		▨	■																					
R4				■																		▨		
R5					■	▨			▨															
R6					▨	■																		
R7							■	▨																
R8							▨	■																
R9					▨				■															
R10										■	▨						▨							▨
R11										▨	■													
R12												■				▨								
R13													■					▨	▨					
R14														■	▨					▨				
R15														▨	■									
R16												▨				■								
R17										▨							■							
R18													▨					■						
R19													▨						■					
R20														▨						■	▨		▨	
R21																				▨	■			
R22	▨			▨																		■		
R23																				▨			■	
R24										▨														■

Figure 4-7: DSM of accelerator pedal module requirements showing relationships between requirements (grey boxes) from Engineer 3.

Each of the individual DSMs are compared to one another and the DSM from the manual collaborative study. Cohen's Kappa is used to relate and compare these DSM's to one another. The kappa value between any two DSMs is a measure of the agreement between them. To use Cohen's Kappa, the relationships in each DSM are extracted and placed into $n \times 1$ vectors respectively, where n is the number of relationships in the DSM. Both vectors are then related to one another. The resulting Kappa values are shown in Table 4-4.

Table 4-4: Cohen's Kappa values comparing the individual manual results to the results of the manual collaborative study.

Comparison	Kappa Value
E1 – E2	0.57
E1 – E3	0.49
E2 – E3	0.68
E1 – Collaborative	0.61
E2 – Collaborative	0.58
E3 – Collaborative	0.56

Kappa values are considered relative to their application. Ideally, the kappa value would be 1.00, indicating complete agreement between two conceptual models. However, these kappa values show that the highest kappa value relationship between any two engineers or group of engineers is 0.68. Therefore, to expect a computational tool to exceed a kappa value of 0.68 when compared to manual findings may be unreasonable. In addition, the highest value obtained by comparing the individual results to the manual study is 0.61. As are all the individual manual results in this test case, all requirement analyzers are compared to the manual study. This test case provides a reasonable frame of reference on which to judge the requirement analyzers.

4.4 Test Case 2: Semantic Mapping

The Semantic Analyzer applies semantic meaning to the requirement elements by mapping them to a semantic ontology. This semantic interpretation method must be validated. The validation seeks to prove that the supplied semantics are accurate with manually obtained results. Using the semantics supplied to the requirements, the similarity metric between the requirements is found through a semantic comparison of the requirements.

Accuracy of the semantic interpretation is measured against the knowledge of 6 raters. The background of each rater is provided in Table 4-5.

Table 4-5: Education level of raters used in the semantic interpretation study.

ID	Engineering Design	Requirements	Linguistics
Rater 1	Undergraduate	Graduate	Graduate
Rater 2	Graduate	Graduate	Undergraduate
Rater 3	Graduate	Graduate	Undergraduate
Rater 4	Graduate	Graduate	Graduate
Rater 5	Graduate	Graduate	Undergraduate
Rater 6	Graduate	Graduate	Undergraduate

Each rater was provided a list of the significant terms mapped to the semantic ontology. Each significant term was paired with the definition provided by the semantic ontology. Using the definition of the term, each rater individually asserted whether the term was correctly or incorrectly mapped to the semantic ontology. A sample of this process is shown in Table 4-6.

Table 4-6: Sample of semantic validation process showing method for raters assessing terms.

Requirement	Term	Definition	Rater 1	Rater 2	…
1	burrs	seed vessel having hooks or prickles	Incorrect	Correct	…
2	pedal	a lever that is operated with the foot	Correct	Correct	…
2	noises	electrical or acoustic activity that can disturb communication	Correct	Correct	…
3	sound	the subjective sensation of hearing something	Correct	Correct	…
…	…	…	…	…	

The raters were provided information about the pedal module and provided the original requirements document. The original requirements document was provided so that the raters could identify the context in which the term was used. The results of the study are shown in Figure 4-8.

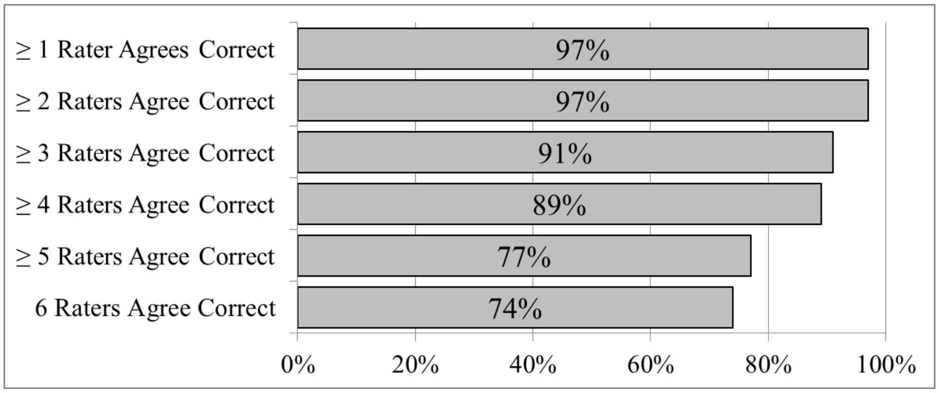


Figure 4-8: Percent of the terms mapped correctly based on the number of raters needed to agree to consider a term correctly mapped.

Figure 4-8 shows accuracy based on the constraint of number of raters that must agree. For instance, the third bar in Figure 4-8 shows 91% mapping accuracy when at least three raters agree that the term is mapped correctly. Figure 4-8 also shows what percent of the terms were incorrect based on the constraint of number of raters. The notable outcomes of this validation are presented in Table 4-7.

Table 4-7: Results of semantic validation.

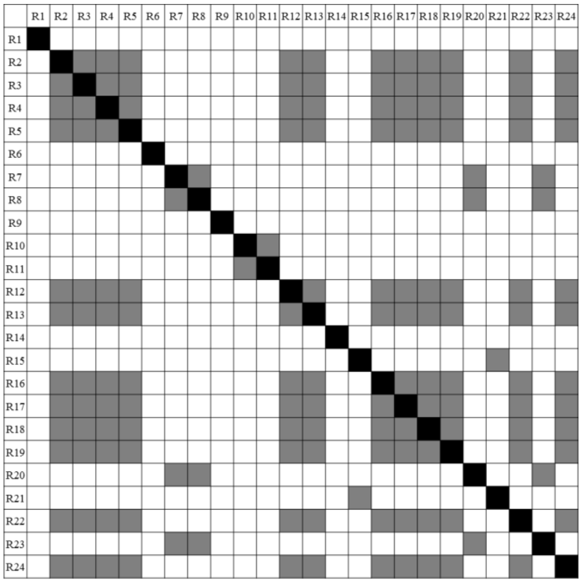
Statistic	Value
Number of Raters	6
Number of Terms	35
6 Raters Agree Correct	74% (26/35)
≥4 Raters Agree Correct (majority)	89% (31/35)
6 Raters Agree Incorrect	3% (1/35)

The results of the semantic validation show that the worst case accuracy of the semantic mapping is 74%. This percent means that every rater agreed that the term was correctly mapped to the semantic ontology for 74% of the terms. Over half the raters agreed that a term was correctly mapped for 89% of the terms. In only one instance (3%) did every rater believed that the term was incorrectly mapped to the ontology. With this validation, the effective accuracy of the semantic mapping can be considered between 74-89%.

4.5 Test Case 3: Manual to Syntax Comparison

As discussed in Section 2.5, many existing methods for gaining conceptual understanding of requirements are based upon syntax and string matching methods alone. This test case serves a representation of the capabilities of syntax and string matching methods to form relationships between requirements. In particular, the Syntax Analyzer results are compared the results obtained from the manual study in Section 4.2.

The results of both the Syntax Analyzer and the manual study are binary DSMs that state either the requirements are related or are not related to one another. The manual study DSM is provided in Section 4.2. The Syntax Analyzer DSM is provided in Figure 4-9.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■																							
R2		■	▨	▨	▨							▨	▨			▨	▨	▨	▨			▨		▨
R3		▨	■	▨	▨							▨	▨			▨	▨	▨	▨			▨		▨
R4		▨	▨	■	▨							▨	▨			▨	▨	▨	▨			▨		▨
R5		▨	▨	▨	■							▨	▨			▨	▨	▨	▨			▨		▨
R6						■																		
R7							■	▨												▨			▨	
R8							▨	■												▨			▨	
R9									■															
R10										■	▨													
R11										▨	■													
R12		▨	▨	▨	▨							■	▨			▨	▨	▨	▨			▨		▨
R13		▨	▨	▨	▨							▨	■			▨	▨	▨	▨			▨		▨
R14														■										
R15															■						▨			
R16		▨	▨	▨	▨							▨	▨			■	▨	▨	▨			▨		▨
R17		▨	▨	▨	▨							▨	▨			▨	■	▨	▨			▨		▨
R18		▨	▨	▨	▨							▨	▨			▨	▨	■	▨			▨		▨
R19		▨	▨	▨	▨							▨	▨			▨	▨	▨	■			▨		▨
R20							▨	▨												■			▨	
R21															▨						■			
R22		▨	▨	▨	▨							▨	▨			▨	▨	▨	▨			■		▨
R23							▨	▨												▨			■	
R24		▨	▨	▨	▨							▨	▨			▨	▨	▨	▨			▨		■

Figure 4-9: DSM of accelerator pedal module requirements showing relationships between requirements (grey boxes) found via the Syntax Analyzer.

Cohen's Kappa is used to relate and compare these DSM's to one another. The resulting kappa value is:

$$K = 0.26$$

This kappa value represents a fair agreement between the syntactic conceptual model and the manual model. Analysis of the other requirement analyzers provides further insight into the meaning of this kappa value.

4.6 Test Case 4: Manual to Component LSA Comparison

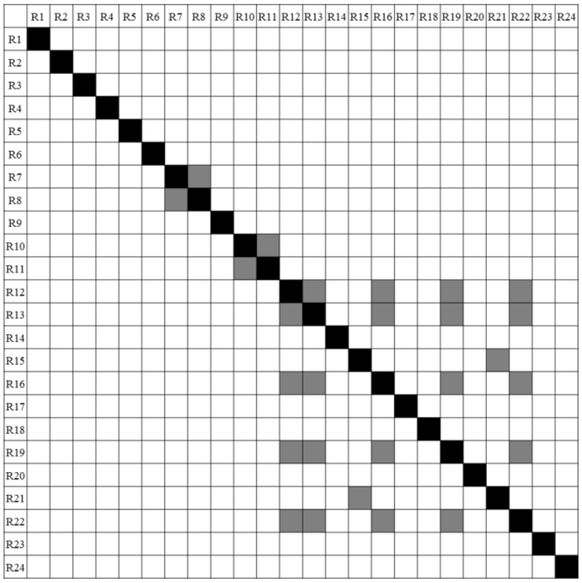
This test case compares the performance of the Component LSA Analyzer to the manually obtained results from Section 4.2. Similar to Test Case 3 described in Section 4.5, the resulting DSMs from both analyses are compared using Cohen's Kappa. Unlike the Syntax Analyzer, the Component LSA Analyzer is not on a binary scale. The values of the component LSA DSM vary from -1.00 to 1.00, where less than 0.00 indicates no relationship and 1.00 indicates an identical relationship. The results of the Component LSA Analyzer must be on a binary scale in order to use Cohen's Kappa.

To enable the comparison, a set of semantic threshold values are implemented creating new DSMs that are on binary scales. These semantic threshold values are the sensitivity of the tool to semantic meaning. For example, at a high semantic threshold (low sensitivity) the terms must be nearly synonymous in order to be able to assert that any two requirements are related. Below the threshold value, the new DSM cells are assigned a value of zero, indicating no relationship between the requirements. Above or equal to the threshold value, new DSM cells are assigned a value of one, indicating a relationship between the requirements. 10 threshold values are used for this test case. Using the 10 new DSMs obtained from assigning threshold values to the original DSM, each DSM can be compared to the manually obtained DSM from Section 4.2. The Cohen's Kappa values are obtained using the method described in Section 4.5. The resulting Kappa values at each cutoff value are shown in Table 4-8.

Table 4-8: Cohen's Kappa values comparing the Component LSA Analyzer to manually obtained results.

Cutoff Value	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Kappa Value	0.25	0.25	0.26	0.26	0.28	0.33	0.33	0.36	0.43	0.48

Table 4-8 shows that the highest agreement between the Component LSA Analyzer model and the manually obtained model is at a threshold value of 1.0. Figure 4-10 shows the DSM obtained at this threshold.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■																							
R2		■																						
R3			■																					
R4				■																				
R5					■																			
R6						■																		
R7							■	▨																
R8							▨	■																
R9									■															
R10										■	▨													
R11										▨	■													
R12												■	▨			▨			▨			▨		
R13												▨	■			▨			▨			▨		
R14														■										
R15															■						▨			
R16												▨	▨			■			▨			▨		
R17																	■							
R18																		■						
R19												▨	▨			▨			■			▨		
R20																				■				
R21															▨						■			
R22												▨	▨			▨			▨			■		
R23																							■	
R24																								■

Figure 4-10: DSM of accelerator pedal module requirements showing relationships between requirements (grey boxes) found via the Component LSA Analyzer at a semantic threshold value of 1.0.

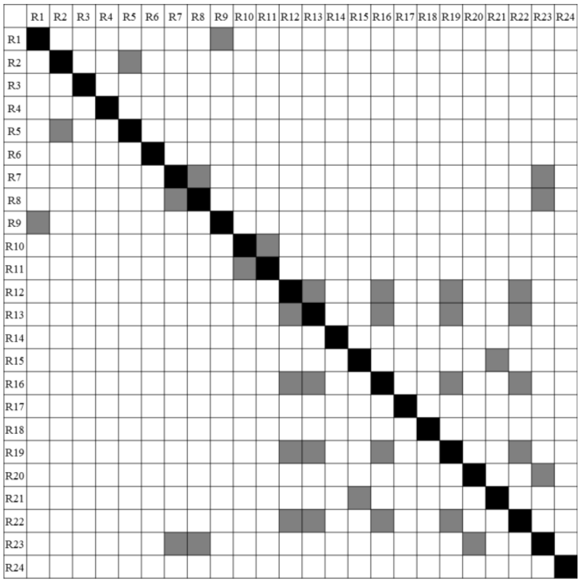
4.7 Test Case 5: Manual to Semantics Comparison

This test case compares the performance of the Semantic Analyzer to the manually obtained results from Section 4.2. The method for comparison is identical to the method used for Test Case 4 described in Section 4.6. The DSM values from the Semantic Analyzer vary from 0.00 to 1.00, where 0.00 indicates no relationship and 1.00 indicates an identical or synonymous relationship. The semantic threshold values applied to the Semantic DSM are the same as those used in Test Case 4. Table 4-9 shows the results of the comparison between the Semantic Analyzer and the manually obtained values.

Table 4-9: Cohen's Kappa values comparing the Semantic Analyzer to the manually obtained results.

Cutoff Value	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Kappa Value	0.30	0.30	0.30	0.30	0.30	0.43	0.29	0.30	0.30	0.30

Table 4-9 shows that the highest agreement between the Semantic Analyzer model and the manually obtained model is at a threshold value of 0.6. Figure 4-11 shows the DSM obtained at this threshold.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■								▨															
R2		■			▨																			
R3			■																					
R4				■																				
R5		▨			■																			
R6						■																		
R7							■	▨															▨	
R8							▨	■															▨	
R9	▨								■															
R10										■	▨													
R11										▨	■													
R12												■	▨			▨			▨			▨		
R13												▨	■			▨			▨			▨		
R14														■										
R15															■						▨			
R16												▨	▨			■			▨			▨		
R17																	■							
R18																		■						
R19												▨	▨			▨			■			▨		
R20																				■			▨	
R21															▨						■			
R22												▨	▨			▨			▨			■		
R23							▨	▨												▨			■	
R24																								■

Figure 4-11: DSM of accelerator pedal module requirements showing relationships between requirements (grey boxes) found via the Semantic Analyzer at a semantic threshold value of 0.6.

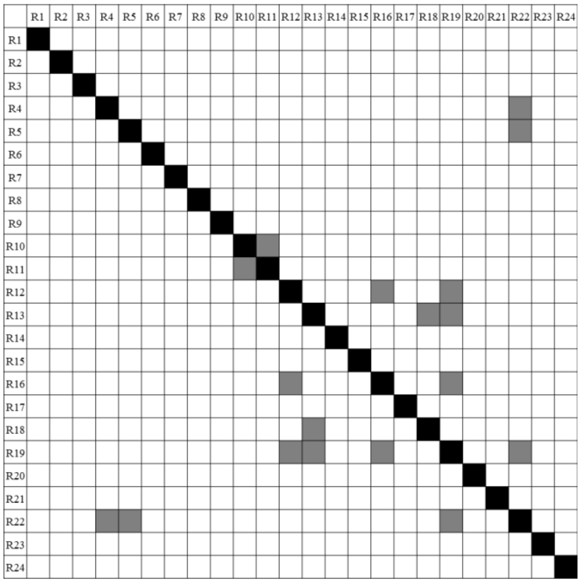
4.8 Test Case 6: Manual to LSA Comparison

This test case compares the performance of the LSA Analyzer to the manually obtained results from Section 4.2. Similar to Test Case 4 and 5, the comparison method uses Cohen’s Kappa to compare the converted LSA DSMs to the manually obtained results. The values of the LSA DSM vary from -1.00 to 1.00, where less than 0.00 indicates no relationship and 1.00 indicates an identical relationship. The kappa values comparing the LSA Analyzer to the manual results are shown in Table 4-10.

Table 4-10: Cohen’s Kappa values comparing the LSA Analyzer to the manually obtained results.

Cutoff Value	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Kappa Value	0.16	0.24	0.27	0.33	0.36	0.35	0.41	0.43	0.39	0.35

Table 4-10 shows that the highest agreement between the LSA Analyzer model and the manually obtained model is at a threshold value of 0.8. Figure 4-12 shows the DSM obtained at this threshold.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■																							
R2		■																						
R3			■																					
R4				■																		▨		
R5					■																	▨		
R6						■																		
R7							■																	
R8								■																
R9									■															
R10										■	▨													
R11										▨	■													
R12												■				▨			▨					
R13													■					▨	▨					
R14														■										
R15															■									
R16												▨				■			▨					
R17																	■							
R18													▨					■						
R19												▨	▨			▨			■			▨		
R20																				■				
R21																					■			
R22				▨	▨														▨			■		
R23																							■	
R24																								■

Figure 4-12: DSM of accelerator pedal module requirements showing relationships between requirements (grey boxes) found via the LSA Analyzer at a semantic threshold value of 0.8.

4.9 Requirement Analyzer Comparison

From the collected data in the test cases, the requirement analyzers can be compared to one another. In the test cases, all requirement analyzers were compared to the collaborative manually obtained results. Figure 4-13 shows all of the Cohen's Kappa values collected from the test cases. The manual line represents the highest agreement (0.61) between an individual engineer and the collaborative manually obtained results. The syntax line represents the agreement between the Syntax Analyzer and the manual collaborative results. These are used as a frame of reference for the requirement analyzers that used semantics.

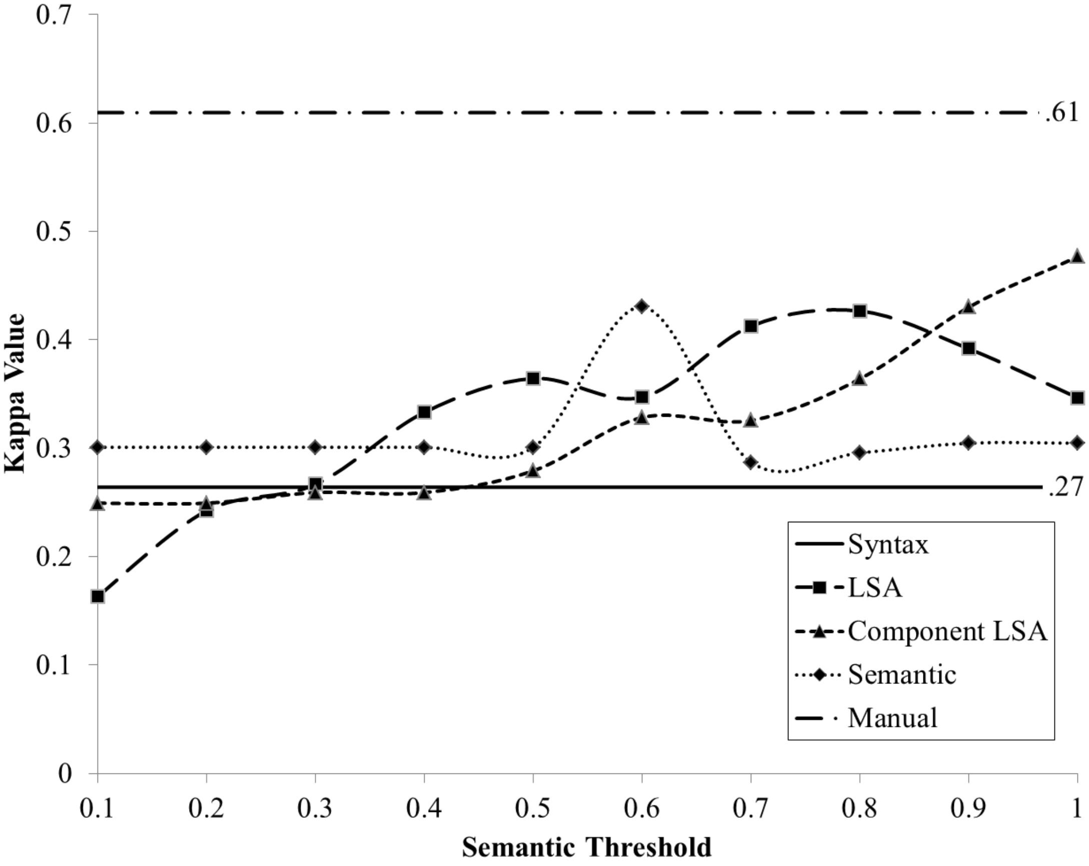


Figure 4-13: Cohen's Kappa values of each analyzer by semantic threshold value.

Higher Cohen's Kappa values show that the conceptual model for that analyzer better represents the manual model. Figure 4-13 shows that no analysis performs better than the manually-formed relationships. These results suggest that the semantics added by the requirement analyzers cannot capture all of the context and concepts that an engineer uses to relate requirements. The analyzers that use LSA to relate requirements (LSA Analyzer and Component LSA Analyzer) may be limited by the context that is used to relate requirements. The context that LSA uses to relate requirements is limited to how terms within the document are related to one another. Expanding the scope of the LSA to include some corpus data could improve the results. Also, Semantic Analyzer is limited by the ability of the semantic ontology to relate requirements. While the semantic mapping is at least 74% accurate, the relations within the ontology are based upon expert knowledge in linguistics. If the relations within the ontology are not ideal, this factor could add to the discrepancy between manual findings and computational results. As semantic analysis tools improve, the abilities of semantic requirement analyzers should also improve.

While the results show that the semantic analysis methods do not fully agree with manually-found concepts, the data does show that all analyzers employing a semantic interpretation perform better than syntactic and string matching methods alone. This is evident in Figure 4-13 as all semantic requirement analyzers outperform the Syntax Analyzer at a threshold value above 0.5. At low thresholds (high sensitivity), requirements that are only slightly related are related to one another. At this high sensitivity, syntactic methods alone can outperform semantic methods. Figure 4-13 shows that at a threshold of 0.1, the syntax analyzer outperforms both the LSA and Component LSA Analyzer. However, when semantics are added to a requirement analysis, this data shows that there is an increased ability to differentiate between requirements. At a threshold of 0.9, every analyzer that employs semantics outperforms the Syntax Analyzer. This finding supports the hypothesis that semantics increase the computational abilities of forming requirement relationships beyond string matching and syntax alone.

The Semantic Analyzer performs better than the Syntax Analyzer at every threshold. This finding shows that employing semantics in a requirement analysis can be beneficial no matter the sensitivity to semantics. Even when mildly related requirements are found via semantics, they have the capability to outperform syntactic methods alone. The data also shows that the Semantic Analyzer has a specific threshold range where it has optimum performance—where the agreement between the Semantic Analyzer and manual analysis is highest. Figure 4-13 shows that the Semantic Analyzer has a substantially higher agreement with manually obtained concepts at a threshold of 0.6. If a requirement analysis could be trained to find these areas of optimum performance, requirement analysis methods could provide much more valuable results to designers.

The Component LSA Analyzer improves as the threshold increases. It outperforms the Syntactic Analyzer at a threshold of 0.5. This shows that if only the requirements that are highly semantically related to one another are observed, the relationships between requirements are more significant. This finding is also generally true for the LSA Analyzer. The LSA Analyzer outperforms the Syntactic Analyzer at a threshold of 0.3. From these results, one can conclude that a latent analysis to find relationships between requirements is more beneficial when only requirements that are very similar are considered related to one another. The Semantic Analysis did not use LSA to relate requirements, but to relate terms. The semantic ontology was used to relate requirements. This may be the reason that the trend in Figure 4-13 for the Semantic Analyzer is different than those using LSA to relate requirements.

This research seeks to build upon other research in the requirement analysis field. In particular, the research in the area of requirement change propagation prediction uses a DSM model of relationships between requirements [12, 13]. This method employs syntactic and string matching methods to automatically find relationships between requirements. The research of this thesis shows that the relationships found can be improved using semantics. Also, selection of keywords is performed manually in current research. This research has shown that LSA is a viable option for finding those keywords. While, adding semantics to requirement analysis has not matched the conceptual understanding of engineers, it has shown a proof of concept and promise for further research. The subsequent sections detail the reasoning for the discrepancies between the requirement analyzers and the manual analysis methods.

4.9.1 Manual to Syntax Analyzer Results Comparison

From Figure 4-13, it is shown that manual methods have an agreement of 0.61, while the Syntactic Analyzer agrees with the manual analysis with a kappa value of 0.27. The manual agreement of 0.61 represents the highest agreement between an individual engineer from Test Case 1 and the manual collaborative study. The kappa values show that the Syntax Analyzer results do not agree with the manual collaborative study as well as an individual engineer agrees. This discrepancy can be attributed to the inability of string comparison and syntactic methods alone to capture complex relationships between requirements.

The Syntax Analyzer finds the subject and direct object of a requirement and pairwise compares them to the same elements in the other requirement statements. Any string match draws a relationship between the requirements. For instance, requirement 7 states:

> *Pedal angle generated by the actuating force must be limited.*

Requirement 8 states:

Pedal angle should be in the range of 16 degrees (+/- 0.5 degrees).

In this instance, the subject *angle* is the same in both requirements and the requirements are related by the Syntax Analyzer. The manual collaborative study results and all of the individual engineers agree that requirements 7 and 8 are related.

However, sometimes string matching relates two requirements that the engineers study did not find. For example, requirement 3 states:

Accelerator pedal module should not make noticeable sound when knocked.

Requirement 5 states:

Accelerator pedal module must use external voltage supply.

The Syntax Analyzer relates these two requirements because the subjects are string matched. However, this disagrees with the results of the collaborative study.

The Syntax Analyzer can also miss relationships due to only using explicit syntactic information. For instance, requirement 14 states:

Slope response on the pedal (i.e. spring stiffness) must be within 0.8 N per degrees and 1.1 N per degrees.

Requirement 21 states:

The ascending force should be 30 percent (+6 N / -3 N) of the descending force.

In these requirements, neither the subjects nor direct objects are sting matches. However, the manual analysis related these requirements because of the knowledge of forces. This example and the previous example are attributed to the ability of engineers to use implicit knowledge in addition to the explicit syntactic information in a requirement statement.

4.9.2 Manual to LSA and Component LSA Results Comparison

At a semantic threshold of 0.6 both the LSA Analyzer and Component LSA Analyzer have higher kappa values than the Syntax Analyzer. The LSA Analyzer agrees with the collaborative study with a kappa value of 0.35, and the Component LSA Analyzer agrees with the collaborative study with a kappa value of 0.33. As previously stated, the Syntactic Analyzer has an agreement of 0.26. The increase in agreement in analyzer using LSA is due to the implicit information added by LSA. For instance requirement 9 states:

> *The sensor must be redundant with respect to output voltage.*

Requirement 11 states:

> *The requirements refer to an electrical wiring output.*

The Syntax Analyzer did not relate these two requirements that the collaborative study related because neither the subject nor direct object string matched. However, LSA looks at all terms in the requirement and weights them based on a normalization scheme. Using this information, the LSA Analyzer found the relationship between these two requirements.

Using LSA also improved the agreement to the manual analysis by not identifying incorrect relationships that the Syntax Analyzer identified. For instance, requirement 3 states:

> *Accelerator pedal module should not make noticeable sound when knocked.*

Requirement 17 states:

> *Accelerator pedal module must not send a signal to the electronic control unit against the driver's wishes.*

Neither the LSA nor Component LSA Analyzers identified the relationship between these requirements. This result agrees with the manual analysis. However, the Syntax Analyzer identified this relationship because the requirements' subjects string match.

4.9.3 Manual to Semantic Analyzer Comparison

At the 0.6 semantic threshold, the Semantic Analyzer had the highest agreement to the manual analysis of all the requirement analyzers. The kappa value between the Semantic Analyzer and the collaborative study is 0.43. The Semantic Analyzer maps the subject and direct object in a requirement statement to a semantic ontology. The ontology path length is then used to derive a relationship value between the requirements. As with LSA, the Semantic Analyzer minimizes the number of identified relationships that disagree with the collaborative study. For example, requirement 4 states:

> *Accelerator pedal module must fit the rest of the car.*

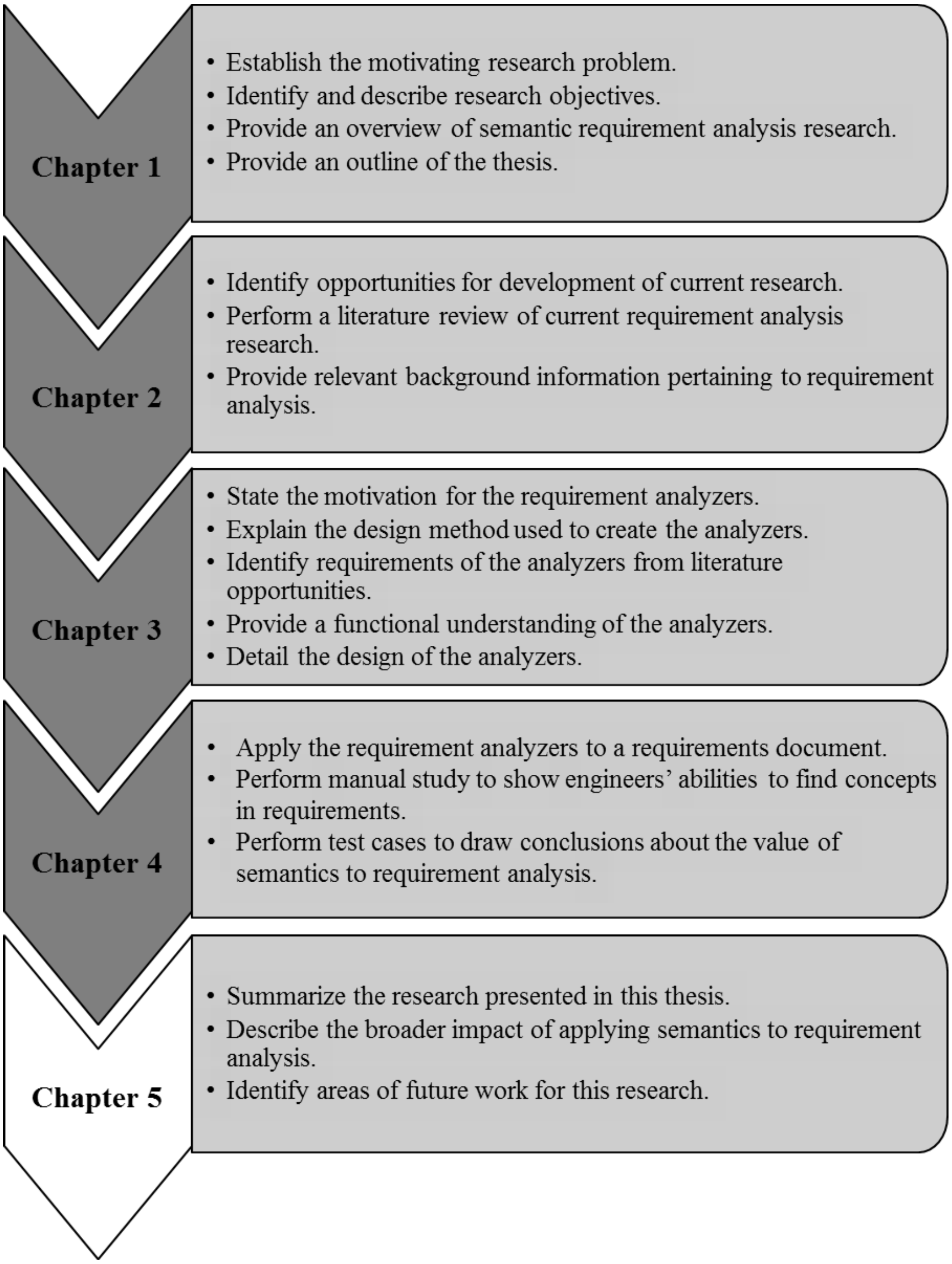
Requirement 12 states:

> *If pedal breaks, the pedal must not be operational.*

The Syntax Analyzer asserted the relationship between these two requirements where the collaborative results did not. Further, the Semantic Analyzer has shown greater improvement over the LSA and Component LSA Analyzers at the 0.6 threshold as both of these analyzers asserted a relationship between requirements 4 and 12. The minimization of false positives shows the most increase in agreement between the manual analyses and requirement analyzers.

4.10 Chapter Conclusions

This chapter details a set of test cases to show the findings of the three requirement analyzers employing semantics. The BMW accelerator pedal module requirements were evaluated by the analyzers and the results were compared. Test Case 2 shows that the semantic mapping within the Semantic Analyzer has at least 74% accuracy. Test Cases 3-6 show that the analyzers that implement semantics agree more with manually found concepts than the Syntax Analyzer. Further investigation shows that the semantic requirement analyzers agree more with the manual results because of the ability to minimize the number of false positive relationships. A semantic analysis increases the computational ability to filter requirement relationships because of the increase in implicit knowledge. Further, the LSA Analyzer was able to discover relationships found by the manual analysis that the Syntax Analyzer did not. This shows that semantics can find implicit relationships not able to be captured by syntax and string matching alone. Figure 4-14 shows the completed chapters (grey chevrons) and upcoming chapters (white chevrons) along with their respective deliverables.



Chapter 1

- Establish the motivating research problem.
- Identify and describe research objectives.
- Provide an overview of semantic requirement analysis research.
- Provide an outline of the thesis.

Chapter 2

- Identify opportunities for development of current research.
- Perform a literature review of current requirement analysis research.
- Provide relevant background information pertaining to requirement analysis.

Chapter 3

- State the motivation for the requirement analyzers.
- Explain the design method used to create the analyzers.
- Identify requirements of the analyzers from literature opportunities.
- Provide a functional understanding of the analyzers.
- Detail the design of the analyzers.

Chapter 4

- Apply the requirement analyzers to a requirements document.
- Perform manual study to show engineers' abilities to find concepts in requirements.
- Perform test cases to draw conclusions about the value of semantics to requirement analysis.

Chapter 5

- Summarize the research presented in this thesis.
- Describe the broader impact of applying semantics to requirement analysis.
- Identify areas of future work for this research.

Figure 4-14: Overview of thesis chapters showing chapters 1-4 completed (grey chevrons).

CHAPTER FIVE
CONCLUSIONS AND FUTURE WORK

Chapter Objectives:

1. Summarize the research presented in this thesis.
2. Describe the broader impact of applying semantics to requirement analysis.
3. Identify areas of future work for this research.

This thesis presents the development and application of semantics in requirement analysis. Current research has identified opportunities to implement semantics in requirement analyzers to extend the existing requirement analysis methods. The contributions of these analyzers are organized into three research objectives. This chapter identifies how these objectives are met by this research. Through completion of the research objectives the contributions of the analyzers to requirement analysis research is realized. Further, the broader impact of a semantic requirement analysis is discussed and possible ways to extend the analyzers and this research are analyzed.

5.1 Fulfillment of Research Objectives

5.1.1 RO 1: Supplementing Requirements with Semantics

Section 3.2.5.2 demonstrates that a computational method for applying semantic meaning has been realized. The Semantic Analyzer achieves this by mapping requirement terms to a semantic ontology. Significant terms are first identified by the analyzer. These significant terms represent terms that provide the requirement statement with meaning. Using LSA to identify the context of a requirements document, the significant terms in a requirement are mapped to a semantic ontology [14]. In this way, semantic meaning is supplied to each requirement. For instance, requirement 2 states:

> *Accelerator pedal module may not radiate disturbing mechanical noises (airborne sounds or mechanical vibrations).*

The Semantic Analyzer identifies *pedal* and *noises* as the significant terms in the requirement statement. These terms are mapped to the semantic ontology. The semantic ontology provides a definition of each mapped term. The definitions for *pedal* states:

> *A lever that is operated with the foot.*

All six raters identified that this mapping was correct based upon the definition provided. This meaning of pedal was mapped to automatically using insight into the context of the requirement terms from LSA. This helped select the definition above as opposed to another meaning of *pedal* in the semantic ontology such as:

> *A sustained bass note.*

Similarly, the definition for *noises* states:

> *Electrical or acoustic activity that can disturb communication.*

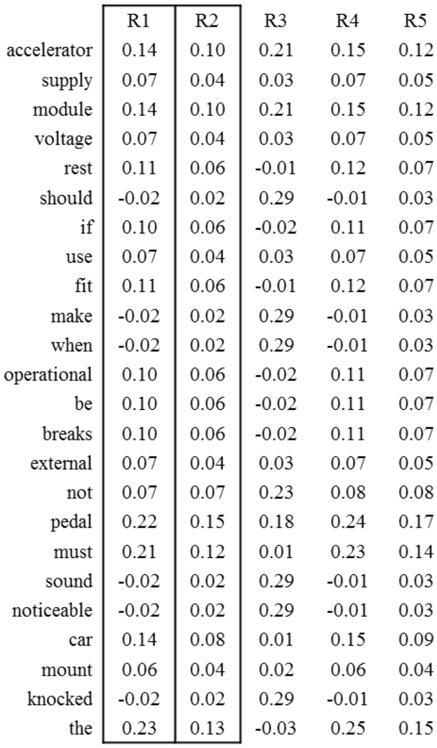
As with *pedal* this meaning was automatically mapped to by the Semantic Analyzer. It was chosen over other meanings in the ontology such as:

> *A loud outcry of protest or complaint.*

As validated by Test Case 1 in Section 4.4, the method of supplying semantics to the requirements list has been demonstrated by this research to be at least 74% accurate.

5.1.2 RO 2: Forming Semantic Relationships

The requirement analyzers create conceptual models that relate requirements to one another. While the Syntax Analyzer uses only string matching and syntax to draw relationships between requirements, the other three requirement analyzers implement semantics. In the LSA Analyzer and the Component LSA Analyzer, the reduced-term document matrix is used to create requirement vectors. These vectors are then related using cosine similarity. For example, on a sample of five requirements from the accelerator pedal module requirements document, five requirement vectors are obtained via the reduced term-document matrix shown in Figure 5-1. Figure 5-1 has two of the column vectors highlighted in boxes.



	R1	R2	R3	R4	R5
accelerator	0.14	0.10	0.21	0.15	0.12
supply	0.07	0.04	0.03	0.07	0.05
module	0.14	0.10	0.21	0.15	0.12
voltage	0.07	0.04	0.03	0.07	0.05
rest	0.11	0.06	-0.01	0.12	0.07
should	-0.02	0.02	0.29	-0.01	0.03
if	0.10	0.06	-0.02	0.11	0.07
use	0.07	0.04	0.03	0.07	0.05
fit	0.11	0.06	-0.01	0.12	0.07
make	-0.02	0.02	0.29	-0.01	0.03
when	-0.02	0.02	0.29	-0.01	0.03
operational	0.10	0.06	-0.02	0.11	0.07
be	0.10	0.06	-0.02	0.11	0.07
breaks	0.10	0.06	-0.02	0.11	0.07
external	0.07	0.04	0.03	0.07	0.05
not	0.07	0.07	0.23	0.08	0.08
pedal	0.22	0.15	0.18	0.24	0.17
must	0.21	0.12	0.01	0.23	0.14
sound	-0.02	0.02	0.29	-0.01	0.03
noticeable	-0.02	0.02	0.29	-0.01	0.03
car	0.14	0.08	0.01	0.15	0.09
mount	0.06	0.04	0.02	0.06	0.04
knocked	-0.02	0.02	0.29	-0.01	0.03
the	0.23	0.13	-0.03	0.25	0.15

Figure 5-1: Sample rank-reduced term-document matrix on five requirements highlighting column vectors R1 and R2.

Performing the cosine similarity between vectors R1 and R2 in Figure 5-1, yields the relationship value between the two requirements. Requirement 1 is:

If pedal breaks, the pedal must not be operational.

Requirement 2 is:

Accelerator pedal module must mount to car.

In this case, the cosine similarity value between these two requirements is 0.97. The cosine similarity values are on a scale from -1.00 to 1.00 where 0.00 and less is no relationship and 1.00 is identical. While 0.97 seems intuitively high for these requirements, LSA is based upon the given context. With a sample of 5 requirements, the context is not complete and the given example serves only as a way to demonstrate the method.

The Semantic Analyzer maps to a semantic ontology then relates the requirements based upon the connections within the ontology. For instance, requirement 11 states:

The requirements refer to an electrical wiring output.

Requirement 13 states:

If pedal breaks, the pedal must not be operational.

Identifying the significant terms in these requirement statements yields *requirements* in requirement 11 and *pedal* in requirement 13. Traversing the ontology provides a similarity value on a scale from 0.00 to 1.00 where 0.00 is no relationship and 1.00 is synonymy. In this instance, the obtained value between *pedal* and *requirements* is 0.11.

These connections consider semantic relationships such as synonymy, hypernymy, hyponymy, and meronymy. The output conceptual models from these analyses are $n \times n$ DSMs, where n is the number of requirements. The values in the DSM identify the strength of relation between requirements in a requirements document. The value of these conceptual models to requirement analysis and engineering design are evaluated by RO 3.

5.1.3 RO 3: Value of Linguistic Semantics to Design

Chapter Four seeks to understand the application of supplying semantics to a requirements document. The requirement analyzers developed in Chapter Three are each applied to a requirements document for a BMW accelerator pedal module to find the relationships between requirements. In addition, three engineers were asked to individually draw relationships between requirements. Further, a study was conducted where a group of three engineers collaboratively drew relationships between the requirements.

To judge the ability of the requirement analyzers, all results are compared to the manual collaborative study. The comparisons are performed in Test Case 1 (Section 4.3) and Test Cases 3-6 (Sections 4.5-4.8). The ability of each requirement analysis is quantified by the level of agreement between the manual collaborative study and each analyzer. The level of agreement was measured using Cohen’s Kappa. Test Case 1 showed that conceptual models between engineers moderately agree as the highest kappa value between an individual engineer and the collaborative study was 0.61. Test Cases 3-6 showed that none of the analyzers perform as well as the manually obtained results.

However, when comparing the analyzers that implement semantics against the Syntax Analyzer, the semantic analyzers agree more with manually found concepts.

Further statistical analysis of the requirement analyzers provides reasoning for how the analyzers performed against the manual analysis. Table 5-1 shows the statistical analysis of all analyzers at 0.6 semantic threshold.

Table 5-1: Statistical analysis results at 0.6 semantic threshold. Cohen's Kappa value is in relation to the manual collaborative study results.

Analyzer	True Positives	True Negatives	False Positives	False Negatives	Cohen's Kappa
Syntax	18	181	54	23	.26
LSA	11	211	24	30	.35
Component LSA	18	190	45	23	.33
Semantic	9	226	9	32	.43

In certain instances, all analyzers identified relationships between requirements that agreed with the collaborative study. This corresponds to the overlap of true positive throughout all analyzers. For instance, requirement 12 states:

> *If pedal breaks, the pedal must not be operational.*

Requirement 16 states:

> *If pedal breaks, the pedal must be recognizable as broken.*

All four analyzers from Table 5-1 identified the relationship between these two requirements. Identification of this relationship is straightforward because of the strong text similarity and string matched subjects. For the two LSA analyzers, LSA creates vectors based on the importance of terms in a statement—and importance of latent terms that should be in a statement. Therefore, if two requirements have a large portion of similar text, they are more likely to be related as with requirements 12 and 16. The semantic analyzer captures the term *pedal* in each requirement and maps it to the ontology where the ontology realizes the identical terms are direct synonyms.

The LSA Analyzer, which evaluates the whole requirement statement, is able to identify relationships that engineers identified but the Syntax Analyzer did not identify. For instance, requirement 9 states:

The sensor must be redundant with respect to the output voltage.

Requirement 11 states:

The requirements refer to an electrical wiring output.

The manual collaborative study related these two requirements where the Syntax Analyzer did not but the LSA Analyzer did. The use of LSA on the entire requirement allowed for a computational selection of the significant terms of the requirement statement as opposed to the Syntax Analyzer where the subject and direct object were pre-selected as significant terms. This allowed for the LSA Analyzer to compare the terms *output* in both requirements. Also the LSA Analyzer compared other terms such as *sensor*, *electrical, wiring*, and *voltage*, which often appear in the same statement and therefore are latently related. Similarly, requirements 7 and 18 were related by the LSA Analyzer. Requirement 7 states:

Pedal angle generated by the actuating force must be limited.

Requirement 18 states:

> *Pedal must not exhibit "stick slip effect" (instability in the force angle – response behavior).*

The Syntax Analyzer did not draw the relationship between requirements 7 and 18, where the LSA Analyzer and manual study did. Again, LSA enabled the analyzer to select the significant terms such as *force* and *angle* where the Syntax Analyzer selected *pedal*. This selection of terms, plus the ability to weight the terms' importance and find the latent significance of terms in a requirement resulted in the relating of these two requirements.

Another reason for the increase in agreement between the results of the semantic analyzers and manual analysis is the minimization of the false positives between a semantic analyzer and the manual analysis. For instance, while the Syntax Analyzer identified the most true positives (18) at a semantic threshold of 0.6, it had the most false positive relationships (54). This means that 54 relationships were found between requirements by the Syntax Analyzer that the manual analysis did not. Conversely, the Semantic Analyzer, which agreed most with the manual results, only had nine false positives.

Table 5-1 shows that the Component LSA Analyzer matches or outperforms the Syntax Analyzer in every category. It has as many true positives and false negatives as the Syntax Analyzer while improving in true negatives and false positives. This is evident in the improved agreement between the Component LSA Analyzer as seen in the Cohen's Kappa value as compared to the Syntax Analyzer.

These results show that enhancing requirement analysis with semantics improves the computational ability to differentiate between related requirements and unrelated requirements. For instance, requirements 2 and 17 of the BMW accelerator pedal module are related by the Syntax Analyzer but not by any of the analyzers that use semantics. Requirement 2 reads:

> *Accelerator pedal module may not radiate disturbing mechanical noises (airborne sounds or mechanical vibrations).*

Requirement 17 reads:

> *Accelerator pedal module must not send a signal to the electronic control unit against the driver's wishes.*

The Syntax Analyzer identifies that the subjects of both requirements, *pedal*, are string matched and therefore related. However, the semantic analysis shows that while the two requirements share the same subject, the remaining terms are not semantically related and therefore there is no relationship between the requirements. By implementing semantics, computational methods can better filter requirement relationships that string matching methods identify that are not identified by engineers.

The analyzers in this research are applied to requirement analyses that draw relationships between requirements. Research in the field of change propagation in requirements uses a formal model that relates requirements to one another [12, 13]. The goal of the change propagation tool is to predict what other requirements will change if a requirement is changed. The current method uses string matching and syntax to draw relationships. This research has shown that using semantics can create relationships that agree more with manually found relationships. Implementation of semantics in the change propagation tool could further the capabilities of this tool. Further, the keywords to use for drawing relationships are performed manually. Implementing a method such as LSA enables this method to be automated.

5.2 Broader Impact of Semantics in Requirement Analysis

This research extends the computational support available for engineering design. By improving requirement analysis, the requirements document can be refined earlier. This refinement adds value to the design by saving effort and money that would be needed if the design was refined later in the process. Computational support is beneficial as it seeks to not only improve the results of requirement analysis, but also increase the efficiency of the design process. By automating a tedious process, designers can accomplish the task quicker and/or while performing another task in parallel.

5.3 Future Work

5.3.1 Integration of Analyzer with Existing Tools

The requirement analyzers have been shown to apply linguistic semantics to requirements and find implicit relationships between requirements that cannot be found by string matching or syntactic methods. Most existing requirement analysis tools do not employ a semantic analysis, and therefore cannot find these relationships [12, 9, 19, 13]. The next phase of for a semantic requirement analysis is to integrate it with an existing tool. Then, validation of a specific tool can be performed to show that the semantic analysis has improved the results of tool. The following research question summarizes the areas of future work identified for integration of the tool:

How can the semantic requirement analysis tool be integrated with an existing requirement analysis tool to improve the results?

5.3.2 Implementing Machine-Learning Techniques

While the semantic requirement analysis tool has been largely-automated, there are parameters that must be hard-coded into the analysis that could change between requirement documents. For instance, the performed LSA has to define a rank value to which the term-document matrix is reduced. Parameters such as this dimension value can be input into a vector along with defining parameters of the requirements document. This vector can then be used to optimize the analysis for each requirements document using machine-learning techniques such as classifiers or neural nets. This area of future work would allow for further automation and adaptability of the tool. The research question for this future work is:

How can machine-learning techniques improve a semantic interpretation of a requirements document?

5.3.3 Further Validation of Tool

The application of semantics to requirement terms has been shown by this research to be at least 74% accurate. However, validation on requirements documents of different sizes, designs, formats, and phases in the design process should be considered. Hypotheses about these results can be drawn based on the implemented method of applying semantics and the performed validation. The semantic analysis relies on the context provided by the requirements document. As long as the full context of the design is represented in the requirements document, the accuracy of the tool should not change significantly. This means that different size, designs, and formats should not significantly affect the accuracy. However, as the requirements document is refined over the phases of the design process, more information is known about the design. This may indicate that the accuracy could be better on a later revision of a requirements document. The research question for this area of future work is:

How does the variation across requirements documents affect the ability of the tool to provide reliable results?

REFERENCES

[1] G. Pahl, W. Beitz, J. Feldhusen and K. H. Grote, Engineering Design: A Systematic Approach, London: Springer-Verlag, 2007.

[2] K. T. Ulrich and S. D. Eppinger, Product Design and Development, McGraw-Hill Inc., 1995.

[3] K. N. Otto and K. L. Wood, Product Design, Upper Saddle River: Prentice-Hall, Inc., 2001.

[4] V. Berzins, C. Martell, Luqi and P. Adams, "Innovations in Natural Language document Processing for Requirements Engineering," *Monterey Workshop 2007,* pp. 125-146, 2008.

[5] O. Ormandjieva, I. Hussain and L. Kosseim, "Toward a text classification system for the quality assessment of software requirements written in natural language," in *SOQUA*, Dubrovnik, 2007.

[6] M.-C. d. Marneffe, B. MacCartney and C. D. Manning, "Generating Typed Dependency Parses from Phrase Structure Parses," in *Language Resources and Evaluation Conference*, Genoa, 2006.

[7] C. Lamar, *Linguistic Analysis of Natural Language Engineering Requirements,* M.S. Thesis: Clemson University, USA, 2009.

[8] C. Lamar and G. M. Mocko, "Linguistic Analysis of Natural Language Engineering Requirement Statements," in *TMCE 2010*, Ancona, Italy, 2010.

[9] Z. Y. Chen, S. Yao, J. Q. Lin, Y. Zeng and A. Eberlein, "Formalisation of product requirements: from natural language descriptions to formal specifications," *Int. J. Manufacturing Research,* vol. 2, no. 3, pp. 362-387, 2007.

[10] W. M. Wilson, L. H. Rosenberg and L. E. Hyatt, "Automated Quality Analysis of Natural Language Requirement Specifications," *NASA Software Assurance Technology Center.*

[11] F. Fabbrini, M. Fusani, S. Gnesi and G. Lami, "The Linguistic Approach to the Natural Language Requirements Quality: Benefit of the use of an Automatic Tool," in *26th Annual NASA Goddard Software Engineering Workshop,* Washington DC, USA, 2001.

[12] B. W. Morkos, Computational Representation and Reasoning Support for Requirements Change Management in Complex System Design, Clemson: Clemson Department of Mechanical Engineering, 2012.

[13] B. Morkos, P. Shankar and J. Summers, "Predicting Requirement Change Propagation Using Higher Order Design Structure Matrice: An Industry Case Study of Engineering Design," *Journal of Engineering Design,* vol. 23, no. 12, pp. 905-926, 2011.

[14] G. A. Miller, "WordNet: A Lexical Database for English," *Communications of the ACM,* vol. 38, no. 11, pp. 39-41, 1995.

[15] D. Ott, "Defects in Natural Language Requirement Specifications at Mercedes-Benz: An Investigation Using a Combination of Legacy data and Expert Opinion," in *IEEE International Conference on Requirements Engineering,* Chicago, 2012.

[16] A. Ferrari and S. Gnesi, "Using Collective Intelligence to Detect Pragmatic Ambiguities," in *IEEE International Requirements Engineering Conference,* Chicago, 2012.

[17] H. Krishnan and P. Samuel, "Relative Extraction Methodology for Class Diagram Generation using Dependency Graph," in *IEEE ICCCT*, 2010.

[18] E.-V. Chioasca, "Using Machine Learning to Enhance Automated Requirements Model Transformation," in *IEEE ICSE 2012*, Zurich, 2012.

[19] L. Chen and Y. Zeng, "Automatic Generation of UML Diagrams from Product Requirements Described by Natural Language," in *ASME 2009 International Design Engineering Technical Conferences & Computer and Information in Engineering Conference,* San Diego, 2009.

[20] J. M. McLellan, A Proposed Method to Identify Requirements Significant to Mass Reduction, Clemson: Clemson University, 2010.

[21] J. M. McLellan, J. R. Maier, G. M. Fadel and G. Mocko, "A Method for Identifying Requiremetn Critical to Mass Reduction," in *International Design Structure Matrix Conference,* Greenville, SC, 2009.

[22] J. Zelle, Python Programming: An Introduction to Computer Science, Wilsonville, OR: Franklin, Beedle & Associates Inc., 2004.

[23] S. Bird, E. Klein and E. Loper, Natural Language Processing with Python, Sebastopol, California: O'Reilly Media Inc., 2009.

[24] BMW, "RealOEM.com Online BMW Parts Catalog," [Online]. Available: http://realoem.com/bmw/. [Accessed 18 April 2013].

APPENDICES

5.4 Appendix A: Analyzer Scripts

This appendix contains the primary scripts for each of the requirement analyzers as well as all of the functions used. The coding is written in Python. The scripts and functions are delimited by file.

1. Semantic Analyzer

```
# main.py

# By: Alex Lash
# Created: 2/14/13
# Last modified: 2/22/13

# Executable that takes text-based dependency parses and creates DSM
# conceptual models.

from deptree_text_to_obj import *
from requirement import Requirement
from stopword_remove import *
from lsa import *
from latent_analysis import *
from ontology_analysis import *
from semantic_mapping import *
from compare import *

def main():

    # Open file of requirements.
    # Either raw NL (slower) or tagged requirement statements (faster).

    deptree_str_lst = open_file().split('\n\n')

    deptree_lst = []

    for deptree_str in deptree_str_lst:

        deptree = deptree_str_to_obj(deptree_str)

        deptree_lst.append(deptree)

    print deptree_lst

    nouns_lst_by_req = []
##    verbs_lst_by_req = []
    tokens_lst_by_req = []

    # Get artifacts and functions
    for deptree in deptree_lst:

        nouns_lst = deptree.nouns(gram_rel = ['nsubjpass','nsubj','dobj'])
##        verbs_lst = deptree.verbs(gram_rel = ['dobj'])
        tokens_lst = deptree.tokens(numtags=False)
        nouns_lst = list(set(nouns_lst))
##        verbs_lst = list(set(verbs_lst))

        nouns_lst_by_req.append(nouns_lst)
##        verbs_lst_by_req.append(verbs_lst)
        tokens_lst_by_req.append(tokens_lst)

    # Remove stopwords
    nouns_lst_by_req = remove_stops(nouns_lst_by_req)

##    verbs_lst_by_req = remove_stops(verbs_lst_by_req)

    print("\nAll requirements extracted.")

    # Run LSA on requirements.
    print("\nRun LSA...")

    dim_cutoff_val = int(raw_input("\nLSA dimension cutoff value (0 - # reqs):
"))

    print("\n     Running LSA on nouns...")

    n_td_mat,nouns = lsa(nouns_lst_by_req, csv=True, normalize=False,
s_cutoff=dim_cutoff_val)

    print("\n     Complete.")

##    print("\n     Running LSA on verbs...")

##        v_td_mat,verbs = lsa(verbs_lst_by_req, csv=False, normalize=False,
s_cutoff=dim_cutoff_val)

##    print("\n     Complete.")

    print("\n     Running general LSA...")

    td_mat,tokens = lsa(tokens_lst_by_req, csv=False, log_entropy=True,
s_cutoff=dim_cutoff_val)

    print("\n     Complete.")

    print("\nLSA complete.")

    print
    print nouns
##    print
##    print verbs

    # Get latent keywords.
    print("\nFinding latent word relationships...")

    sim_cutoff_val = float(raw_input("\nSimilarity cutoff value (0 - 1): "))

    print("\n    Finding similar nouns in requirements document...")

    n_latent_dict = latent_keywords(td_mat,tokens,sim_val = sim_cutoff_val)

    noun_dict = modify_dict(n_latent_dict,nouns)

    print "noun_dict"
    print noun_dict

    print("\n    Similar nouns in requirements document found.")

##    print("\n    Finding similar verbs in requirements document...")

##    v_latent_dict = latent_keywords(td_mat,tokens,sim_val = sim_cutoff_val)

##    verb_dict = modify_dict(v_latent_dict,verbs)

##    print("\n    Similar verbs in requirements document found.")

    print("\nLatent relationships found.")

    # Get ontological keywords.
    print("\nFinding possible definitions for words...")

    def_cutoff_val = int(raw_input("\nMax number of definitions per word (1 -
100): "))

    print("\n     Finding possible artifacts...")

    artifact_dict = synset_keywords(nouns,count_max=def_cutoff_val,p_of_s='n')

    print
    print artifact_dict

    print("\n     Possible artifacts found.")

##    print("\n     Finding possible functions...")

##                                                   function_dict             =
synset_keywords(verbs,count_max=def_cutoff_val,p_of_s='v')

##    print("\n     Possible functions found.")

    print("\nDefinitions found...")

    # Map nouns to artifacts.
    print("\nPerforming semantic analysis...")

    print("\n     Defining each noun term (mapping to an artifact)...")

    n_semantic_dict = define_terms(noun_dict,artifact_dict,pos='n')

    print
    print n_semantic_dict

    print("\n     Noun terms defined.")

##    print("\n     Defining each verb term (mapping to a function)...")

##    v_semantic_dict = define_terms(verb_dict,function_dict,pos='v')

##    print("\n     Verb terms defined.")

    print("\n     Relating noun terms to requirements...")

    n_semantic_lst_by_req = mapping(nouns_lst_by_req,n_semantic_dict)
    print
    print nouns_lst_by_req

    print("\n     Noun terms related to requirements.")

##    print("\n     Relating verb terms to requirements...")

##    v_semantic_lst_by_req = mapping(verbs_lst_by_req,v_semantic_dict)

##    print("\n     Verb terms related to requirements.")

    print
    print n_semantic_lst_by_req

    for lst in n_semantic_lst_by_req:

        print

        for i in lst:

            try:
                print i.definition

            except:
                print i

    print("\nSemantic analysis complete.")

    # Pairwise compare requirements
    print("\nCreating DSM and outfiles for visualization...")

    n_dsm = ontology_dsm(n_semantic_lst_by_req,name="n_sem")

##    v_dsm = ontology_dsm(v_semantic_lst_by_req,name="v")

    print"\nFiles written."

# ------------------------------------------------------------------------

def tag_check():

    print('Is the input file pretagged?')
    print('     (1) Yes')
    print('     (2) No')

    while True:

        response = raw_input()

        if response == '1' or response.lower() == 'yes':

            return True

        if response == '2' or response.lower() == 'no':

            return False

        print('Not a valid response\nType (1) Yes or (2) No.')

# ------------------------------------------------------------------------
```

```
def open_file():

    while True:

        fname = raw_input("File Name: ")

        try:

            infile = open(fname,'r')

            break

        except:

            print('File not found. Try again. ctrl-C to exit.\n')

    f = infile.read()

    infile.close()

    return f

# -------------------------------------------------------------------------

if __name__ == "__main__":
    main()
```

2. LSA & Component LSA Analyzers

```
# lsa_only.py

# By: Alex Lash
# Created: 2/14/13
# Last modified: 2/22/13

# Executable that takes text-based dependency parses and creates DSM
# conceptual models.

from deptree_text_to_obj import *
from requirement import Requirement
from stopword_remove import *
from lsa import *
from compare import *

def main():

    # Open file of requirements.
    # Either raw NL (slower) or tagged requirement statements (faster).

    deptree_str_lst = open_file().split('\n\n')

    deptree_lst = []

    for deptree_str in deptree_str_lst:

        deptree = deptree_str_to_obj(deptree_str)
        deptree_lst.append(deptree)

    print deptree_lst

    nouns_lst_by_req = []
##    verbs_lst_by_req = []
    tokens_lst_by_req = []

    # Get artifacts and functions
    for deptree in deptree_lst:

        nouns_lst = deptree.nouns(gram_rel = ['nsubjpass','nsubj','dobj'])
##        verbs_lst = deptree.verbs(gram_rel = ['dobj'])
        tokens_lst = deptree.tokens(numtags=False)

        nouns_lst = list(set(nouns_lst))
##        verbs_lst = list(set(verbs_lst))

        nouns_lst_by_req.append(nouns_lst)
##        verbs_lst_by_req.append(verbs_lst)
        tokens_lst_by_req.append(tokens_lst)

    # Remove stopwords
    nouns_lst_by_req = remove_stops(nouns_lst_by_req)

##    verbs_lst_by_req = remove_stops(verbs_lst_by_req)

    print("\nAll requirements extracted.")

    # Run LSA on requirements.
    print("\nRun LSA...")

    dim_cutoff_val = int(raw_input("\nLSA dimension cutoff value (0 - # reqs): 
"))

    print("\n     Running LSA on nouns...")

    n_td_mat,nouns = lsa(nouns_lst_by_req, csv=False, normalize=False, 
s_cutoff=dim_cutoff_val)

    print("\n     Complete.")

    print("\n     Running LSA on verbs...")

##        v_td_mat,verbs = lsa(verbs_lst_by_req, csv=False, normalize=False, 
s_cutoff=dim_cutoff_val)

    print("\n     Complete.")

    print("\n     Running general LSA...")

    td_mat,tokens = lsa(tokens_lst_by_req, csv=True, log_entropy=True, 
s_cutoff=dim_cutoff_val)

    print("\n     Complete.")

    print("\nLSA complete.")

    print
    print nouns
```

```
##    print
##    print verbs

    # Pairwise compare requirements
    print("\nCreating DSM and outfiles for visualization...")

    dsm = lsa_dsm(n_td_mat,name="n_lsa")
    dsm_too = lsa_dsm(td_mat,name="all_lsa")

    print"\nFiles written."



# ------------------------------------------------------------------------

def tag_check():

    print('Is the input file pretagged?')
    print('     (1) Yes')
    print('     (2) No')

    while True:

        response = raw_input()

        if response == '1' or response.lower() == 'yes':

            return True

        if response == '2' or response.lower() == 'no':

            return False

        print('Not a valid response\nType (1) Yes or (2) No.')

# ------------------------------------------------------------------------

def open_file():

    while True:

        fname = raw_input("File Name: ")

        try:

            infile = open(fname,'r')

            break

        except:

            print('File not found. Try again. ctrl-C to exit.\n')

    f = infile.read()

    infile.close()

    return f

# ------------------------------------------------------------------------
```

```
if __name__ == "__main__":
    main()
```

3. Syntax Analyzer

```
# string_matching.py

# By: Alex Lash
# Created: 2/14/13
# Last modified: 2/22/13

# Executable that takes text-based dependency parses and creates DSM
# conceptual models.

from deptree_text_to_obj import *
from requirement import Requirement
from stopword_remove import *
##from lsa import *
##from latent_analysis import *
##from ontology_analysis import *
##from semantic_mapping import *
from compare import *

def main():

    # Open file of requirements.
    # Either raw NL (slower) or tagged requirement statements (faster).

    deptree_str_lst = open_file().split('\n\n')

    deptree_lst = []

    for deptree_str in deptree_str_lst:

        deptree = deptree_str_to_obj(deptree_str)

        deptree_lst.append(deptree)

    print deptree_lst

    nouns_lst_by_req = []
##    verbs_lst_by_req = []
##    tokens_lst_by_req = []

    # Get artifacts and functions
    for deptree in deptree_lst:

        nouns_lst = deptree.nouns(gram_rel = ['nsubjpass','nsubj','dobj'])
##        verbs_lst = deptree.verbs(gram_rel = ['dobj'])
##        tokens_lst = deptree.tokens(numtags=False)

        nouns_lst = list(set(nouns_lst))
##        verbs_lst = list(set(verbs_lst))

        nouns_lst_by_req.append(nouns_lst)
##        verbs_lst_by_req.append(verbs_lst)
##        tokens_lst_by_req.append(tokens_lst)
```

```
    # Remove stopwords
    nouns_lst_by_req = remove_stops(nouns_lst_by_req)

##    verbs_lst_by_req = remove_stops(verbs_lst_by_req)

    print("\nAll requirements extracted.")

    # Pairwise compare requirements
    print("\nCreating DSM and outfiles for visualization...")

    n_dsm = string_dsm(nouns_lst_by_req,name="n_str")

##    v_dsm = string_dsm(verbs_lst_by_req,name="v")

    print"\nFiles written."

# -----------------------------------------------------------------------

def tag_check():

    print('Is the input file pretagged?')
    print('     (1) Yes')
    print('     (2) No')

    while True:

        response = raw_input()

        if response == '1' or response.lower() == 'yes':

            return True

        if response == '2' or response.lower() == 'no':

            return False

        print('Not a valid response\nType (1) Yes or (2) No.')

# -----------------------------------------------------------------------

def open_file():

    while True:

        fname = raw_input("File Name: ")

        try:

            infile = open(fname,'r')

            break

        except:

            print('File not found. Try again. ctrl-C to exit.\n')

    f = infile.read()

    infile.close()
    return f

# ------------------------------------------------------------------------

if __name__ == "__main__":
    main()
```

4. Dependency and Dependency Tree Classes

```
# dependency.py
# Set of classes for dependency trees in Python.

from copy import deepcopy

class Dependency:

    def __init__(self,token,gram_rel,dep_token):

        """Create a dependency object consisting of a token, a grammatical
relation
        , and a dependent token."""

        # Create token
        self.token = token

        # Creat grammatical relation
        self.gram_rel = gram_rel

        # Relate to a dependent token
        self.dep_token = dep_token

    def token(self):

        """Finds the token of the dependency."""

        return self.token

    def gram_rel(self):

        """Returns the grammatical relation of the dependency."""

        return self.gram_rel

    def dep_token(self):

        """Returns the dependent token of the dependency."""

        return self.dep_token

    def obj(self):

        """Builds an image of the dependency object for viewing, comparing,
etc."""

        return (self.token,self.gram_rel,self.dep_token)

# ------------------------------------------------------------------------
```

```
class DependencyTree:

    def __init__(self):

        """Create empty dependency tree."""

        self.dep_list = []

    def insert(self,token,gram_rel,dep_token):

        """Add a dependency object to the tree."""

        # Create dependency object using Dependency class
        dep_obj = Dependency(token,gram_rel,dep_token)

        # If dependency not already in tree add the dependency to the tree.
        if not self.__in(dep_obj,self.dep_list):
            self.dep_list.append(dep_obj)

        # Otherwise, state that the dependency already exists.
        else:
            print ('Dependency object:  {obj}\nalready exists in tree. Object
not added.\n'.format(obj=dep_obj.obj()))

    def __search(self, token=[], gram_rel=[], dep_token=[]):

        """Input a list of tokens, grammatical relations, and/or dependant
tokens
        as search parameters. Returns a list of dependency objects matching
        parameters."""

        # Create a blank list that will have searched data added to be
returned.
        return_list = []


        # Traverse tree to find dependencies that match searched criteria.
        for dep_item in self.dep_list:
            if (dep_item.token[0] in token or not token) and (dep_item.gram_rel
in gram_rel or not gram_rel) and (dep_item.dep_token[0] in dep_token or not
dep_token):
                return_list.append(dep_item)

        return return_list

    def verbs(self, gram_rel =
['subj','csubj','nsubjpass','nsubj','obj','dobj','iobj','agent']):

        """Find significant verbs from the dependency tree. Defaults are verbs
of subjects
        and objects"""

        # Traverse dependency tree and find verbs. Default is root verb.
        dep_list = self.__search([],gram_rel,[])

        verb_list = []

        # From the found dependencies pull out the verbs and put into a new
list to return.
        for dep in dep_list:
            verb_list.append(dep.token[0])

        return verb_list

    def nouns(self, gram_rel =
['subj','csubj','nsubjpass','nsubj','obj','dobj','iobj','nn']):

        """Find significant nouns from the dependency tree. Default are
subjects
        and objects."""

        # Traverse dependency tree and find nouns. Defaults are objects and
subject.
        dep_list = self.__search([],gram_rel,[])

        noun_list = []

        # From the found dependencies pull out the nouns and put into a new
list to return.
        for dep in dep_list:
            noun_list.append(dep.dep_token[0])

        return noun_list

    def tokens(self,numtags=True):

        """Return a list of all the tokens in the dependency tree as a list of
tuples."""

        # Traverse tree to return all dependencies.
        dep_list = self.__search()

        token_list = []

        # Create a list of all the tokens (including dependent tokens).
        for dep in dep_list:
            token_list.append(dep.dep_token)
            token_list.append(dep.token)

        # Delete duplicates due to dependencies having multiple grammatical
relations
        # and/or being both a token and a dependent token in different
dependencies.
        token_list = list(set(token_list))

        # Remove tree root because it is an implied token.
        token_list.remove(('root',0))

        word_lst = []
        if not numtags:
            for token in token_list:
                word_lst.append(token[0])

            token_list = word_lst

        return token_list

    def __in(self,dep_obj,dep_lst):

        """Modified in function to correctly find if a dependency object is in
        a list of dependency objects."""

        for dep_item in dep_lst:
            if dep_item.obj() == dep_obj.obj():
                return True

        return False
```

5. Dependency Syntax Conversion Functions

```
# Convert a string
# Function that takes a string dependency tree parse and converts it to a
# Python object.
# Input a string dependency tree parse having dependences delimited with
# '\n' --> Output a dependency tree object.

# Main function: 'deptree_str_to_obj'
# Could suppress functions 'word_str_to_tup' and 'dep_str_to_lst' but left
# available for possible utility.

from dependency import *

def word_str_to_tup(word_str):

    """Reformats the string 'Word-#' as the tuple (word,#)"""

    rev_word_lst = word_str[::-1].split('-',1)[::-1]

    word_lst = []
    for word in rev_word_lst:
        word_lst.append(word[::-1])

##    print word_lst

    word_lst[1] = int(word_lst[1])

    word_lst[0] = word_lst[0].lower()

    word_tup = tuple(word_lst)

    return word_tup

def dep_str_to_lst(dep_str):

    """Refomats the string 'dep(word-#, dep_word-#)'
    as ['dep', ('word', '#'), ('dep_word', '#')]"""

    # Replace all delimiting characters with ',' and remove whitespace.
    dep_str = dep_str.replace('(',',')
    dep_str = dep_str.replace(')','')
    dep_str = dep_str.replace(' ','')

    # Split along delimiter ','.
    dep_as_lst = dep_str.split(',')

    # Use function 'word_str_to_tup' to reformat the two tokens in the
    # dependency list as tuples.
    for i in range(2):
        word_str = dep_as_lst.pop(1)
        word_tup = word_str_to_tup(word_str)
        dep_as_lst.append(word_tup)

    return dep_as_lst

def deptree_str_to_obj(deptree_str):

    """Converts a string dependency tree from Stanford Parser output to
    a DependencyTree object."""

    # Split dependencies along delimiter '\n'.
    deptree_lst = deptree_str.split('\n')

    # Create an empty DependencyTree object to be populated with Dependency
    # objects.
    deptree_obj = DependencyTree()

    # Populate dependency tree with dependencies
    for dep_str in deptree_lst:

        dep_as_lst = dep_str_to_lst(dep_str)
        deptree_obj.insert(dep_as_lst[1],dep_as_lst[0],dep_as_lst[2])

    return deptree_obj
```

6. Stopword Removal Function

```
# stopword_remove.py

# By: Alex Lash
# Created: 2/22/13
# Last modified: 2/22/13

from nltk.corpus import stopwords
from latent_analysis import alpha

# ------------------------------------------------------------------------

def remove_stops(word_lst_by_req):

    stops = set(stopwords.words('english'))

    keep_lst_by_req = []

    for word_lst in word_lst_by_req:

        keep_lst = [word for word in word_lst if word not in stops]

        final_keep_lst = []

        for term in keep_lst:

            last_char_ascii = ord(term[-1])

            if len(term)>1 and alpha(last_char_ascii):

                final_keep_lst.append(term)

        keep_lst_by_req.append(final_keep_lst)
    return keep_lst_by_req
```

7. Latent Semantic Analysis Functions

```
# lsa.py

# By: Alex Lash
# Created: 1/31/13
# Last modified: 2/19/13

# Set of functions that runs latent semantic analysis on a set of documents and
# provides a matrix of the completed LSA as output.
# Input a list of lists containing the tokenized documents/requirements -->
# outputs a list of numpy arrays that is a SVD document by word matrix.

from numpy import *
from math import *

## -----------------------------------------------------------------------

def tf(lst_of_docs,csv=False):

    # Preallocate dictionaries to store values for calculations

    # Pre-allocate dictionary to store global frequency
    gf = {} # term <Type = str> : collection count <Type = int>

    # Pre-allocate dictionary to store term frequency for each term per
document
    tf = {} # term <Type = str> : count per doc <Type = list of ints>

    num_docs = 0

    # Create gf
    for doc in lst_of_docs:

        for word in doc:

            try:
                gf[word] += 1

            except:
                gf[word] = 1

        num_docs += 1

    # Create tf
    for term in gf:

        count_lst = []

        for doc in lst_of_docs:
            count_lst.append(doc.count(term))

        tf[term] = count_lst

    # Create Term-Document Matrix
    # Pre-allocate term-document matrix
    td_mat = []

    # Add column headers to matrix, i.e. document numbers
    doc_header_row = ['']

    for i in range(num_docs):
        doc_header_row.append("R{doc_num}".format(doc_num = i+1))

    # Place header row with column headers into term-document matrix
    td_mat.append(doc_header_row)

    # Populate term-document matrix
    for term in tf:

        # Add row header, i.e. term
        term_row = [term]

        # Populate matrix with log-entropy tf-idf values
        for freq in tf[term]:
            term_row.append(freq)

        td_mat.append(term_row)

    if csv:
        to_csv(td_mat, fname="tf.csv")

    return td_mat


## ------------------------------------------------------------------------

def tfidf(lst_of_docs,csv=False):
    """Function that runs TF-IDF on a set of documents and provides matrix of
    the completed TF-IDF as output. Uses the TF-IDF model as shown on
    wikipedia under the LSI. Input a list of lists containing the
    tokenized documents/requirements --> outputs a list of lists that is
    the term-document matrix."""

    # Preallocate dictionaries to store values for calculations

    # Pre-allocate dictionary to store global frequency
    gf = {} # term <Type = str> : collection count <Type = int>

    #  Pre-allocate  dictionary  to  store  term  frequency  for  each  term  per
document
    tf = {} # term <Type = str> : count per doc <Type = list of ints>

    # Pre-allocate dictionary to store log values for each term per document
    L_ij = {} # term <Type = str> : log value per doc <Type = list of floats>

    #  Pre-allocate  dictionary  to  store  the  number  of  documents  in  which  each
term appears
    df = {} # term <Type = str> : sum value per doc <Type = list of floats>

    #  Pre-allocate  dictionary  to  store  entropy  global  weight  values  for  each
term per document
    g_i = {} # term <Type = str> : entropy value <Type = float>
    num_docs = 0

    # Create gf
    for doc in lst_of_docs:

        for word in doc:

            try:
                gf[word] += 1

            except:
                gf[word] = 1

        num_docs += 1

    # Create tf
    for term in gf:

        count_lst = []

        for doc in lst_of_docs:
            count_lst.append(doc.count(term))

        tf[term] = count_lst

    # Create L_ij
    for term in tf:

        log_lst = []

        for val in tf[term]:
            log_lst.append(log10(val+1))

        L_ij[term] = log_lst

    # Create df
    for term in tf:

        dfi = 0

        for i in range(len(tf[term])):

            if tf[term][i] > 0:
                dfi += 1

        df[term] = dfi

    # Create g_i
    for term in gf:
        g_i[term] = log2(float(num_docs)/(1+df[term]))

    return createMat(num_docs,L_ij,g_i,csv)

## ---------------------------------------------------------------------------

def logent(lst_of_docs,csv=False):

    """Function that runs TF-IDF on a set of documents and provides matrix of
    the completed TF-IDF as output. Uses the log-entropy model as shown on
    wikipedia under the LSI. Input a list of lists containing the
    tokenized documents/requirements --> outputs a list of lists that is
    the term-document matrix."""

    # Preallocate dictionaries to store values for calculations

    # Pre-allocate dictionary to store global frequency
    gf = {} # term <Type = str> : collection count <Type = int>

    # Pre-allocate dictionary to store term frequency for each term per
document
    tf = {} # term <Type = str> : count per doc <Type = list of ints>

    # Pre-allocate dictionary to store log values for each term per document
    L_ij = {} # term <Type = str> : log value per doc <Type = list of floats>

    # Pre-allocate dictionary to store probability values for each term per
document
    p_ij = {} # term <Type = str> : tf/gf per doc <Type = list of floats>

    # Pre-allocate dictionary to store entropy global weight values for each
term per document
    g_i = {} # term <Type = str> : entropy value <Type = float>

    num_docs = 0

    # Create gf
    for doc in lst_of_docs:

        for word in doc:

            try:
                gf[word] += 1

            except:
                gf[word] = 1

        num_docs += 1

    # Create tf
    for term in gf:

        count_lst = []

        for doc in lst_of_docs:
            count_lst.append(doc.count(term))

        tf[term] = count_lst

    # Create L_ij
    for term in tf:

        log_lst = []

        for val in tf[term]:
            log_lst.append(log10(val+1))

        L_ij[term] = log_lst

    # Create p_ij
    for term in tf:
        p_lst = []

        for val in tf[term]:
            p_lst.append(float(val)/gf[term])

        p_ij[term] = p_lst

    # Create g_i
    for term in p_ij:

        summ = 1

        for val in p_ij[term]:

            if val != 0:
                summ += ((val*log10(val))/log(num_docs))

        g_i[term] = summ

    return createMat(num_docs,L_ij,g_i,csv)

## ----------------------------------------------------------------------

def createMat(num_docs,L_ij,g_i,csv):

    # Create Term-Document Matrix

    # Pre-allocate term-document matrix
    td_mat = []

    # Add column headers to matrix, i.e. document numbers
    doc_header_row = ['']

    for i in range(num_docs):
        doc_header_row.append("R{doc_num}".format(doc_num = i+1))

    # Place header row with column headers into term-document matrix
    td_mat.append(doc_header_row)

    # Populate term-document matrix
    for term in L_ij:

        # Add row header, i.e. term
        term_row = [term]

        # Populate matrix with log-entropy tf-idf values
        for freq in L_ij[term]:
            term_row.append(g_i[term]*freq)

        td_mat.append(term_row)

    if csv:
        to_csv(td_mat)

    return td_mat

## ----------------------------------------------------------------------

def to_csv(mat, fname = "tf_idf.csv"):
    outfile = open(fname, 'w')

    for row in mat:

        for item in row:
            outfile.write("{0};".format(item))

        outfile.write("\n")

    outfile.close()

## ----------------------------------------------------------------------

def
lsa(lst_of_docs,csv=False,normalize=True,log_entropy=False,s_cutoff=999999):

    # Run TF-IDF

    if normalize:

        if log_entropy:
            td_mat = logent(lst_of_docs,csv)

        else:
            td_mat = tfidf(lst_of_docs,csv)

    else:
        td_mat = tf(lst_of_docs,csv)


    col_headers = td_mat.pop(0)

    row_headers = []

    for row in td_mat:
        row_headers.append(row.pop(0))

    td_array = array(td_mat)

    u,s,vT = linalg.svd(td_array,full_matrices=False)

    td_red = rank_reduce(u,s,vT,s_cutoff)

    if csv:

        lsa = td_red.tolist()

        num_docs = len(lsa[0])

        # Pre-allocate term-document matrix
        lsa_mat = []

        # Add column headers to matrix, i.e. document numbers
        lsa_mat.append(col_headers)

        # Populate term-document matrix
        j=0
        for term in row_headers:
            # Add row header, i.e. term
            term_row = [term]

            # Populate matrix with lsa matrix values
            for row_val in lsa[j]:
                term_row.append(row_val)

            lsa_mat.append(term_row)

            j+=1

##        print lsa_mat

        to_csv(lsa_mat,fname="lsa.csv")

    return (td_red,row_headers)

## -----------------------------------------------------------------------

def rank_reduce(u,s,vT,s_cutoff=99999999):

    if s_cutoff <= 0:
        s_cutoff = 1

    red_s = []

    cut_count = 0

    for s_val in s:

        if cut_count >= s_cutoff:
            break

        red_s.append(s_val)

        cut_count += 1

    red_dim = len(red_s)

    red_s = diag(red_s)

    red_u = u[:,:red_dim]

    red_vT = vT[:red_dim,:]

    return red_u.dot(red_s).dot(red_vT)
```

8. Latent Analysis Functions

```
# latent_analysis.py

# By: Alex Lash
# Created: 1/31/13
# Last modified: 2/4/13

from numpy import *
from math import *
from requirement import *
from nltk.corpus import stopwords


def cos_sim(u,v):

    return u.dot(v) / (sqrt(u.dot(u)) * sqrt(v.dot(v)))

# ------------------------------------------------------------------------

def dsm(red_td_mat):

    num_words = len(red_td_mat)

    dsm = zeros((num_words,num_words))

    i = 0
    j = 0

    for row in red_td_mat:

        for other_row in red_td_mat:

            dsm[i,j] = cos_sim(row,other_row)

            j += 1

        i += 1
        j = 0

    return dsm

# ------------------------------------------------------------------------

def latent_keywords(red_td_mat, tokens, sim_val = .9):

    sim_mat = dsm(red_td_mat)

    i = 0

    keyword_dict = {}

    for token in tokens:

        row = sim_mat[i].tolist()

        keywords = []

        j = 0

        for cos_sim in row:

            if cos_sim >= sim_val and token != tokens[j]:

                keywords.append(tokens[j])

            j += 1

        keyword_dict[token] = keywords

        i += 1
    return keyword_dict

# ------------------------------------------------------------------------

def modify_dict(dictionary,words_to_keep):

    stops = set(stopwords.words('english'))

    for term in dictionary:

        definition = dictionary[term]

        dictionary[term] = [word for word in definition if word not in stops]

    keep_dict = {}

    for word in words_to_keep:

        last_char_ascii = ord(word[-1])

        if len(word)>1 and alpha(last_char_ascii):

            keep_dict[word] = dictionary[word]

    for term in keep_dict:

        definition = keep_dict[term]

        keep_items = []

        for item in definition:

            last_char_ascii = ord(item[-1])

            if len(item)>1 and alpha(last_char_ascii):

                keep_items.append(item)

        keep_dict[term] = keep_items

    return keep_dict

# ------------------------------------------------------------------------

def alpha(ascii_code):

    if 65 <= ascii_code <= 90:
        return True

    if 97 <= ascii_code <= 122:
        return True

    return False
```

9. Latent Analysis Functions

```
# latent_analysis.py
# By: Alex Lash
# Created: 1/31/13
# Last modified: 2/4/13

from numpy import *
from math import *
from requirement import *
from nltk.corpus import stopwords


def cos_sim(u,v):

    return u.dot(v) / (sqrt(u.dot(u)) * sqrt(v.dot(v)))

# ------------------------------------------------------------------------

def dsm(red_td_mat):

    num_words = len(red_td_mat)

    dsm = zeros((num_words,num_words))

    i = 0
    j = 0

    for row in red_td_mat:

        for other_row in red_td_mat:

            dsm[i,j] = cos_sim(row,other_row)

            j += 1

        i += 1
        j = 0

    return dsm

# ------------------------------------------------------------------------

def latent_keywords(red_td_mat, tokens, sim_val = .9):

    sim_mat = dsm(red_td_mat)

    i = 0

    keyword_dict = {}

    for token in tokens:

        row = sim_mat[i].tolist()

        keywords = []

        j = 0

        for cos_sim in row:

            if cos_sim >= sim_val and token != tokens[j]:
                keywords.append(tokens[j])
            j += 1
        keyword_dict[token] = keywords
        i += 1
    return keyword_dict
# ------------------------------------------------------------------------
def modify_dict(dictionary,words_to_keep):
    stops = set(stopwords.words('english'))
    for term in dictionary:
        definition = dictionary[term]
        dictionary[term] = [word for word in definition if word not in stops]
    keep_dict = {}
    for word in words_to_keep:
        last_char_ascii = ord(word[-1])
        if len(word)>1 and alpha(last_char_ascii):
            keep_dict[word] = dictionary[word]
    for term in keep_dict:
        definition = keep_dict[term]
        keep_items = []
        for item in definition:
            last_char_ascii = ord(item[-1])
            if len(item)>1 and alpha(last_char_ascii):
                keep_items.append(item)
        keep_dict[term] = keep_items
    return keep_dict
# ------------------------------------------------------------------------
def alpha(ascii_code):
    if 65 <= ascii_code <= 90:
        return True

    if 97 <= ascii_code <= 122:
        return True
    return False
```

10. Ontology Analysis Functions

```
# ontology_analysis.py

# By: Alex Lash
# Created: 1/31/13
# Last modified: 2/4/13

from nltk.corpus import wordnet
from requirement import Requirement

def synset_keywords(tokens,count_max=999999,p_of_s='n'):

    keyword_dict = {}

    for token in tokens:

        synset_list = []

        synsets = wordnet.synsets(token, pos=p_of_s)

        count = 0

        for synset in synsets:

            if count >= count_max:
                break

            # Get synonym (lemma) keywords for a synset
            lems = [lemma.name for lemma in synset.lemmas]

            keywords = []

            for lem in lems:

                split_lems = lem.split('_')

                for split_lem in split_lems:
                    keywords.append(split_lem)

##            # Get definition keywords for a synset
##            definition = synset.definition
##
##            def_obj = Requirement(definition)
##
##            if p_of_s == 'v':
##                def_toks = def_obj.vbs(w_tags = False)
##
##            else:
##                def_toks = def_obj.nns(w_tags = False)
##
##            keywords = keywords + def_toks
##
##            # Get example keywords for a synset
##            examples = synset.examples
```

```
##
##            for example in examples:
##
##                ex_obj = Requirement(example)
##
##                if p_of_s == 'v':
##                    ex_tokens = ex_obj.vbs(w_tags = False)
##
##                else:
##                    ex_tokens = ex_obj.nns(w_tags = False)
##
##                keywords = keywords + ex_tokens

            synset_list.append(keywords)

            count += 1

        keyword_dict[token] = synset_list

        print
        print keyword_dict[token]

    return keyword_dict
```

11. Semantic Mapping Functions

```
# semantic_mapping.py

# By: Alex Lash
# Created: 2/14/13
# Last modified: 2/14/13

from nltk.corpus import wordnet

def define_terms(keyword_dict, definition_dict,pos='n'):

    semantic_dict = {}

    for word in keyword_dict:

        keywords = keyword_dict[word]

        if keywords == []:

            max_index = 0

        else:

            possible_defs = definition_dict[word]

            if possible_defs == []:

                max_index = 0

            else:

                sim_val_lst = []
                for definition in possible_defs:

                    sim_val = max_word_sim(keywords,definition,pos)

                    sim_val_lst.append(sim_val)

                max_index = sim_val_lst.index(max(sim_val_lst))

        try:
            semantic_dict[word] = wordnet.synsets(word,pos)[max_index]

        except IndexError:
            semantic_dict[word] = None

    return semantic_dict

# ------------------------------------------------------------------------

def max_word_sim(text,other_text,pos='n'):

    sim_val_lst = []

    for word in text:

        try:
            synset = wordnet.synsets(word,pos)[0]

        except IndexError:
            synset = word

        for other_word in other_text:

            try:
                other_synset = wordnet.synsets(other_word,pos)[0]

            except IndexError:
                other_synset = other_word

            if type(synset) == type(other_synset):

                if synset == other_synset:
                    sim_val_lst.append(1.0)

                elif type(synset) != str:

                    wup_sim = synset.wup_similarity(other_synset)

                    sim_val_lst.append(wup_sim)

                else:
                    sim_val_lst.append(0.0)

            else:
                sim_val_lst.append(0.0)

    max_sim = max(sim_val_lst)

    return max_sim

# ------------------------------------------------------------------------
```

```
def mapping(word_lst_by_req,dictionary):

    semantic_lst_by_req = []

    for word_lst in word_lst_by_req:

        semantic_lst = []

        for word in word_lst:

            if type(dictionary[word]) == type(None):
                semantic_lst.append(word)

            else:
                semantic_lst.append(dictionary[word])

        semantic_lst_by_req.append(semantic_lst)

    return semantic_lst_by_req
```

12. Compare Requirements Functions

```
# compare.py

# By: Alex Lash
# Created: 2/14/13
# Last modified: 2/19/13

from numpy import *
from math import *
from latent_analysis import cos_sim
from nltk.corpus import wordnet

# ------------------------------------------------------------------------

def semantic_sim_sum(term_lst,other_term_lst,term_cutoff=.5):

    sim_val_lst = []

    for term in term_lst:

        for other_term in other_term_lst:

            if type(term) == str or type(other_term) == str:

                if type(term) == type(other_term):

                    if term == other_term:
                        sim_val_lst.append(1.0)

                    else:
                        sim_val_lst.append(0.0)

                else:
                    sim_val_lst.append(0.0)

            else:
                wup_sim = term.wup_similarity(other_term)

                if type(wup_sim) == type(None):

                    wup_sim = 0.0

                sim_val_lst.append(wup_sim)

    sum_sim = 0

    for val in sim_val_lst:

        if val >= term_cutoff:
            sum_sim += val

        else:
            sum_sim -= val

##    sum_sim = sum(sum_lst)

    return sum_sim

# ------------------------------------------------------------------------

def ontology_dsm(term_lst_by_req,to_file=True,node_edge=False,name=""):

    dsm = []

    for term_lst in term_lst_by_req:

        dsm_row = []

        for other_term_lst in term_lst_by_req:

            if term_lst == [] or other_term_lst == []:
                sim_val = 0.0

            else:
                sim_val = semantic_sim_sum(term_lst,other_term_lst)

            dsm_row.append(sim_val)

        append_row = []

        for val in dsm_row:

            if val >= .8:
                append_row.append(val)

            else:
                append_row.append(0.0)

        dsm.append(append_row)

    if to_file:
        dsm_file(dsm,name)

    if node_edge:
        node_file(dsm,name)

        edge_file(dsm,name)

    return array(dsm)

# ------------------------------------------------------------------------

def max_semantic_sim(term_lst,other_term_lst):

    sim_val_lst = []

    for term in term_lst:

        for other_term in other_term_lst:

            if type(term) == str or type(other_term) == str:

                if type(term) == type(other_term):

                    if term == other_term:
                        sim_val_lst.append(1.0)

                    else:
                        sim_val_lst.append(0.0)

                else:
                    sim_val_lst.append(0.0)

            else:

                wup_sim = term.wup_similarity(other_term)

                if type(wup_sim) == type(None):

                    wup_sim = 0.0

                sim_val_lst.append(wup_sim)

    max_sim = max(sim_val_lst)

    return max_sim

# ------------------------------------------------------------------------

def semantic_cos_sim(term_lst,other_term_lst):

    words = term_lst + other_term_lst

    words = list(set(words))

    term_vector = []
    other_term_vector = []

    for word in words:

        if __in(word,term_lst):
            term_vector.append(1.0)

        else:
            sim_val = max_semantic_sim(term_lst,[word])

            term_vector.append(sim_val)

        if __in(word,other_term_lst):
            other_term_vector.append(1.0)

        else:

            sim_val = max_semantic_sim(other_term_lst,[word])

            other_term_vector.append(sim_val)

    similarity = cos_sim(array(term_vector),array(other_term_vector))

    return similarity

# --------------------------------------------------------------------------

def __in(item,lst):

    try:
        return item in lst

    except AttributeError:
        return False

# --------------------------------------------------------------------------

def string_dsm(term_lst_by_req,to_file=True,node_edge=False,name=""):

    dsm = []

    for term_lst in term_lst_by_req:

        dsm_row = []

        for other_term_lst in term_lst_by_req:

            if term_lst == [] or other_term_lst == []:
                sim_val = 0.0

            else:
                sim_val = semantic_sim_sum(term_lst,other_term_lst)

            dsm_row.append(sim_val)

        append_row = []

        for val in dsm_row:

            if val >= .8:
                append_row.append(val)

            else:
                append_row.append(0.0)

        dsm.append(append_row)
    if to_file:
        dsm_file(dsm,name)

    if node_edge:

        node_file(dsm,name)

        edge_file(dsm,name)

    return array(dsm)

# ------------------------------------------------------------------------

def lsa_dsm(td_mat, to_file=True, node_edge=False, name=""):

    dt_mat = td_mat.T

    dsm = []

    for doc_row in dt_mat:

        dsm_row = []

        for other_doc_row in dt_mat:

            sim_val = cos_sim(doc_row,other_doc_row)

            dsm_row.append(sim_val)

        dsm.append(dsm_row)

    if to_file:
        dsm_file(dsm,name)

    if node_edge:

        node_file(dsm,name)

        edge_file(dsm,name)

    return array(dsm)

# ------------------------------------------------------------------------

def node_file(dsm, name):

    fname = name + "_nodes.csv"

    outfile = open(fname, 'w')

    outfile.write("Id;Label")

    num_reqs = len(dsm[0])

    for i in range(num_reqs):

        outfile.write("\n{0};R{0}".format(i+1))

    outfile.close()
```

```
# ------------------------------------------------------------------------
def edge_file(dsm, name):
    fname = name + "_edges.csv"
    outfile = open(fname, 'w')
    outfile.write("Source;Target;Type;Weight")
    num_reqs = len(dsm[0])
    i = 0
    for dsm_row in dsm:
        for j in range(i+1):

outfile.write("\n{0};{1};Undirected;{2}".format(i+1,j+1,dsm_row[j]))
        i += 1
    outfile.close()
# ------------------------------------------------------------------------
def dsm_file(dsm, name):
    fname = name + "_dsm.csv"
    outfile = open(fname, 'w')
    outfile.write("DSM")
    num_reqs = len(dsm[0])
    for i in range(num_reqs):
        outfile.write(";R{0}".format(i+1))
    outfile.write("\n")
    j = 1
    for row in dsm:
        outfile.write("R{0}".format(j))
        j += 1
        for item in row:
            outfile.write(";{0}".format(item))
        outfile.write("\n")
    outfile.close()
```

13. Requirement Class

```
# requirement.py

# By: Alex Lash
# Created: 1/31/13
# Last modified: 2/19/13

# Tag requirements and extract nouns and verbs.

from nltk.corpus import treebank
from nltk.tokenize import TreebankWordTokenizer
import pickle

## -----------------------------------------------------------------------

class Requirement:
    """[state methods here.]"""



    def __init__(self, req, pretagged=False, trained=True):

        if pretagged:

            tag_word_lst = req.split(' ')

            tagged_req = []

            for tag_word in tag_word_lst:

                tag_tuple = self.__to_tuple(tag_word)

                tagged_req.append(tag_tuple)

            self.tagged_req = tagged_req

        else:

            self.NL_req = req

            if not trained:
                self.__train_tagger()

            f = open('tagger.pickle', 'r')

            tagger = pickle.load(f)

            tokenizer = TreebankWordTokenizer()

            tokenized_req = tokenizer.tokenize(self.NL_req)

            self.tagged_req = tagger.tag(tokenized_req)

        self.nouns = self.__get_nouns()

        self.verbs = self.__get_verbs()
    def __to_tuple(self, word_str):

        """Reformats the string 'Word/tag' as the tuple (word,tag)"""

        rev_word_lst = word_str[::-1].split('/',1)[::-1]

        word_lst = []

        for word in rev_word_lst:
            word_lst.append(word[::-1])

        word_lst[0] = word_lst[0].lower()

        word_tup = tuple(word_lst)

        return word_tup



    def __train_tagger(self):

        from nltk.tag.sequential import ClassifierBasedPOSTagger
        from nltk.tag import DefaultTagger

        train_sents = treebank.tagged_sents()

        default = DefaultTagger('NN')

        tagger = ClassifierBasedPOSTagger(train=train_sents, backoff = default,
cutoff_prob = 0.3)

        f = open('tagger.pickle', 'w')

        pickle.dump(tagger, f)

        f.close()

        print "Tagger trained..."



    def tag_req(self, as_tuples = True):

        tagged_req = self.tagged_req

        req_wo_punct = []

        while tagged_req != []:

            token = tagged_req.pop(0)

            last_char_ascii = ord(token[0][-1])

            if self.__alpha(last_char_ascii):

                req_wo_punct.append(token)

        if not as_tuples:
            req_wo_punct = self.__delimit(req_wo_punct)
        return req_wo_punct


    def __delimit(self,tagged_req, delimiter="/"):

        delimited_tagged_req = []

        for tag_tuple in tagged_req:

            tag_str = self.__join(tag_tuple)

            delimited_tagged_req.append(tag_str)

        return delimited_tagged_req


    def __join(self, tup, char = '/'):

        return tup[0] + char + tup[1]


    def __extractor(self, tag_list):

        extracted_words = [word for word in self.tagged_req if (word[1] in
tag_list)]

        return extracted_words


    def nns(self, as_tuples = True, w_tags = True):

        if not w_tags:

            noun_lst = []

            for tag_tup in self.nouns:

                noun_lst.append(tag_tup[0])

            return noun_lst

        if not as_tuples and w_tags:
            return self.__delimit(self.nouns)

        return self.nouns


    def __get_nouns(self):

        nouns_lst = self.__extractor(['NN','NNS','NNP','NNPS'])

        req_nouns = []

        while nouns_lst != []:
```

```
        noun = nouns_lst.pop(0)

        last_char_ascii = ord(noun[0][-1])

        if self.__alpha(last_char_ascii):

            req_nouns.append(noun)

    return req_nouns


def __alpha(self, ascii_code):

    if 65 <= ascii_code <= 90:
        return True

    if 97 <= ascii_code <= 122:
        return True

    return False


def vbs(self, as_tuples = True, w_tags = True):

    if not w_tags:

        verb_lst = []

        for tag_tup in self.verbs:

            verb_lst.append(tag_tup[0])

        return verb_lst

    if not as_tuples and w_tags:
        return self.__delimit(self.verbs)

    return self.verbs


def __get_verbs(self):

    verbs_lst = self.__extractor(['MD','VB','VBD','VBG','VBN','VBP','VBZ'])

    req_verbs = []

    while verbs_lst != []:

        verb = verbs_lst.pop(0)

        last_char_ascii = ord(verb[0][-1])

        if self.__alpha(last_char_ascii):

            req_verbs.append(verb)
        return req_verbs
```

5.5 Appendix B: Accelerator Pedal Module Requirements

This appendix contains the full list of BMW accelerator pedal module requirements.

1. Surfaces and edges that may be touched during assembly or use may not exhibit sharp burrs.

2. Accelerator pedal module may not radiate disturbing mechanical noises (airborne sounds or mechanical vibrations).

3. Accelerator pedal module should not make noticeable sound when knocked.

4. Accelerator pedal module must fit the rest of the car.

5. Accelerator pedal module must use external voltage supply.

6. Each measuring channel is to be operated from its own voltage supply.

7. Pedal angle generated by the actuating force must be limited.

8. Pedal angle should be in the range of 16 degrees (+/- 0.5 degrees).

9. The sensor must be redundant with respect to output voltage.

10. The requirements of the output signals of the driving pedal module must be maintained at all operating temperatures and over the entire life span of the driving pedal module.

11. The requirements refer to an electrical wiring output.

12. If pedal breaks, the pedal must not be operational.

13. Pedal must not stick (by sticking or hooking).

14. Slope of response on the pedal (i.e., spring stiffness) must be within 0.8 N per degrees and 1.1 N per degrees.

15. Response force must be linear and within 2 N of the response line from required 5 from 0.5 degrees to 13.5 degrees.

16. If pedal breaks, the pedal must be recognizable as broken.

17. Accelerator pedal module must not send a signal to the electronic control unit against the driver’s wishes.

18. Pedal must not exhibit “stick slip effect” (instability in the force angle – response behavior).

19. Pedal must self-return.

20. Force angle must exhibit clear hysteresis from 0 degrees to 16 degrees.

21. The ascending force should be 30 percent (+6 N / -3 N) of the descending force.

22. Accelerator pedal module must mount to car.

23. Re-entry point should be scheduled within first 0.50 pedal angle.

24. Accelerator pedal module must maintain full functional ability over its expected life (5000 hours, from -40 C to 80 C).

5.6 Appendix C: Requirement Analysis Conceptual Models

This appendix contains all the conceptual models for every requirement analyzer and manual analysis.

14. Syntax Analyzer DSM

STR	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R2	0.00		1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00
R3	0.00	1.00		1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00
R4	0.00	1.00	1.00		1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00
R5	0.00	1.00	1.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00
R6	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R7	0.00	0.00	0.00	0.00	0.00	0.00		1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00
R8	0.00	0.00	0.00	0.00	0.00	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00
R9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R12	0.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00		1.00	0.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00
R13	0.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00		0.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00
R14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
R16	0.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00		1.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00
R17	0.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00		1.00	1.00	0.00	0.00	1.00	0.00	1.00
R18	0.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00		1.00	0.00	0.00	1.00	0.00	1.00
R19	0.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00		0.00	0.00	1.00	0.00	1.00
R20	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	1.00	0.00
R21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00
R22	0.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00		0.00	1.00
R23	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00		0.00
R24	0.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00	0.00	

15. LSA Analyzer DSM

LSA	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■	0.19	0.07	-0.08	0.18	0.04	0.07	0.06	0.15	0.03	0.02	0.18	0.13	0.04	0.04	0.08	0.07	0.14	-0.03	0.11	0.04	-0.08	0.08	0.01
R2	0.19	■	0.56	0.35	0.44	0.01	0.02	0.01	-0.10	0.12	-0.07	0.15	0.31	0.06	0.01	0.01	0.12	0.14	0.21	-0.03	0.01	0.51	0.04	0.09
R3	0.07	0.56	■	0.63	0.58	-0.05	0.27	0.44	0.01	0.14	-0.14	0.40	0.26	-0.06	-0.01	0.30	0.54	0.01	0.41	0.03	0.12	0.74	0.57	0.25
R4	-0.08	0.35	0.63	■	0.73	0.02	0.58	0.27	0.55	0.62	0.41	0.67	0.57	0.31	0.19	0.56	0.77	0.37	0.77	0.06	0.19	0.92	0.20	0.35
R5	0.18	0.44	0.58	0.73	■	0.56	0.37	0.18	0.56	0.33	0.09	0.52	0.48	0.19	0.07	0.41	0.59	0.23	0.63	0.07	-0.08	0.81	0.20	0.42
R6	0.04	0.01	-0.05	0.02	0.56	■	0.13	0.05	0.55	0.02	0.03	0.05	-0.03	0.03	0.04	0.05	-0.02	-0.02	0.03	0.02	0.05	0.04	0.08	0.07
R7	0.07	0.02	0.27	0.58	0.37	0.13	■	0.56	0.64	0.43	0.32	0.74	0.67	0.30	0.33	0.66	0.26	0.77	0.71	0.48	0.45	0.41	0.56	0.12
R8	0.06	0.01	0.44	0.27	0.18	0.05	0.56	■	0.24	0.24	0.09	0.25	0.00	0.39	0.39	0.22	0.12	0.17	0.27	0.70	0.31	0.21	0.75	0.02
R9	0.15	-0.10	0.01	0.55	0.56	0.55	0.64	0.24	■	0.66	0.64	0.63	0.32	0.34	0.31	0.60	0.31	0.33	0.50	0.05	0.29	0.31	0.19	0.04
R10	0.03	0.12	0.14	0.62	0.33	0.02	0.43	0.24	0.66	■	0.90	0.38	0.14	0.17	0.12	0.33	0.26	0.17	0.37	0.05	0.15	0.40	0.11	0.17
R11	0.02	-0.07	-0.14	0.41	0.09	0.03	0.32	0.09	0.64	0.90	■	0.15	-0.02	0.10	0.12	0.11	0.24	0.18	0.08	0.03	0.28	0.09	-0.13	-0.08
R12	0.18	0.15	0.40	0.67	0.52	0.05	0.74	0.25	0.63	0.38	0.15	■	0.74	0.34	0.19	0.97	0.34	0.44	0.87	-0.05	0.16	0.59	0.50	0.12
R13	0.13	0.31	0.26	0.57	0.48	-0.03	0.67	0.00	0.32	0.14	-0.02	0.74	■	0.18	0.05	0.62	0.35	0.82	0.85	0.21	-0.05	0.60	0.16	0.17
R14	0.04	0.06	-0.06	0.31	0.19	0.03	0.30	0.39	0.34	0.17	0.10	0.34	0.18	■	0.41	0.32	0.14	0.15	0.40	0.32	0.27	0.22	0.03	0.06
R15	0.04	0.01	-0.01	0.19	0.07	0.04	0.33	0.39	0.31	0.12	0.12	0.19	0.05	0.41	■	0.18	0.09	0.25	0.15	0.30	0.25	0.08	0.01	0.03
R16	0.08	0.01	0.30	0.56	0.41	0.05	0.66	0.22	0.60	0.33	0.11	0.97	0.62	0.32	0.18	■	0.19	0.30	0.82	-0.15	0.12	0.48	0.52	0.10
R17	0.07	0.12	0.54	0.77	0.59	-0.02	0.26	0.12	0.31	0.26	0.24	0.34	0.35	0.14	0.09	0.19	■	0.23	0.41	0.06	0.11	0.68	-0.05	0.12
R18	0.14	0.14	0.01	0.37	0.23	-0.02	0.77	0.17	0.33	0.17	0.18	0.44	0.82	0.15	0.25	0.30	0.23	■	0.56	0.55	0.20	0.29	0.07	0.07
R19	-0.03	0.21	0.41	0.77	0.63	0.03	0.71	0.27	0.50	0.37	0.08	0.87	0.85	0.40	0.15	0.82	0.41	0.56	■	0.19	-0.06	0.79	0.38	0.35
R20	0.11	-0.03	0.03	0.06	0.07	0.02	0.48	0.70	0.05	0.05	0.03	-0.05	0.21	0.32	0.30	-0.15	0.06	0.55	0.19	■	0.06	0.06	0.29	0.05
R21	0.04	0.01	0.12	0.19	-0.08	0.05	0.45	0.31	0.29	0.15	0.28	0.16	-0.05	0.27	0.25	0.12	0.11	0.20	-0.06	0.06	■	-0.06	0.20	0.01
R22	-0.08	0.51	0.74	0.92	0.81	0.04	0.41	0.21	0.31	0.40	0.09	0.59	0.60	0.22	0.08	0.48	0.68	0.29	0.79	0.06	-0.06	■	0.23	0.49
R23	0.08	0.04	0.57	0.20	0.20	0.08	0.56	0.75	0.19	0.11	-0.13	0.50	0.16	0.03	0.01	0.52	-0.05	0.07	0.38	0.29	0.20	0.23	■	0.04
R24	0.01	0.09	0.25	0.35	0.42	0.07	0.12	0.02	0.04	0.17	-0.08	0.12	0.17	0.06	0.03	0.10	0.12	0.07	0.35	0.05	0.01	0.49	0.04	■

16. Component LSA Analyzer DSM

LSA	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R2	0.00	■	0.79	0.50	0.54	0.00	0.00	0.00	0.00	0.00	0.00	0.76	0.76	0.00	0.00	0.76	0.55	0.50	0.76	0.00	0.00	0.76	0.00	0.49
R3	0.00	0.79	■	0.81	0.83	0.02	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.83	0.77	1.00	0.00	0.00	1.00	0.00	0.74
R4	0.00	0.50	0.81	■	0.67	0.28	0.00	0.00	-0.01	0.00	0.00	0.79	0.79	-0.04	0.00	0.79	0.47	0.59	0.79	-0.04	0.00	0.79	0.04	0.51
R5	0.00	0.54	0.83	0.67	■	0.07	0.00	0.00	0.02	0.00	0.00	0.84	0.84	0.01	0.00	0.84	0.87	0.48	0.84	0.06	0.00	0.84	-0.06	0.51
R6	0.00	0.00	0.02	0.28	0.07	■	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.03	0.00	0.00	-0.26	-0.35	0.00	0.09	0.00	0.00	-0.09	0.23
R7	0.00	0.00	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.87	0.00	0.00	0.87	0.00
R8	0.00	0.00	0.00	0.00	0.00	0.00	1.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.87	0.00	0.00	0.87	0.00
R9	0.00	0.00	0.00	-0.01	0.02	0.02	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-0.02	0.01	0.00	-0.01	0.00	0.00	0.01	0.00
R10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R12	0.00	0.76	1.00	0.79	0.84	0.00	0.00	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	1.00	0.85	0.78	1.00	0.00	0.00	1.00	0.00	0.76
R13	0.00	0.76	1.00	0.79	0.84	0.00	0.00	0.00	0.00	0.00	0.00	1.00	■	0.00	0.00	1.00	0.85	0.78	1.00	0.00	0.00	1.00	0.00	0.76
R14	0.00	0.00	0.00	-0.04	0.01	0.42	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00	0.01	0.05	0.00	-0.01	0.00	0.00	0.01	-0.03
R15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
R16	0.00	0.76	1.00	0.79	0.84	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	■	0.85	0.78	1.00	0.00	0.00	1.00	0.00	0.76
R17	0.00	0.55	0.83	0.47	0.87	-0.26	0.00	0.00	-0.02	0.00	0.00	0.85	0.85	0.01	0.00	0.85	■	0.64	0.85	-0.06	0.00	0.85	0.06	0.62
R18	0.00	0.50	0.77	0.59	0.48	-0.35	0.00	0.00	0.01	0.00	0.00	0.78	0.78	0.05	0.00	0.78	0.64	■	0.78	0.03	0.00	0.78	-0.03	0.52
R19	0.00	0.76	1.00	0.79	0.84	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.85	0.78	■	0.00	0.00	1.00	0.00	0.76
R20	0.00	0.00	0.00	-0.04	0.06	0.09	0.87	0.87	-0.01	0.00	0.00	0.00	0.00	-0.01	0.00	0.00	-0.06	0.03	0.00	■	0.00	0.00	0.50	0.00
R21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00	0.00
R22	0.00	0.76	1.00	0.79	0.84	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.85	0.78	1.00	0.00	0.00	■	0.00	0.76
R23	0.00	0.00	0.00	0.04	-0.06	-0.09	0.87	0.87	0.01	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.06	-0.03	0.00	0.50	0.00	0.00	■	0.00
R24	0.00	0.49	0.74	0.51	0.51	0.23	0.00	0.00	0.00	0.00	0.00	0.76	0.76	-0.03	0.00	0.76	0.62	0.52	0.76	0.00	0.00	0.76	0.00	■

17. Semantic Analyzer DSM

Sem	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.90	0.00
R2	0.00	■	0.00	0.00	1.34	0.00	0.00	0.00	0.00	0.00	0.00	0.88	0.88	0.00	0.00	0.88	0.00	0.00	0.88	0.00	0.00	0.88	0.00	0.00
R3	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.89	0.89	0.00	0.00	0.89	0.00	0.00	0.89	0.00	0.00	0.89	0.00	0.00
R4	0.00	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.88	0.88	0.00	0.00	0.88	0.00	0.00	0.88	0.00	0.00	0.88	0.00	0.00
R5	0.00	1.34	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.88	0.88	0.00	0.00	0.88	0.00	0.00	0.88	0.00	0.00	0.88	0.00	0.00
R6	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R7	0.00	0.00	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.50	0.00
R8	0.00	0.00	0.00	0.00	0.00	0.00	1.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.50	0.00
R9	1.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R12	0.00	0.88	0.89	0.88	0.88	0.00	0.00	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	1.00	0.86	0.00	1.00	0.00	0.00	1.00	0.00	0.87
R13	0.00	0.88	0.89	0.88	0.88	0.00	0.00	0.00	0.00	0.00	0.00	1.00	■	0.00	0.00	1.00	0.86	0.00	1.00	0.00	0.00	1.00	0.00	0.87
R14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
R16	0.00	0.88	0.89	0.88	0.88	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	■	0.86	0.00	1.00	0.00	0.00	1.00	0.00	0.87
R17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.86	0.86	0.00	0.00	0.86	■	0.00	0.86	0.00	0.00	0.86	0.00	0.00
R18	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	0.00
R19	0.00	0.88	0.89	0.88	0.88	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.86	0.00	■	0.00	0.00	1.00	0.00	0.87
R20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00	1.03	0.00
R21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00	0.00
R22	0.00	0.88	0.89	0.88	0.88	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.86	0.00	1.00	0.00	0.00	■	0.00	0.87
R23	0.90	0.00	0.00	0.00	0.00	0.00	1.50	1.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.03	0.00	0.00	■	0.00
R24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.87	0.87	0.00	0.00	0.87	0.00	0.00	0.87	0.00	0.00	0.87	0.00	■

18. Individual Engineer 1 DSM

E1	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R2	0.00		1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R3	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R4	0.00	0.00	0.00		0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
R5	0.00	0.00	0.00	0.00		1.00	0.00	0.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R6	0.00	0.00	0.00	0.00	1.00		0.00	0.00	1.00	1.00	1.00	0.00	0.00	0.03	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R7	0.00	0.00	0.00	1.00	0.00	0.00		1.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	1.00	1.00
R8	0.00	0.00	0.00	1.00	0.00	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00
R9	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00		1.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R10	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00		1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
R11	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00		0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
R13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00
R14	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00		1.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00
R15	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00		0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00
R16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
R17	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00
R18	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	0.00	0.00		0.00	1.00	1.00	0.00	0.00	1.00
R19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	1.00	0.00	0.00	1.00
R20	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00		1.00	0.00	0.00	0.00
R21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	1.00	1.00		0.00	0.00	0.00
R22	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00
R23	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00
R24	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	

19. Individual Engineer 2 DSM

E2	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R2	0.00		1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R3	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R4	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
R5	0.00	0.00	0.00	0.00		1.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R6	0.00	0.00	0.00	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R7	0.00	0.00	0.00	0.00	0.00	0.00		1.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R8	0.00	0.00	0.00	0.00	0.00	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R9	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00		0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R10	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00
R11	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00		0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R13	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00		0.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	0.00
R16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00
R18	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00		0.00	0.00	1.00	0.00	0.00	0.00
R19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00
R20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00		1.00	0.00	0.00	0.00
R21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	1.00		0.00	0.00	0.00
R22	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00
R23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00
R24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	

20. Individual Engineer 3 DSM

E3	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
R2	0.00		1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R3	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R4	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
R5	0.00	0.00	0.00	0.00		1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R6	0.00	0.00	0.00	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R7	0.00	0.00	0.00	0.00	0.00	0.00		1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R8	0.00	0.00	0.00	0.00	0.00	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R9	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
R11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00
R14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
R15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00	0.00
R18	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00	0.00
R19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00	0.00	0.00	0.00
R20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00		1.00	0.00	1.00	0.00
R21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00		0.00	0.00	0.00
R22	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		0.00	0.00
R23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00		0.00
R24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	

21. Manual Collaborative Study DSM

COL	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24
R1	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R2	0.00	■	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R3	0.00	1.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R4	0.00	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
R5	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R6	0.00	0.00	0.00	0.00	1.00	■	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R7	0.00	0.00	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	0.00
R8	0.00	0.00	0.00	0.00	0.00	0.00	1.00	■	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00
R9	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	■	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
R11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	■	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
R12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	1.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
R13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	■	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00
R14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	■	1.00	0.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	0.00
R15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	■	0.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	0.00
R16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	■	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
R17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	■	0.00	0.00	0.00	0.00	0.00	0.00	1.00
R18	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	■	1.00	1.00	1.00	0.00	0.00	0.00
R19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	■	1.00	0.00	0.00	0.00	1.00
R20	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	1.00	■	1.00	0.00	0.00	1.00
R21	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	1.00	0.00	1.00	■	0.00	0.00	0.00
R22	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	0.00	0.00
R23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	■	0.00
R24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	1.00	1.00	0.00	0.00	0.00	■