

ABSTRACT

Title of thesis: **FLOOR (Framework for Linking Ontology Objects and Textual Requirements): A New Requirements Engineering Tool that Provides Real-time Feedback**

Edward Zontek-Carney, Master of Science, 2017

Thesis directed by: Associate Professor Mark Austin
Department of Civil and Environmental Engineering
and Institute for Systems Research

Cost overruns on complex system-of-systems development programs frequently trace back to problems with requirements. For increasingly complex systems, a key capability is the identification and management of requirements early in a system's life cycle, when errors are cheapest and easiest to correct. Significant work has been done to apply natural language processing (NLP) to the domain of requirements engineering. Recently, requirements engineering tools have been developed that use NLP to leverage both domain ontologies and requirement templates, which define acceptable sentence structures for requirements. Domain ontologies provide terminology consistency, and enable rule-checking during the testing of requirements. This thesis introduces FLOOR, a new software tool for requirements engineering that leverages NLP. FLOOR not only integrates domain ontologies and requirement templates, but also supports importing multiple external domain ontologies.

FLOOR (Framework for Linking Ontology Objects and Textual
Requirements): A New Requirements Engineering Tool that
Provides Real-time Feedback

by

Edward Zontek-Carney

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science in Systems
Engineering
2017

Advisory Committee:
Professor Mark A. Austin, Chair
Professor A. Yavuz Oruc
Professor Huan Xu

© Copyright by
Edward Zontek-Carney
2017

Dedication

To Millie Jane Zontek-Carney.

Acknowledgments

I would like to acknowledge my parents, Kelly and Ed, my sister, Bonnie, and my wife, Amy, for their long-lasting love and support.

Table of Contents

List of Figures	vi
List of Abbreviations	viii
1 The Need for Model-based Systems Engineering	1
1.1 Problem Statement	1
1.1.1 What is Model-based Systems Engineering	1
1.1.2 State-of-the-Art Model-based Systems Engineering	2
1.2 Project Objectives	3
1.3 Contributions and Organization	5
2 Related Work	7
2.1 Natural Language Processing	7
2.1.1 Natural Language Processing Techniques	8
2.1.2 Natural Language Processing Tools	12
2.2 Requirements Engineering	12
2.2.1 Requirement Templates	13
2.2.2 Ontologies	14
2.2.3 Requirements Engineering Tools	18
3 FLOOR Software Architecture	20
3.1 FLOOR Overview	20
3.2 Class Hierarchy	21
3.3 NLP Libraries	23
4 Requirements Engineering with FLOOR	24
4.1 Working with FLOOR	24
4.1.1 Loading Existing Files	24
4.1.2 Requirement Template Matching	25
4.1.3 Ontology Term Matching	25
4.1.4 Generating Analysis Reports	26
4.1.5 Exporting Requirements	27

5	Case Study Problems	29
5.1	Case Study 1: Simple Requirement Template Matching	29
5.1.1	Creating and Printing Requirements and Requirement Templates	29
5.1.2	Tokenization and POS-Tagging	31
5.1.3	Matching Requirements with Requirement Templates	31
5.2	Case Study 2: Working with Requirements from NASA Goddard . . .	33
5.2.1	Import Data	33
5.2.2	Results	36
5.3	Case Study 3: Scalability Analysis	36
6	Conclusions and Future Work	38
6.1	Conclusions	38
6.2	Future Work	38
	Appendices	41
A	RichTextFX License Agreement	41
	Bibliography	43

List of Figures

1.1	Pillars of SysML: structure, behavior, requirements and parametrics.	3
1.2	Manual translation of text into high-level textual requirements.	4
1.3	Framework for automated transformation of text (documents) into textual requirements (semi-formal models).	4
2.1	Output from first step on building chunking grammar. Purpose: Simply pick nouns from test sentence.	10
2.2	Output from second step on building chunking grammar. Purpose: Identify noun phrases.	10
2.3	Output from third step on building chunking grammar. Purpose: Form noun phrases.	10
2.4	Output from fourth step on building chunking grammar. Purpose: Identify the adjective preceding the first noun phrase.	10
2.5	Example requirement template and instance.	13
2.6	Simple ontology, rules, and event-driven evolution of semantic graphs.	14
2.7	Schematic for state-of-the-art traceability and ontology-enabled traceability for system design and management.	16
2.8	Connecting textual requirements to semantic models of system structure and behavior.	17
3.1	Real-time feedback: user interface.	21
3.2	FLOOR: class diagram.	23
4.1	Import options on the FLOOR File Menu.	25
4.2	Real-time feedback: matching requirement templates and ontology terms.	26
4.3	Example Testability Report.	27
5.1	Simple template matching: Create and print requirements.	30
5.2	Simple template matching: Create and print templates.	30
5.3	Simple template matching: Tokenization and POS tagging.	31
5.4	Simple template matching: Textual requirements matched with templates.	32

6.1	FLOOR: Requirement template and domain ontology match.	39
-----	--	----

List of Abbreviations

API	Application Programming Interface
CSV	Comma Separated Values
DODT	Domain Ontology Design Tool
DOORS	Dynamic Object-Oriented Requirements System
FLOOR	Framework for Linking Ontology Objects and Textual Requirements
GUI	Graphical User Interfaces
INCOSE	International Council on Systems Engineering
MBSE	Model-Based Systems Engineering
NLP	Natural Language Processing
OWL	Web Ontology Language
POS	Part of Speech
SysML	System Modeling Language
UML	Unified Modeling Language
XML	Extensible Mark-up Language

Chapter 1: **The Need for Model-based Systems Engineering**

1.1 Problem Statement

This thesis describes a new approach to the interpretation, development, and analysis of textual requirements, through the use of application-specific ontologies and natural language processing. It builds upon our previous work in exploring ways in which model-based systems engineering might benefit from techniques in natural language processing [5, 6].

1.1.1 What is Model-based Systems Engineering

Model-based systems engineering (MBSE) is a system development approach in which the focus and primary artifacts of development are models, as opposed to documents [7, 9]. Maintaining a central system model supports error prevention, error correction, reuse, and team-based development. As systems of interest have become increasingly complex, a need has arisen for MBSE development tools with enhanced automation capabilities.

1.1.2 State-of-the-Art Model-based Systems Engineering

Most widely-used MBSE tools are focused on the development and decomposition of system architecture, as opposed to requirements. Modern MBSE tools (e.g., Rhapsody or MagicDraw) support the Systems Modeling Language (SysML), and provide a framework for reuse and collaboration based on the development of a single, central model. SysML does contain a Requirement Diagram (see Figure 1.1), which captures requirement text, and can link requirements to other system objects, such as test cases. Plug-ins for SysML-based tools exist (e.g. DataHub) that enable the integration of a requirements management database (e.g., DOORS) with the system model [12]. Even so, current MBSE tools do not address the process of requirements development and/or analysis. The underlying assumption is that the text of each requirement is determined by a process external to the MBSE development. No feedback addressing the quality of new requirements is presented to the user while creating a Requirement Diagram. The critical capability to fully develop requirements during early system life cycle phases, when it is cheapest and easiest to correct errors, is largely overlooked by modern MBSE tools.

While engineers are looking for semi-formal and formal models to work with, the reality remains that many large-scale projects begin with hundreds, or even thousands, of pages of textual requirements. Initial requirements sets may be inadequate due to incompleteness, ambiguity, and/or several other factors. State-of-the-art practice (see Figure 1.2) involves the manual translation and decomposition of text into a semi-formal format (suitable for representation in a requirements

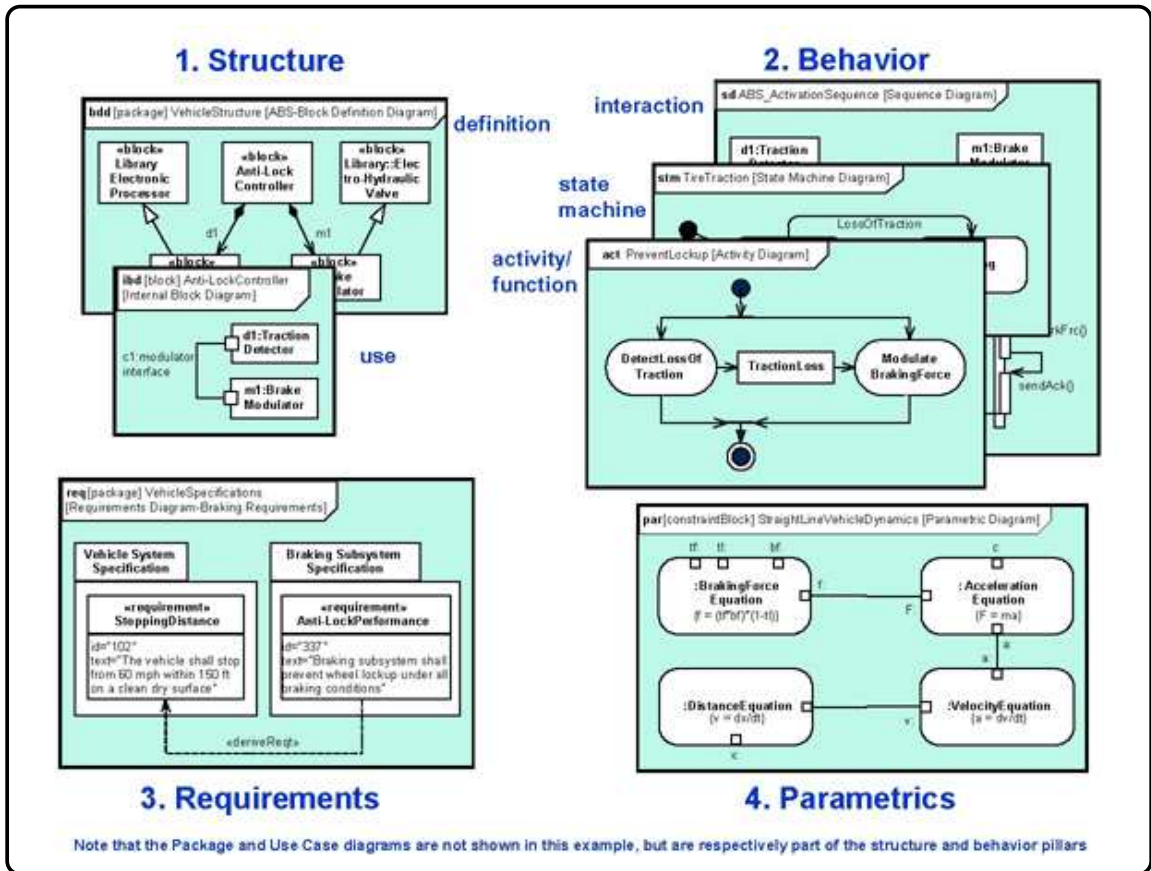


Figure 1.1: Pillars of SysML: structure, behavior, requirements and parametrics.

database) – a slow and error-prone process.

1.2 Project Objectives

This work is motivated by a strong need for computer processing tools that can help requirements engineers overcome and manage these challenges. During the past twenty years, significant work has been done to apply natural language processing (NLP) to the domain of requirements engineering [3, 28, 29]. Applications range from using NLP to extract ontologies from a requirements specification, to using NLP to verify the consistency and/or completeness of a requirements specification.

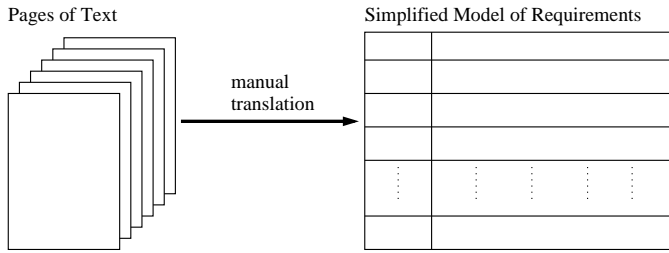


Figure 1.2: Manual translation of text into high-level textual requirements.

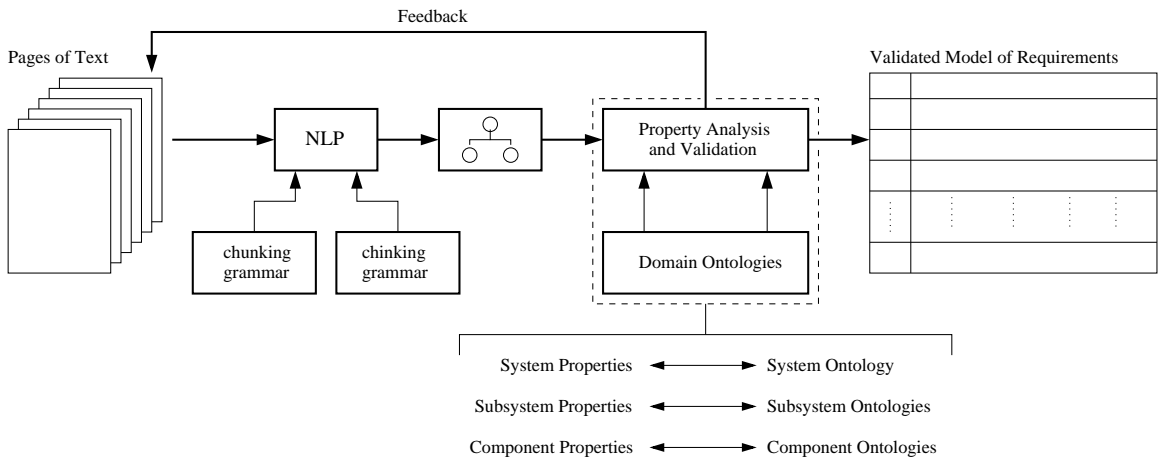


Figure 1.3: Framework for automated transformation of text (documents) into textual requirements (semi-formal models).

Our near-term research objectives are to use modern natural language processing (NLP) tools to ingest and tag a set of requirements, and to use the results to offer support to systems engineers during their task of further decomposing the initial requirements set. We propose applying NLP in two separate ways: requirement template matching, and ontology term matching. A requirement template is a predetermined sentence structure that is deemed suitable for use in writing requirements [20]. Our goal is to inform the author whether a requirement, at the time of writing, matches to a library of existing requirement templates. Leveraging NLP to enforce the use of requirement templates can increase the clarity and testability of requirements. An ontology is a set of concepts present in a particular domain, and

the relationships between them [8]. Our goal is to inform the author of a requirement, at the time of writing, whether the author's new requirement uses terminology that is consistent with an existing ontology. Using NLP to match terms against an existing ontology (or multiple ontologies) can address the completeness and ambiguity of a requirement set [5]. We aim to apply each method not only during real-time requirements development, but also during post-processing a requirement set, in the form of requirements analysis reports.

Figure 1.3 shows the framework for automated transformation of text (documents) into textual requirements (semi-formal models) described in this paper. NLP techniques are applied to textual requirements, and the analyzed text is then compared against a library of requirement templates, and a library of ontologies. Multiple ontologies can be used, perhaps for different levels of a system's hierarchy. Ontologies may be domain-specific, or interdisciplinary (e.g., an ontology of physical units).

1.3 Contributions and Organization

The contributions of this work are as follows:

1. A prototype software tool for requirements engineering that provides real-time feedback to the user regarding the quality of newly written requirements. Similar tools do exist, but we provide a novel implementation.
2. The capability to use multiple ontologies during requirements development. To our knowledge, this is a new accomplishment in the field of requirements

engineering.

- 3.** An analysis of the utility of the new tool based on a case study containing real requirements provided from an industry partner.

This thesis is organized as follows: Chapter 2 presents an overview of related work in the areas of natural language processing and requirements engineering. Chapter 3 describes the design and implementation of FLOOR. Chapter 4 walks through a typical use case for FLOOR, and provides an analysis of FLOOR's utility based on a case study of industry-provided requirements. Chapter 5 summarizes our contributions and suggests opportunities for future growth.

Chapter 2: **Related Work**

2.1 Natural Language Processing

Natural language processing (NLP) is a field of computer science and linguistics primarily focused on developing automated techniques for parsing and interpreting standard text. Since the 1980s, most NLP frameworks incorporate statistical and machine-learning methodologies to analyze textual corpora. Depending on the ultimate goal of the processing, an NLP sequence features different steps. For the purposes of term and sentence structure matching, a typical NLP workflow features the following steps: tokenization, part-of-speech tagging, and chunking. Tokenization is the deconstruction of text into individual elements, based on a predetermined set of delimiters. Often, the delimiters are simply a combination of white space and punctuation marks. Part-of-speech tagging (POS-tagging) ingests tokenized text as an input, and outputs the sequence of part-of-speech tags corresponding to each input token. Chunking uses the tags to determine whether adjacent tokens belong to the same phrase, or chunk.

2.1.1 Natural Language Processing Techniques

Tokenization. Tokenization is the deconstruction of text into individual elements, or tokens, based on a predetermined set of delimiters. An example delimiter is white space. Periods, commas, and other punctuation marks are frequently used as well [2]. In general, any character can be used as a delimiter between tokens. Tokenization is a well-understood problem, but challenges still exist. In English, for example, a tokenizer must address contractions, hyphenated words, and unusual symbols. Depending on the end goal of the NLP, different rules may be desirable for such corner cases.

Part-of-Speech Tagging. Part-of-speech tagging labels each individual token with its particular part-of-speech. Most modern POS-tagging algorithms rely on a model that is trained in advance on representative corpora. A POS-tag consists of one, two, or three characters – a label that corresponds to a specific part-of-speech. An example of a POS-tag is JJ, which the Penn Treebank Project uses for denoting an adjective [30]. In fact, the Penn Treebank tag-set has become the de-facto standard for POS-tagging. It consists of 48 POS-tags in total, including several representing punctuation marks (not typically thought of as parts-of-speech, but certainly valid pieces of text that must be addressed).

Chunking. Chunking is the process by which POS-tagged tokens are segmented and labeled into phrases, or chunks. As an example, consider the sentence: “Systems engineers shall work.” In this sentence, “systems engineers” is one chunk, – a

noun phrase. The other chunk, “shall work,” is a verb phrase. In order to accomplish chunking, a particular grammar can be defined [5]. More commonly, chunker algorithms rely on a trained model, similar to most POS-taggers.

Automatic Term Recognition and Automatic Indexing. Strategies for automatic term recognition and automatic indexing fall into the general area of computational linguistics [22]. Algorithms for single-term indexing date back to the 1950s, and for indexing two or more words to the 1970s [13]. Modern techniques for multi-word automatic term recognition are mostly empirical, and employ combinations of linguistic information (e.g., POS-tagging) and statistical information acquired from the frequency of usage of terms in candidate documents [4, 18]. The resulting terms can be useful in more complex tasks such as semantic search, question-answering, identification of technical terminology, automated construction of glossaries for a technical domain, and ontology construction [16, 21, 24].

A Simple Example. Consider the test sentence:

```
"When I work as a senior systems engineer, I truly enjoy my work."
```

Tokenizing the sentence gives:

```
[ ( 'When', 'WRB'), ('I', 'PRP'), ('work', 'VBP'), ('as', 'RB'), ('a', 'DT'),  
  ('senior', 'JJ'), ('systems', 'NNS'), ('engineer', 'NN'), (',', ','),  
  ('I', 'PRP'), ('truly', 'RB'), ('enjoy', 'VBP'), ('my', 'PRP$'),  
  ('work', 'NN'), ('.', '.') ]
```

The first thing to notice from the output is that the tags are two or three letter codes.

Each one represents a lexical category or part of speech. For instance, WRB stands for *Wh-adverb*, including *how*, *where*, *why*, etc. PRP stands for *Personal pronoun*;

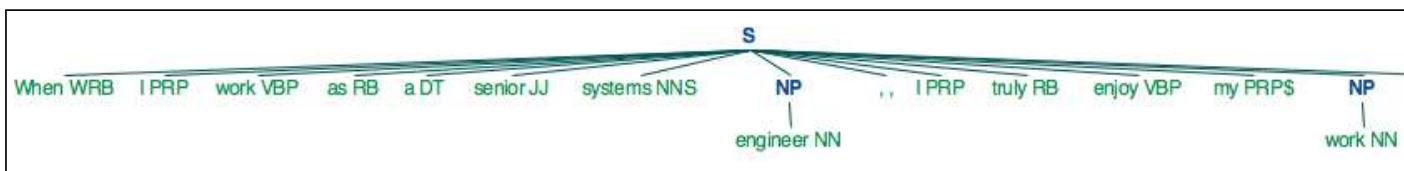


Figure 2.1: Output from first step on building chunking grammar. Purpose: Simply pick nouns from test sentence.

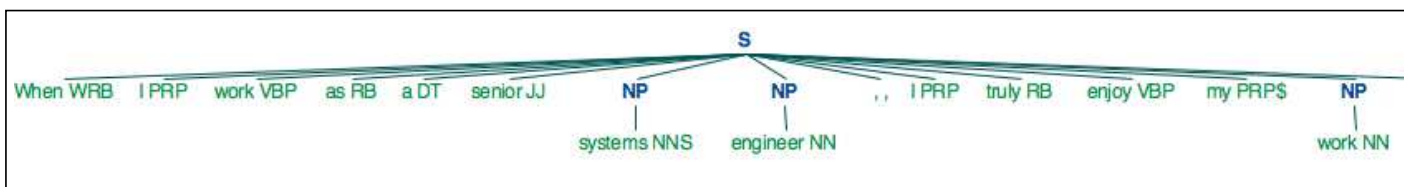


Figure 2.2: Output from second step on building chunking grammar. Purpose: Identify noun phrases.

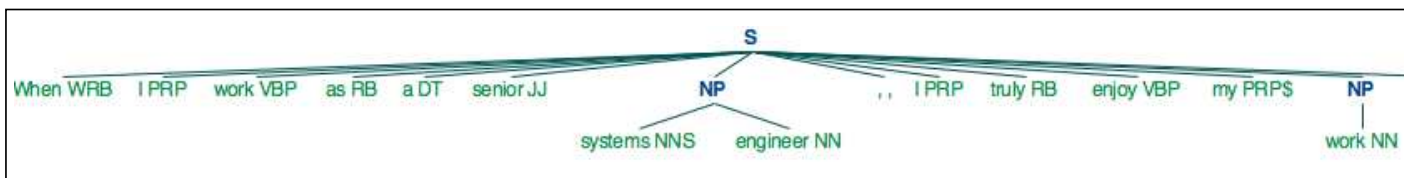


Figure 2.3: Output from third step on building chunking grammar. Purpose: Form noun phrases.

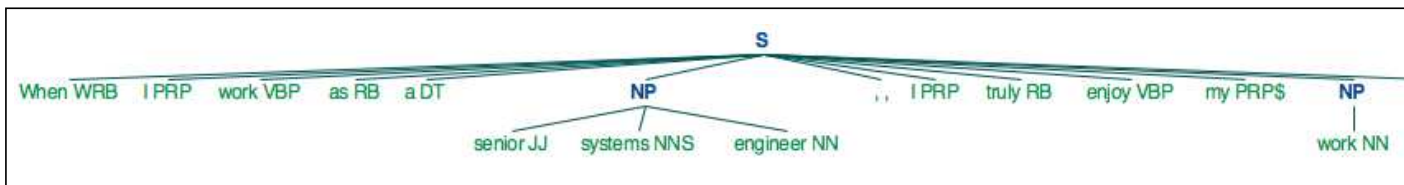


Figure 2.4: Output from fourth step on building chunking grammar. Purpose: Identify the adjective preceding the first noun phrase.

RB for *Adverb*; *JJ* for *Adjective*, *VBP* for *Present verb tense*, and so forth [30]. These categories are more detailed than presented in [19], but they can all be traced back to those ten major categories. It is important to note the possibility of one-to-many relationships between a word and the possible tags. For our test example, the word *work* is first classified as a verb, and then at the end of the sentence, is classified as a noun, as expected. Moreover, we found two nouns (i.e., objects), so we can affirm that the text is saying something about *systems*, *an engineer* and *a work*. But we know more than that. We are not only referring to *an engineer*, but to a *systems engineer*, and not only a *systems engineer*, but a *senior systems engineer*. This is our *entity* and we need to *recognize* it from the text. To do this, we need to somehow tag groups of words that represent an entity (e.g., sets of nouns that appear in succession: (*'systems'*, *'NNS'*), (*'engineer'*, *'NN'*)). Modern NLP tools offer regular expression processing support for identifying groups of tokens, specifically noun phrases, in the text.

Figures 2.1 through 2.4 illustrate the progressive refinement of our test sentence by the chunking parser. The purpose of the first pass is to simply pick the nouns from our test sentence. Figure 2.1 is a graphical representation of the results. Subsequent analyses identify the presences of plural nouns (NNS), form single noun phrases, and identify situations where words are located between adjectives and nouns. The latter steps identify two entities, *senior systems engineer* and *work*, and that is precisely what we want.

2.1.2 Natural Language Processing Tools

NLP has benefited greatly from the open-source era, as many prolific NLP packages are available on a variety of platforms. One popular NLP tool is General Architecture for Text Engineering (GATE), a Java-based NLP suite containing an integrated GUI. OpenNLP is another Java library for performing NLP, distributed by Apache. The Natural Language Toolkit (NLTK), written in Python is yet another mature NLP tool [26]. All of these tools contain libraries that support tokenization, POS-tagging, and chunking, as well as several other NLP functions. We have mentioned only a few packages of particular interest here, but for a more thorough survey of modern NLP tools, see [25].

2.2 Requirements Engineering

Requirements engineering is the process by which system requirements are created, decomposed, and maintained. Requirements engineering is a critical discipline for complex systems development, as failures in requirements can very easily have long-lasting, costly impacts on future system development. It is therefore critical that system requirements be written and decomposed effectively. The quality of requirements can be measured in many ways. According to the International Council on Systems Engineering (INCOSE), characteristics of a high-quality requirement set include, but are not limited to: completeness, containing requirements describing all desired capabilities; consistency, the absence of requirements that contradict one another; singularity, containing requirements that each describe exactly one capa-

bility; testability, containing requirements that can each be individually verified; and unambiguity, the absence of requirements that have multiple interpretations [27]. We contend that in providing feedback regarding requirement templates and ontology term matching, FLOOR assists systems engineers in writing requirements that achieve these five criteria.

2.2.1 Requirement Templates

Requirement templates, or boilerplates, were introduced by Dick, Hull, Jackson [20] in 2002. The concept is to maintain a repository of acceptable sentence structures to be used for writing requirements. Requirements can then be written in a clear and consistent manner, thereby improving singularity, testability, and unambiguity. An example of a requirement template and a corresponding requirement is given below in 2.5. In the example, the underlined instance phrases correspond to the angle-bracketed place-holders in the preceding requirement template. Requirement templates may be added as necessary, and many requirements may be written based on the same template.

Template:	The <system> shall <action> <condition>.
Instance:	The <u>cursor</u> shall <u>move</u> <u>upon mouse movement</u> .

Figure 2.5: Example requirement template and instance.

Requirement templates have been integrated into several NLP-based requirements engineering tools, most recently by DODT [14]. DODT's implementation supports the combination of multiple requirement templates when creating a new

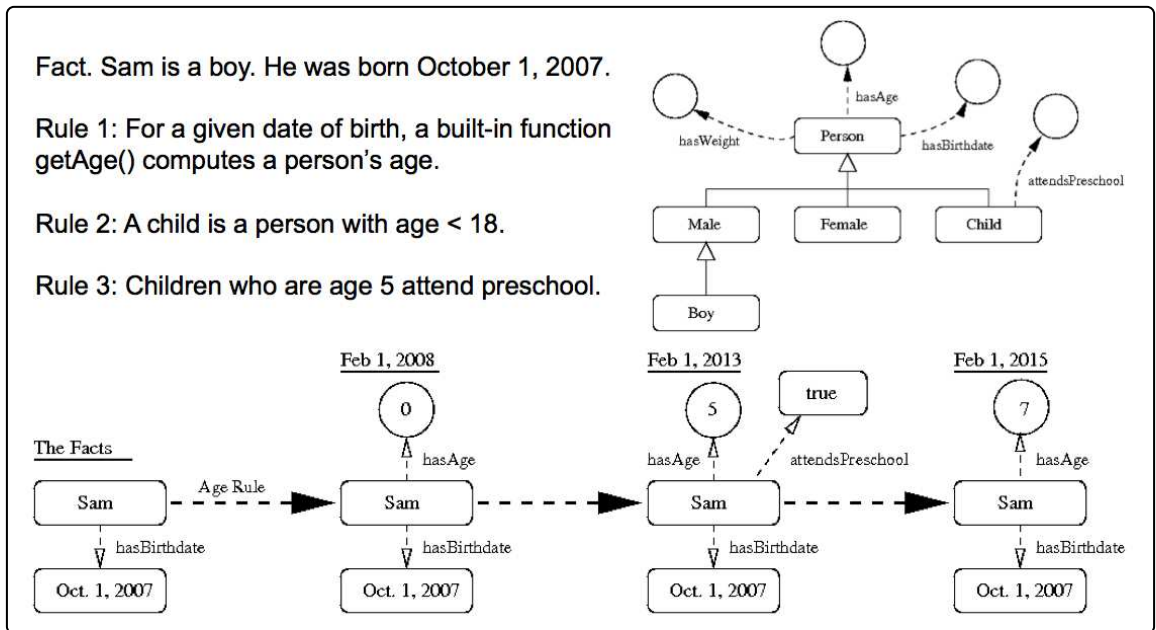


Figure 2.6: Simple ontology, rules, and event-driven evolution of semantic graphs.

requirement. This method keeps the number requirement templates relatively small, while still allowing for sufficient complexity. There is no standard format for requirement templates, and finding an existing set of requirement templates remains a challenge.

2.2.2 Ontologies

A domain ontology describes the concepts related to a specific domain, the relationships among those concepts, and the attributes of data needed to describe individuals (or instances) of the concepts. These notions are not unlike a class hierarchy and data attributes one finds in object-oriented design methods. Instances of ontologies are modeled as graphs that can be instantiated with data, and can respond - dynamically evolve - to external events. From a requirements engineering perspective, domain ontology integration is implemented by DODT [31].

Figure 2.6 shows, for example, the relationship among classes and properties in a simplified family ontology. A person has properties: `hasAge`, `hasWeight` and `hasBirthdate`. Male and Female are subclasses of the class Person and, as such, will inherit all of the properties associated with Person. Boy is a specialization of Male. A Child is a Person who may (or may not) attend Preschool. The upper left-hand side of Figure 2.6 shows one fact and three rules. Sam is a boy born on October 1, 2007. Given a birthdate and a current time, a built-in function `getAge()` computes Sams age. Further rules can be defined for when a person is child and when they attend preschool. Some of the data (e.g., Sams date of birth) remains constant over time. Other data is dynamic and is controlled by the family rules.

Pathway to Ontology-Enabled Traceability for System Design and Management. From a systems engineering standpoint, this simple scenario is appealing because it suggests an opportunity for modeling requirements, system structure, and system behavior with semantic graphs that dynamically evolve in response to events [8].

The systems architecture for state-of-the-art requirements traceability and its connection into the proposed model is shown in the upper and lower sections of Figure 2.7.

In state-of-the-art traceability mechanisms, design requirements are connected directly to design solutions (e.g. objects in the engineering model). Our contention is that even in the earliest stages of system development, a better approach is to develop requirements by asking the question: What concepts (or group of design

State-of-the-Art Traceability



Proposed Model for Ontology-Assisted Development of Requirements and Traceability

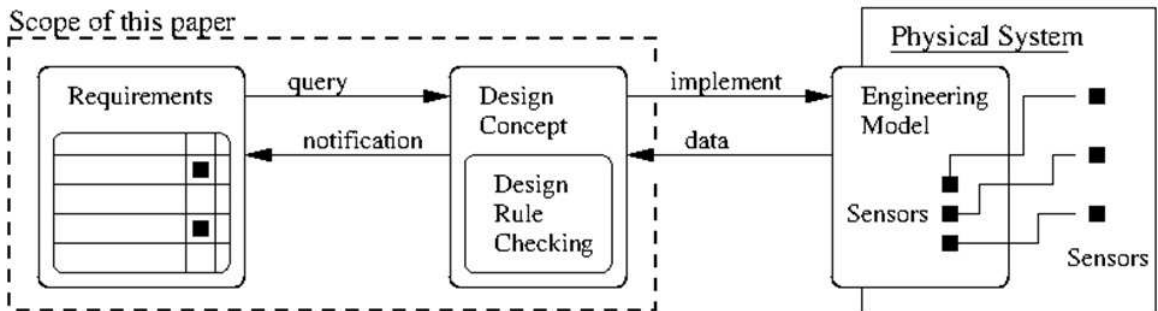


Figure 2.7: Schematic for state-of-the-art traceability and ontology-enabled traceability for system design and management.

concepts) will I need to apply to create (and later on, satisfy) a requirement? Design solutions are the instantiation/implementation of these concepts.

In the lower half of Figure 2.7, textual requirements, ontology models, and engineering models provide distinct views of design:

1. Requirements are a statement of what is required,
2. Engineering models - not within the scope of this paper - are a statement of how the required functionality and performance might be achieved, and
3. Ontologies and their associated rules are a statement of concepts justifying a tentative solution.

During design, mathematical and logical rules are derived from textual requirements, which in turn, are connected to elements in an engineering model. A key benefit of the proposed approach is that design rule checking can be applied at the earliest state

possible - as long as data is available for the evaluation of rules, rule checking can commence; the textual requirements and engineering models need not be complete [11]. During the system operation, traceability links enable the evaluation of cause-and-effect relationships between changes (events) at the system/component level and their effects on stakeholder requirements [10]. Present-day system methodologies and tools are not designed to handle projects in this way.

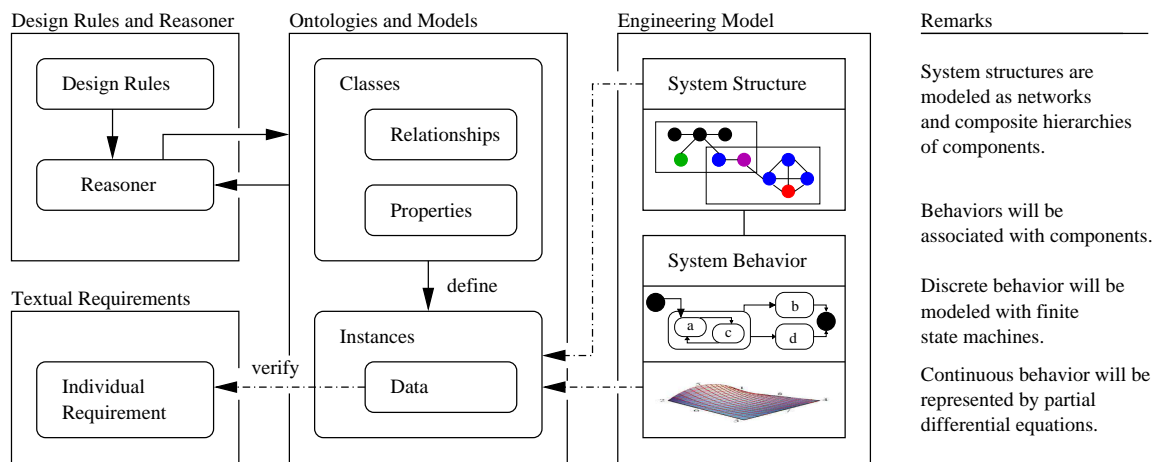


Figure 2.8: Connecting textual requirements to semantic models of system structure and behavior.

Figure 2.8 pulls together the different pieces of the proposed architecture shown in Figure 2.7. A subset of the textual requirements will be described in terms of mathematical and logical expressions for design rule checking. The pathway from engineering models of system structure and behavior back to individual requirements, with the data associated with ontology instances being used to verify whether a requirement is currently satisfied [11, 10].

Our contention is that in the earliest stages of system development, strategic approaches to the development of textual requirements will benefit from constant feedback on the relationship of concepts expressed in the text and the concepts,

data, and rules defined in the associated ontologies. Specifically, when writing requirements, using terminology taken directly from a domain ontology can greatly improve consistency.

2.2.3 Requirements Engineering Tools

The application of NLP to requirements engineering is not a new idea [17], [23], [29]. In fact, there are many commercially available requirements engineering tools that integrate NLP in some way. Some existing tools use NLP to analyze requirements for specific characteristics, like the presence of the word *shall*. Such tools include DESIRE, Qualicen Scout, and QVscribe. Other tools, such as RETA and Semios, support requirement templates as well. A few tools support both requirement templates and term matching - the two key features of FLOOR. These tools include Lexior, the Requirements Authoring Tool (RAT), and the Domain Ontology Design Tool (DODT). For a more thorough survey of NLP-based tools for requirements engineering, see [32].

Out of all available NLP-based requirements engineering tools, DODT is the most prolific. DODT uses NLP to leverage both requirement templates and domain ontologies, in a similar fashion to FLOOR [14], [15]. DODT has been utilized in an industry setting on “real” requirements, with promising results [31]. There are some key differences between DODT and FLOOR. First, DODT is a dual-purpose tool: it features editors for both requirements and domain ontologies (as the name suggests). There are advantages and disadvantages to this approach. One advantage

is direct access to the domain ontology while editing requirements, which allows for immediate insertion of terms into the domain ontology if deemed necessary. The downside is that the requirements and domain ontology become coupled potentially limiting the reusability of the domain ontology across separate development efforts. Also, DODT requires exactly one domain ontology. As we demonstrate in Chapter 4, FLOOR allows the requirements engineer to import multiple external domain ontologies, enabling extensibility and reusability across multiple domains involved in team development of complex engineering systems.

Chapter 3: **FLOOR Software Architecture**

3.1 FLOOR Overview

We began the design of FLOOR with the mission of providing real-time feedback to the user regarding a new requirement's applicability to requirement templates, and its terminological consistency with loaded ontologies. This goal led to two central questions during the development of FLOOR:

1. What is a logical methodology for using NLP to extract the information we need about requirement text?
2. What is a logical methodology displaying the feedback to the user?

The answer to the former is described in the following sections. As for displaying feedback, requirement templates and ontology terms each required a user interface decision. For displaying matching requirement templates, we chose a pop-up context menu. The pop-up automatically updates based on new text entered by the user. For displaying matching ontology terms, we elected to alter the font color of matching (or non-matching) terms. Matching terms take on a green font color, and non-matching terms assume a red font color. The font color for individual words also automatically based on new text entered by the user. Figure 3.1 illustrates the

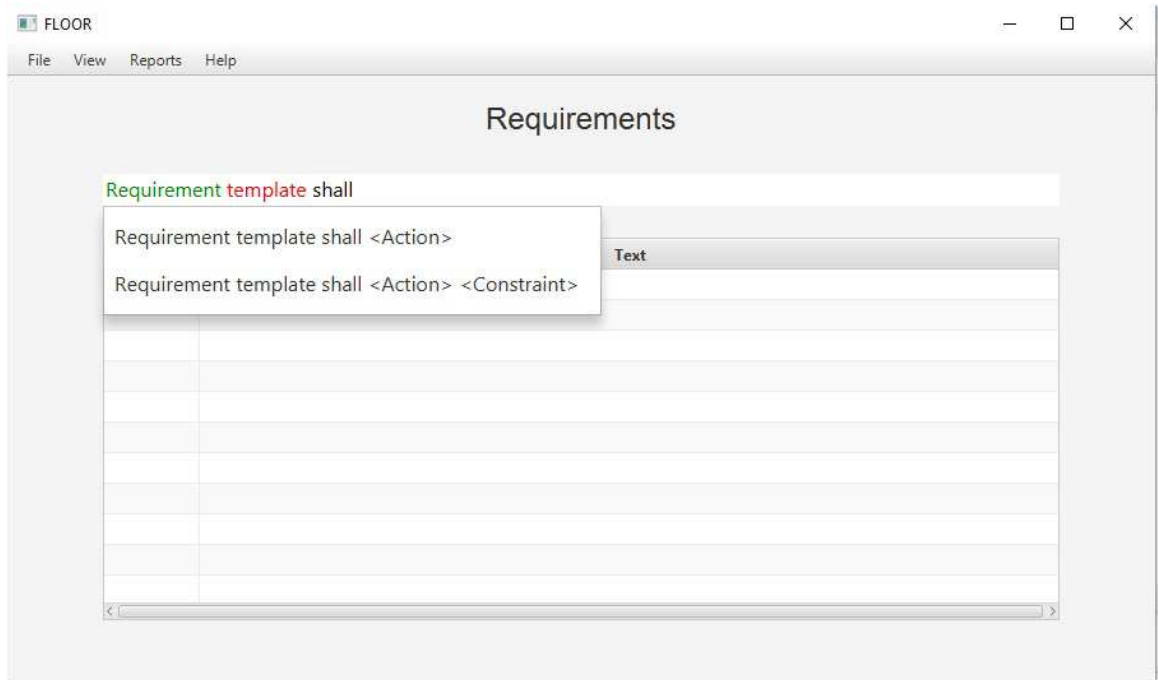


Figure 3.1: Real-time feedback: user interface.

various feedback elements. In Section 4.1, 4.2 figure illustrates the feedback user interfaces during a real example.

3.2 Class Hierarchy

FLOOR is written in Java, and runs as a standalone JavaFX application. From a structural perspective, the FLOOR project contains three packages: FLOOR, ReqTemps, and TemplateChunk. The FLOOR package contains all logic for controlling the GUI. The ReqTemps package, short for Requirements and Templates, contains a set of classes that serve as the data model for the objects operated on by the FLOOR package. TemplateChunk, contains enumerations referenced by templates instances. This layer is what enables FLOOR to interpret the NLP-processed requirement text, i.e., how sentence fragments are mapped to potential requirement

templates.

Before delving into the correlation of partial sentences and matching requirement templates, we must first understand the data model. ReqTemps contains a Requirement class and a Template class. Requirement has the properties “ID,” a string, and “text,” also a string. The Template class is more interesting - it contains a name, again just a string, and an ArrayList of type Attribute.

Figure 3.2 is a high-level class diagram depicting the FLOOR architecture. Attribute is an interface class that allows a Template to contain components that have different characteristics. Following that direction, AbstractAttribute is an abstract class that implements the Attribute interface. AbstractAttribute contains and AttributeType (enumeration) called “type,” and a string called “text.” Individual Attributes that extend AbstractAttribute include Condition, Article, Subsystem, Modal, Action, Entity, and Constraint. These are the elements of a Template.

A Condition represents a conditional phrase, like “upon mouse movement.” An Article is either “A,” “An,” or “The.” A Subsystem is the subject of a requirement. A Modal is either “Will,” “Must,” or “Shall.” An Action is the function called for by a requirement. An Entity is the object acted on by a requirements action. (“Object” is not used because it is a reserved keyword in Java.) A Constraint places a condition on an Action. These Attributes are the building blocks of a Template.

By construction, some Attributes of a Template may only originate from chunks of text with a certain type. For example, a Subsystem is always present in a noun phrase chunk. We enforce this in our implementation by providing public enumerations for each AttributeType. These enumerations are contained in sepa-

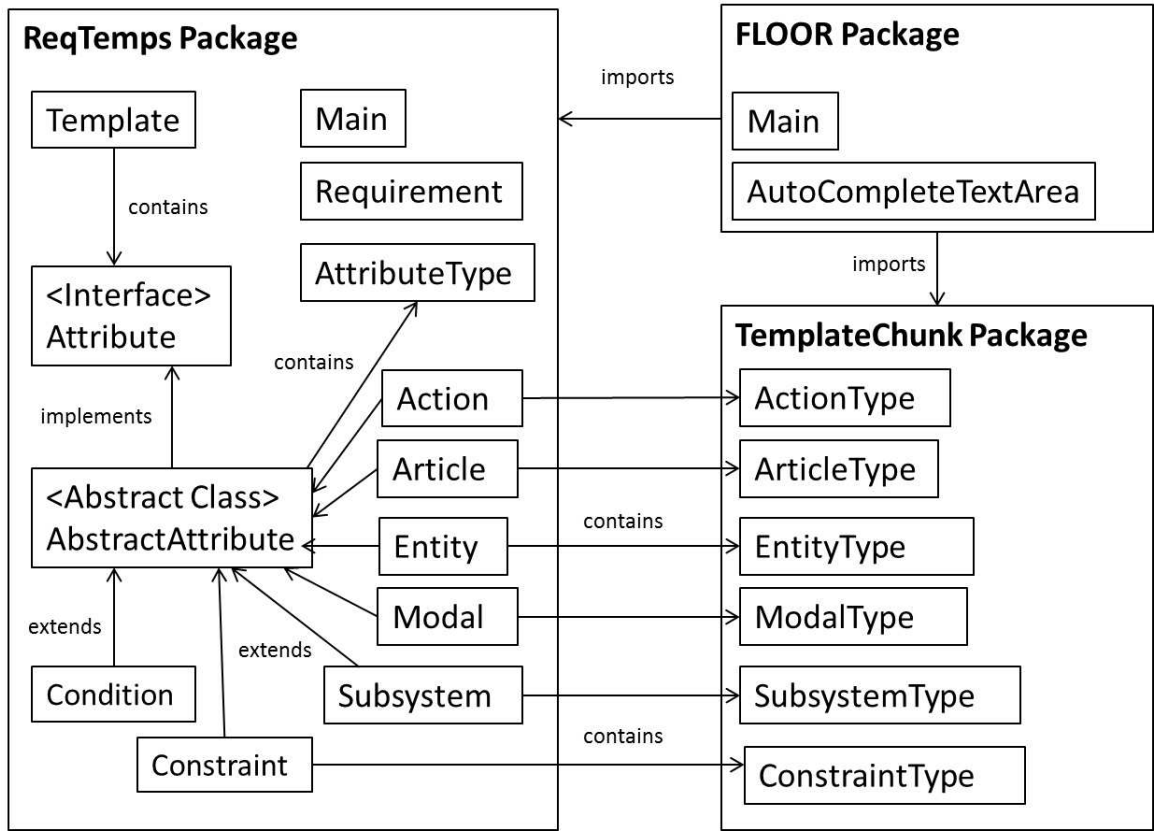


Figure 3.2: FLOOR: class diagram.

rate package, TemplateChunk. Each Attribute has an AttributeType enumerated in the TemplateChunk package.

3.3 NLP Libraries

To perform the NLP required to analyze requirements, FLOOR uses OpenNLP, an the open-source NLP library written in Java, and made available by Apache [2]. Specifically, FLOOR employs the tokenizer, POS-tagger, and chunker utilities provided by OpenNLP. The POS-tagger tags tokenized text according to the Penn Treebank tag-set, and the chunker accepts the same tags as input. Both the POS-tagger and chunker are pre-trained on English language corpora.

Chapter 4: **Requirements Engineering with FLOOR**

4.1 Working with FLOOR

In this section, we present the steps taken in the typical use case for FLOOR. First, the user selects supporting CSV files to load, containing requirements, requirement templates, and ontology terms. Next, the user begins to type a new requirement into the editor. Real-time feedback appears for requirement template and ontology term matching. After all new requirements have been entered, the user can generate reports to analyze the new requirements set. Once all new requirements have been entered, the user may then export the new requirements set to a CSV (comma separated variable) file.

4.1.1 Loading Existing Files

Figure 4.1 shows FLOOR's File Menu. The Import menu option allows the user to select CSV files containing existing requirements, requirement templates, and ontologies. If multiple ontology files are imported, FLOOR matches terms against each one. The user may select an option from the View Menu at any point, to examine the current requirements set (and editor), the loaded set of requirement

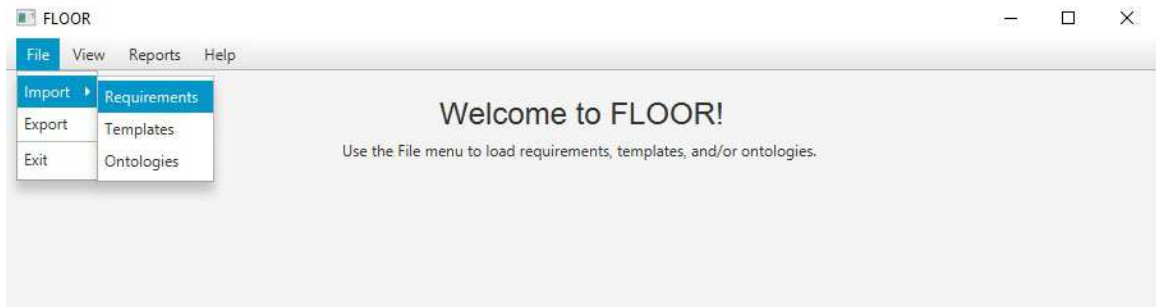


Figure 4.1: Import options on the FLOOR File Menu.

templates, or the loaded set of ontologies.

4.1.2 Requirement Template Matching

Figure 4.2 shows the real-time feedback provided when a user begins typing a new requirement. The first few words of the new requirement match to multiple requirement templates, which are displayed via a context menu. When new text is added to the requirement, the context menu containing requirement templates automatically updates. When the user is satisfied with the content of a new requirement, pressing *Enter* adds the new requirement to the bottom of the Requirement Table. Matching to a requirement template is not strictly enforced – it is ultimately the user’s decision whether a requirement is complete.

4.1.3 Ontology Term Matching

Also notice that in Figure 4.2, the word *windshield* has a green font color, and the word *multiply* has a red font color. Changes in font color occur automatically, based on ontology term matching. When the user types a word that matches a term in a loaded ontology, it automatically shows up in green. Likewise, if a noun,

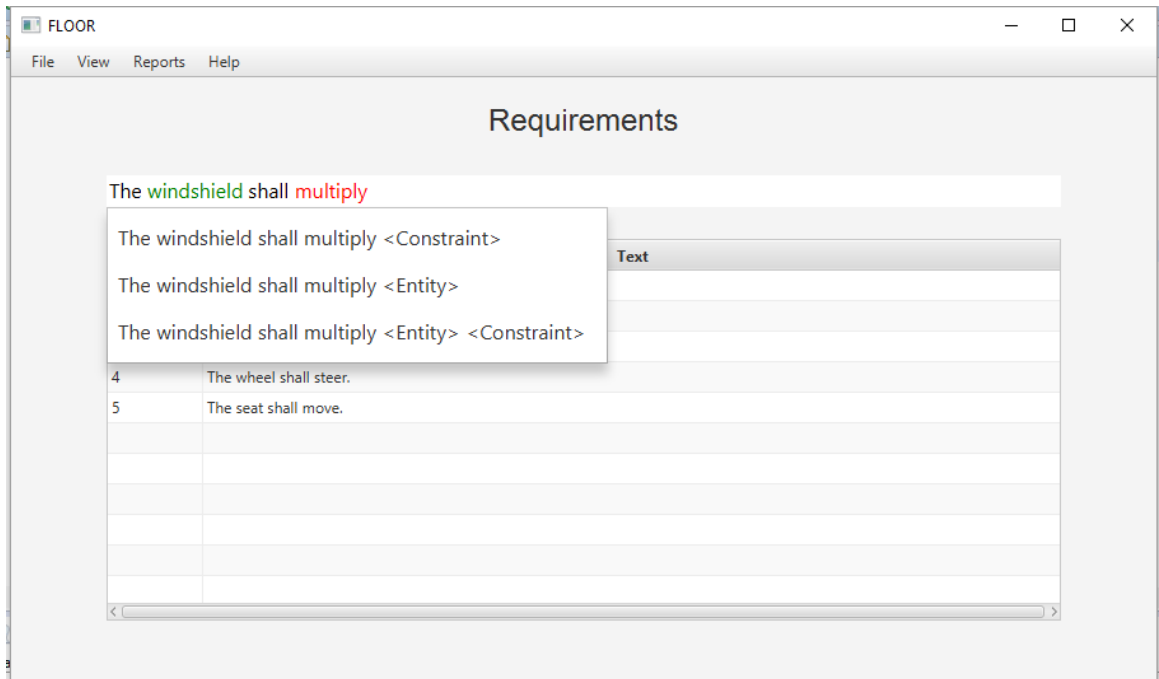
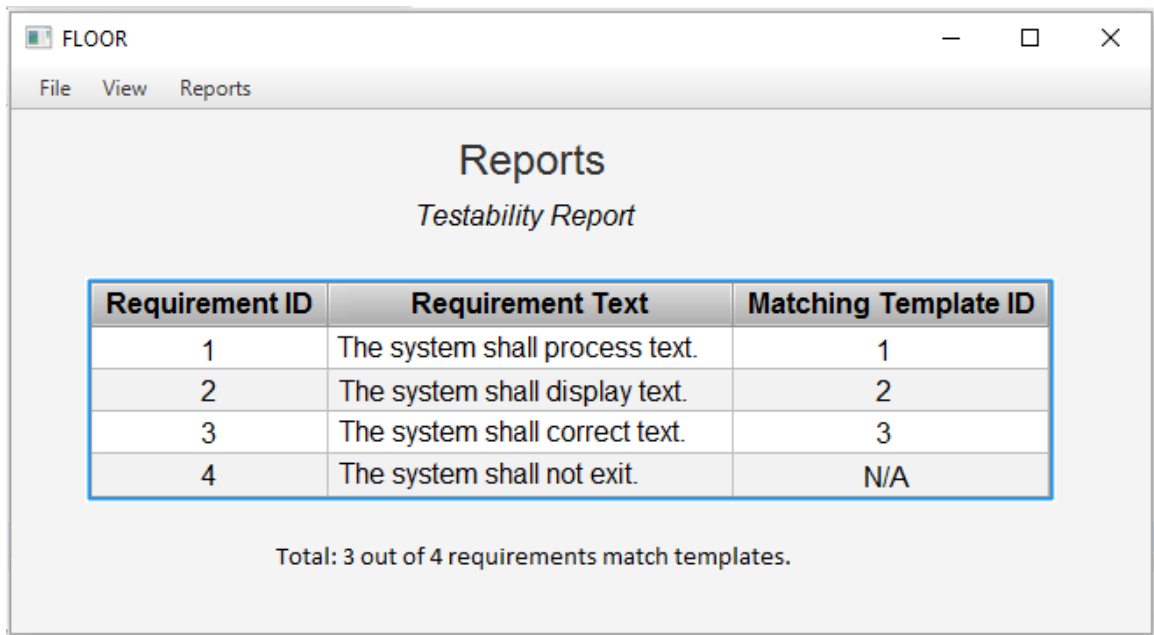


Figure 4.2: Real-time feedback: matching requirement templates and ontology terms.

verb, or adjective does not match a term in a loaded ontology, its font color is automatically changed to red. Non-matching terms are limited to certain parts of speech to prevent overwhelming the user with many subordinating words showing up in red.

4.1.4 Generating Analysis Reports

The Reports Menu allows the user to generate reports for two metrics: completeness and testability. The Completeness Report analyzes whether each term in the loaded ontologies appears at least once in any single requirement. The Testability Report analyses whether each requirement matches to one of the loaded requirement templates. We argue that the Testability Report indirectly addresses both unambiguity and singularity, as long as all loaded requirement templates pro-



Requirement ID	Requirement Text	Matching Template ID
1	The system shall process text.	1
2	The system shall display text.	2
3	The system shall correct text.	3
4	The system shall not exit.	N/A

Total: 3 out of 4 requirements match templates.

Figure 4.3: Example Testability Report.

mote both of these qualities. Each report provides a result for each requirement (or domain ontology term), as well as the total number of requirements (or domain ontology terms) satisfying the report metric.

An example screenshot of the testability report is shown below in Figure 4.3. In the figure, three out of four requirements match to requirement templates. The fourth requirement contains the phrase *shall not*, which does not match to a loaded requirement template. The reporting feature of FLOOR gives the requirements engineer a second line of defense (the first being the real-time feedback from the Requirement Editor) for requirements analysis.

4.1.5 Exporting Requirements

The File Menu contains an Export option 4.1. The Export option prompts the user to specify a location and file name for a CSV file. The generated CSV file

contains the current content of the Requirement Table, as seen in the Requirements View.

Chapter 5: Case Study Problems

5.1 Case Study 1: Simple Requirement Template Matching

As a precursor to a full use case for FLOOR, we first demonstrate the functionality provided by the subordinate ReqTemps package. ReqTemp's main method creates several requirements and requirement templates, and then matches them accordingly. In the following subsections, we include figure depicting the output of each step, and a brief description.

5.1.1 Creating and Printing Requirements and Requirement Templates

Part 1 instantiates several requirements, and Part 2 instantiates several templates. The requirements are shown below in Figure 5.1. Figure 5.2 shows the requirement templates.

```
=====
PART 1: Create & Print Requirements
=====
The monitor will display images.
The cursor must move upon mouse movement.
Within 1 second, the monitor shall update.
The processor will accept interrupts.
The processor shall execute.
```

Figure 5.1: Simple template matching: Create and print requirements.

```
=====
PART 2: Create & Print Templates
=====
Template_1:
AttributeType: Article
AttributeType: Subsystem
AttributeType: Modal
AttributeType: Action

Template_2:
AttributeType: Condition
AttributeType: Article
AttributeType: Subsystem
AttributeType: Modal
AttributeType: Action

Template_3:
AttributeType: Article
AttributeType: Subsystem
AttributeType: Modal
AttributeType: Action
AttributeType: Condition

Template_4:
AttributeType: Article
AttributeType: Subsystem
AttributeType: Modal
AttributeType: Action
AttributeType: Entity
```

Figure 5.2: Simple template matching: Create and print templates.

5.1.2 Tokenization and POS-Tagging

Part 3 uses OpenNLP’s “SimpleTokenizer” to parse the set of input requirements into individual words. Then, OpenNLP’s POS-tagger labels each token with a part-of-speech. The result are shown below in Figure 5.3.

```
=====
PART 3: Tokenization and POS-Tagging
=====
The|monitor|will|display|images|.
DT|NN|MD|VB|NNS|.
The|cursor|must|move|upon|mouse|movement|.
DT|NN|MD|VB|IN|NN|NN|.
Within|1|second|,|the|monitor|shall|update|.
IN|CD|JJ|,|DT|NN|MD|VB|.
The|processor|will|accept|interrupts|.
DT|NN|MD|VB|NNS|.
The|processor|shall|execute|.
DT|NN|MD|VB|.
```

Figure 5.3: Simple template matching: Tokenization and POS tagging.

5.1.3 Matching Requirements with Requirement Templates

Part 4 processes the tagged tokens to match them to requirement templates, and prints the resulting HashMap. The final processing is based on the location of the modal (*will*, *must*, or *shall*) in the tokenized requirement, as well as knowledge of how the templates are constructed. With this information, ReqTemps works backwards to uncover the matching template, as shown in Figure 5.4. ReqTemps does not explicitly use the results of the POS-tagger – that piece of functionality is left to FLOOR.

```
=====
PART 4: Match Requirements to Templates
=====
{The processor shall execute.=Template_1:
AttributeType: Article
AttributeType: Subsystem
AttributeType: Modal
AttributeType: Action
, The monitor will display images.=Template_4:
AttributeType: Article
AttributeType: Subsystem
AttributeType: Modal
AttributeType: Action
AttributeType: Entity
, The processor will accept interrupts.=Template_4:
AttributeType: Article
AttributeType: Subsystem
AttributeType: Modal
AttributeType: Action
AttributeType: Entity
, The cursor must move upon mouse movement.=Template_3:
AttributeType: Article
AttributeType: Subsystem
AttributeType: Modal
AttributeType: Action
AttributeType: Condition
, Within 1 second, the monitor shall update.=Template_2:
AttributeType: Condition
AttributeType: Article
AttributeType: Subsystem
AttributeType: Modal
AttributeType: Action
}
```

Figure 5.4: Simple template matching: Textual requirements matched with templates.

5.2 Case Study 2: Working with Requirements from NASA Goddard

The utility of FLOOR will ultimately be determined by systems engineers in industry performing requirements engineering tasks every day. That said, it is still interesting to use FLOOR to retroactively analyze a requirements set, assuming some default set of requirement templates and ontologies. Although the FLOOR's benefits during initial requirement creation will not be observed by this method, the reporting features can at least be studied.

5.2.1 Import Data

For this case study, we obtained a requirement set consisting of 14 requirements written for NASA's Global Precipitation Mission (GPM) Project. The test requirements are listed in Table 5.1. In the absence of any requirement templates actually used during the requirements' creation, we chose a default set of eight fairly simple templates listed in Table 5.2. We also worked backwards to create ontologies, one for acronyms found in the requirements, and one for physical units. The following tables contain case study the requirements, requirement templates, and ontologies.

ID	Requirement Text
1	The GPM shall make measurements that enable the determination of rainfall mean drop size, encompassing mean drop sizes ranging from 0.5 to 3 mm.
2	The PIS shall include a DFPR.
3	The CSB shall be capable of ingesting an average rate of 95 kbps continuously from the PR-U, and 95 kbps continuously from the PR-A.
4	The spacecraft bus shall provide position information from the GPS receiver to the DPR.
5	The CS shall use a GPS receiver for orbit position information and time determination.
6	The CS shall accommodate the PIS with technical resources (mass, power, FOV, command and data, etc.) and operating environment (pointing, thermal control, etc.)
7	The CS shall provide structural support for the PR-U, PR-A, GMI, and auxiliary instruments, with a total mass of up to 1027 kg.
8	The CSB shall provide orientation and clear Field-of-View for each instrument in accordance with the instrument mechanical ICDs.
9	The CS shall provide the PR-U, PR-A, GMI, and auxiliary instruments with DC unregulated power up to 896 watts steady state at beginning of life and 796 W end of life.
10	The CSB shall be capable of ingesting an average rate of 20 kbps continuously from the GMI, with no more than 1 kbps of that as housekeeping data.
11	The CS shall provide structural support and stability sufficient to maintain coalignment among the various instruments' mechanical reference surfaces to within 0.1 deg (3 sigma) per axis.
12	The DPR and CS design shall overlap the PR-A and PR-U beams sufficiently (no more than 0.3 degrees apart) that drop-size distribution can be determined.
13	If aligning the DPR radar beams so that they both sample the same volume proves to be sufficiently difficult such that the instrument cannot measure accurate drop size distribution; then GPM may not meet one of its Level 1 requirements.
14	The DPR shall make measurements in both Ku and Ka frequency bands.

Table 5.1: Case study requirement set (Source: NASA's GPM Project).

ID	Requirement Template
1	<i><article> <subsystem> <modal> <action></i>
2	<i><article> <subsystem> <modal> <action> <entity></i>
3	<i><article> <subsystem> <modal> <action> <constraint></i>
4	<i><article> <subsystem> <modal> <action> <entity> <constraint></i>
5	<i><condition> <article> <subsystem> <modal> <action></i>
6	<i><condition> <article> <subsystem> <modal> <action> <entity></i>
7	<i><condition> <article> <subsystem> <modal> <action> <constraint></i>
8	<i><condition> <article> <subsystem> <modal> <action> <entity> <constraint></i>

Table 5.2: Case study requirement templates.

ID	Term
CS	Core Spacecraft
CSB	Core Spacecraft Bus
DFPR	Dual Frequency Precipitation Radar
DPR	Dual Precipitation Radar
FOV	Field of View
GMI	GPM Microwave Imager
GPM	Global Precipitation Mission
GPS	Global Positioning System
ICD	Interface Control Document
PIS	Primary Instrument Suite
PR-A	Precipitation Radar A
PR-U	Precipitation Radar U

Table 5.3: Case study acronym ontology.

Term	Definition
deg	degrees
Ka	26.5 to 40 GHz
Ku	12 to 18 GHz
kbps	kilobits per second
kg	kilograms
mm	millimeters
W	watts

Table 5.4: Case study units ontology.

5.2.2 Results

We found that only 5 out of the 14 requirements matched to one of our eight requirement templates. The reasons for certain requirements' failures to match were interesting. One requirement contained hyphenated terms that caused problems for the chunker, consequently the requirement was not adequately processed. In another case, a seemingly well-constructed requirement did not match any requirement templates because we did not load any requirement templates that contained a constraint followed by a condition.

5.3 Case Study 3: Scalability Analysis

This section discusses the scalability of FLOOR as related to large requirement sets and ontologies. It is important for FLOOR to be usable for both the requirements development and requirements analysis of large requirement sets and/or ontologies. To test the scalability of FLOOR, we measured the response time of the Requirements View after importing requirement sets of varying sizes. The requirement sets were sized as follows: 100, 500, 1000, 5000, and 10000 requirements. For each case, we imported an ontologies containing 100 and 1000 terms, and also timed the report generation. FLOOR successfully loaded the Requirements View for each set, and averaged required approximately 4 seconds of loading time for every 1000 requirements. We also note that the rate was similar for report generation, and was unaffected by the size and number of loaded ontologies.

We considered this load rate to be satisfactorily scalable, given that upwards

of 7000 requirements would require just a 30-second wait time. However, the case could be made that the current implementation is not quite robust enough to handle requirement sets containing 15000 or more requirements, since a full minute of wait time might be unacceptable. From this standpoint, FLOOR's handling of requirement sets could be further optimized, but it is not a critical need at this time.

Chapter 6: **Conclusions and Future Work**

6.1 Conclusions

This thesis introduces FLOOR, a new tool for requirements engineering that provides real-time feedback to the user regarding the quality of new requirements. FLOOR leverages NLP to match new requirement text to potential requirement templates, and to alert the user of terminological consistency with existing ontologies. A critical feature of FLOOR is ability to load and match against multiple ontologies – a new development in the requirements engineering field. We see FLOOR as a building block towards the next generation of model-based systems engineering tools with enhanced automation, enabling systems engineers to recognize and solve problems as early in the system life cycle as soon as possible.

6.2 Future Work

Looking to the future, we envision several improvements to FLOOR. The prototype introduced in this thesis is mainly concerned with the creation of new requirements, as opposed to the editing and maintenance of existing requirements. To that end, FLOOR could be equipped with the capability of selecting and editing

existing requirements. Another limitation of FLOOR is the treatment of ontologies as only lists of terms.

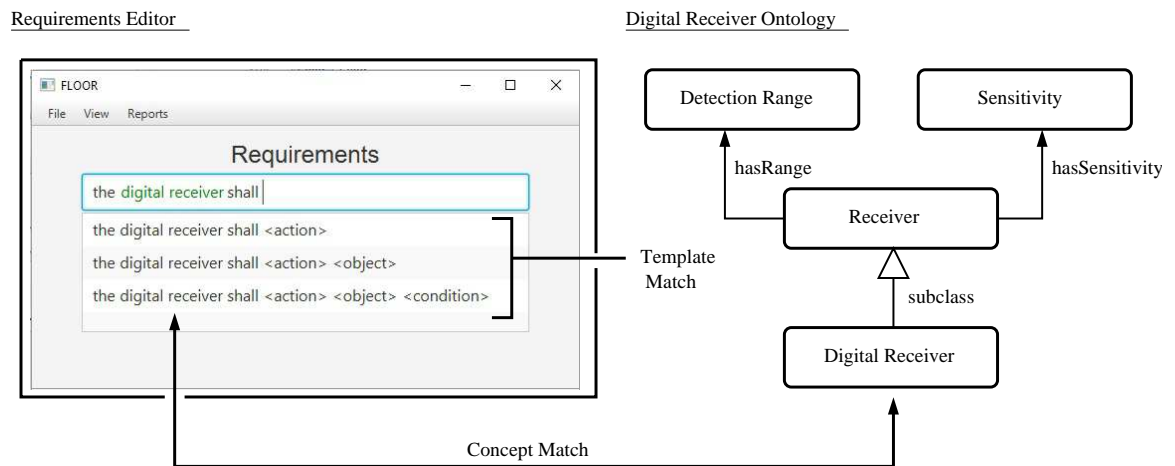


Figure 6.1: FLOOR: Requirement template and domain ontology match.

As illustrated in Figure 6.1, further integration with mainstream ontology formats, like OWL, and semantic modeling software tools, like Jena, would enable FLOOR to use the conceptual information contained within ontologies [1, 33].

Longer-term opportunities for future work include:

1. Distributing FLOOR to corporations in the systems engineering industry, with the goal of obtaining user accounts on the tool's utility, and desired improvements. FLOOR has been presented to Northrop Grumman at two internal symposia, and a beta version will soon be made available to the company.
2. Increasing the number of reporting options. Additional reporting options could include ambiguity, consistency, and singularity, as well as other quality metrics.
3. Using of several large requirements sets as training corpora for the POS-tagging and chunking models used by FLOOR. In this way, FLOOR could become

more tailored to the specific textual patterns that frequently arise in requirement text.

4. Integrating of FLOOR as a plug-in for a requirements database, e.g., DOORS.

Using this approach, existing requirements management practices could be augmented with FLOOR's enhancements for requirements development and analysis.

Appendix A: RichTextFX License Agreement

Copyright (c) 2013-2017, Tomas Mikula and contributors

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DIS-

CLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

[Contact GitHub API Training Shop Blog About](#)

[2017 GitHub, Inc. Terms Privacy Security Status Help](#)

Bibliography

- [1] 2013. *Apache Jena*, accessible at : <http://www.jena.apache.org>; Accessed on 11/27/13.
- [2] 2016. *Apache OpenNLP*, accessible at : <https://opennlp.apache.org>; Accessed on 04/01/16.
- [3] Ambriola V. and Gervasi V. Processing Natural Language Requirements. In *Proceedings 12th IEEE International Conference Automated Software Engineering*, pages 36–45. IEEE Comput. Soc, 1997.
- [4] Ananiadou S. A Methodology for Automatic Term Recognition. In *Proceedings of 15th International Conference on Computational Linguistics (COLING94)*, pages 1034–1038, 1994.
- [5] Arellano A., Zontek-Carney E., and Austin M. A. Frameworks for Natural Language Processing of Textual Requirements. *International Journal On Advances in Systems and Measurements*, 8(No. 3 and 4):230–240, December 2015.
- [6] Arellano A., Zontek-Carney E., and Austin M. A. Natural Language Processing of Textual Requirements. In *The Tenth International Conference on Systems (ICONS 2015)*, pages 93–97, Barcelona, Spain, April 19–24 2015.
- [7] Austin M.A., and J.S. Baras J.S. “*An Introduction to Information-Centric Systems Engineering*”. Tutorial F06, INCOSE, Toulouse, France, June 2004.
- [8] Austin M.A. and Wojcik C.E. Ontology-Enabled Traceability Mechanisms. In *20th Annual International Symposium of The International Council on Systems Engineering (INCOSE 2012)*, Chicago, USA, July 12-15 2012.
- [9] Austin M.A., Mayank V., and Shmunis N. PaladinRM: Graph-Based Visualization of Requirements Organized for Team-Based Design. *Systems Engineering: The Journal of the International Council on Systems Engineering*, 9(2):129–145, May 2006.

- [10] Delgoshaei, P. and Austin, M.A. and Pertzborn, A. A Semantic Framework for Modeling and Simulation of Cyber-Physical Systems. *International Journal On Advances in Systems and Measurements*, 7(3-4):223–238, December 2014.
- [11] Delgoshaei, P. and Austin, M.A and Veronica, D.A. A Semantic Platform Infrastructure for Requirements Traceability and System Assessment. *The Ninth International Conference on Systems (ICONS 2014)*, February 2014.
- [12] Dynamic Object Oriented Requirements System (DOORS). See <http://www.telelogic.com/products/doorsers/doors/>. 2009.
- [13] Earl L.L. Experiments in Automatic Extracting and Indexing. *Information Storage and Retrieval*, 6(6):273–298, 1970.
- [14] Farfeleder S., Moser T., Krall A., et al. Ontology-Driven Guidance for Requirements Elicitation. *The Semantic Web: Research and Applications*, ():212–226, 2010.
- [15] Farfeleder S., Moser T., Krall A., et al. DODT: Increasing Requirements Formalism using Domain Ontologies for Improved Embedded Systems Development. In *IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 212–226, 2011.
- [16] Fedorenko D., Astrakhantsev N., and Turdakov D. Automatic Recognition of Domain-Specific Terms: An Experimental Evaluation. In *Proceedings of SYRCoDIS 2013*, pages 15–23, 2013.
- [17] Ferreira D., Silva A. A Controlled Natural Language Approach for Integrating Requirements and Model-Driven Engineering. In *International Conference on Software Engineering Advances*, 2009.
- [18] Frantzi K., Ananiadou S., and Mima H. Automatic Recognition of Multi-Word Terms: The C-Value/NC-Value Method. *International Journal on Digital Libraries*, 3(2):115–130, 2000.
- [19] Haspelmath M. Word Classes and Parts of Speech. 2001.
- [20] Hull E., Jackson K. and Dick J. *Requirements Engineering*. Springer, 2002.
- [21] Judea A., Schutze E., and Bruegmann S. Unsupervised Training Set Generation for Automatic Acquisition of Technical Terminology in Patents. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 290–300, Dublin, Ireland: Dublin City University and Association for Computational Linguistics, 2014.
- [22] Kageura K. and Umino B. Methods of Automatic Term Recognition: A Review. *Terminology*, 3(2):259–289, 1996.

- [23] Kof L. From Requirements Documents to System Models: A Tool for Interactive Semi-Automatic Translation. In *IEEE International Requirements Engineering Conference*, 2010.
- [24] Kozakov L., Park Y., Fin T., et al. Glossary Extraction and Utilization in the Information Search and Delivery System for IBM Technical Support. *IBM Systems Journal*, 43(3):546–563, 2004.
- [25] Krithika L.B., Akondi K.V. Survey on Various Natural Language Processing Toolkits. *World Applied Sciences Journal*, 32(3):399–402, 2014.
- [26] NLTK Project. Natural Language Toolkit NLTK 3.0 documentation.
- [27] Requirements Working Group, INCOSE. Guide for Writing Requirements. *INCOSE Technical Report*, ():, 2012.
- [28] Rolland C. and Proix C. A Natural Language Approach for Requirements Engineering. In *Advanced Information Systems Engineering*, pages 257–277. Springer, 1992.
- [29] Ryan K. The Role of Natural Language in Requirements Engineering. In *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 240–242. IEEE Comput. Soc. Press, 1993.
- [30] Santorini B. Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision), 1990.
- [31] Stalhane T., Wien T. The DODT Tool Applied to Subsea Software. In *IEEE 22nd International Requirements Engineering Conference*, 2014.
- [32] Tommila T., Pakonen A. Controlled natural language requirements in the design and analysis of safety critical I&C systems. In *SAREMAN project*, 2013.
- [33] World Wide Web Consortium(W3C). *OWL 2 Web Ontology Language Profiles (Second Edition)*. In *W3C Recommendation 11 December 2012, Available at: <http://www.w3.org/TR/2012/REC-owl2-profiles-20121211/>*, 2012.