Frameworks for Natural Language Processing of Textual Requirements

Andres Arellano
Government of Chile,
Santiago, Chile
Email: andres.arellano@gmail.com

Edward Zontek-Carney
Northrop Grumman Corporation,
Baltimore, MD 21240, USA
Email: Ecarney1@umd.edu

Mark A. Austin
Department of Civil Engineering,
University of Maryland,
College Park, MD 20742, USA
Email: austin@isr.umd.edu

***Abstract*—Natural language processing is the application of automated parsing and machine learning techniques to analyze standard text. Applications of NLP to requirements engineering include extraction of ontologies from a requirements specification, and use of NLP to verify the consistency and/or completeness of a requirements specification. This paper describes a new approach to the interpretation, organization, and management of textual requirements through the use of application-specific ontologies and natural language processing. We also design and exercise a prototype software tool that implements the new framework on a simplified model of an aircraft.**

Keywords-Systems Engineering; Ontologies; Natural Language Processing; Requirements; Rule Checking.

I. INTRODUCTION

Problem Statement. Model-based systems engineering development is an approach to systems-level development in which the focus and primary artifacts of development are models, as opposed to documents. This paper describes a new approach to the interpretation, organization, and management 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 [1] [2].

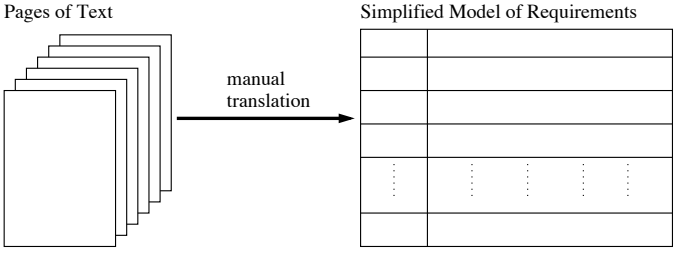

Figure 1. Manual translation of text into high-level textual requirements.

As engineering systems become increasingly complex the need for automation arises. A key required capability is the identification and management of requirements during the early phases of the system design process, when errors are cheapest and easiest to correct. While engineers are looking for semi-formal and formal models to work with, the reality remains that many large-scale projects begin with hundreds – sometimes thousands – of pages of textual requirements, which may be inadequate because they are incomplete, under specified, or perhaps ambiguous. State-of-the art practice (see Figure 1) involves the manual translation of text into a semi-formal format (suitable for representation in a requirements database). This is a slow and error prone process. A second key problem is one of completeness. For projects defined by hundreds/thousands of textual requirements, how do we know a system description is complete and consistent?

Scope and Objectives. Looking ahead, our work is motivated by a strong need for computer processing tools that will 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] [4] [5]. Applications range from using NLP to extract ontologies from a requirements specification, to using NLP to verify the consistency and/or completion of a requirements specification.

Our near-term research objectives are to use modern language processing tools to scan and tag a set of requirements, and offer support to systems engineers in their task of defining and maintaining a comprehensive, valid and accurate body of requirements. The general idea is as follows: Given a set of textual descriptions of system requirements, we could analyze them using natural language processing tools, extracting the objects or properties that are referenced within the requirements. Then, we could match these properties against a defined ontology model corresponding to the domain of this particular requirement. Such a system would throw alerts in case of system properties lacking requirements, and requirements that are redundant and/or conflicting.

Figure 2 shows the framework for automated transformation of text (documents) into textual requirements (semi-formal models) described in this paper. Briefly, NLP processing techniques are applied to textual requirements to identify parts of speech – sentences are partitioned into words and then classified as being parts of speech (e.g., nouns, verbs, etc.). Then, the analyzed text is compared against semantic models consisting of domain ontologies and ontologies for specific applications. System ontologies are matched with system properties; subsystem ontologies are matched with subsystem properties, and component ontologies are matched with component properties. Feedback is necessary when semantic descriptions of applications do not have complete coverage, as defined by the domain ontologies.

The contents of this paper are as follows: Section II

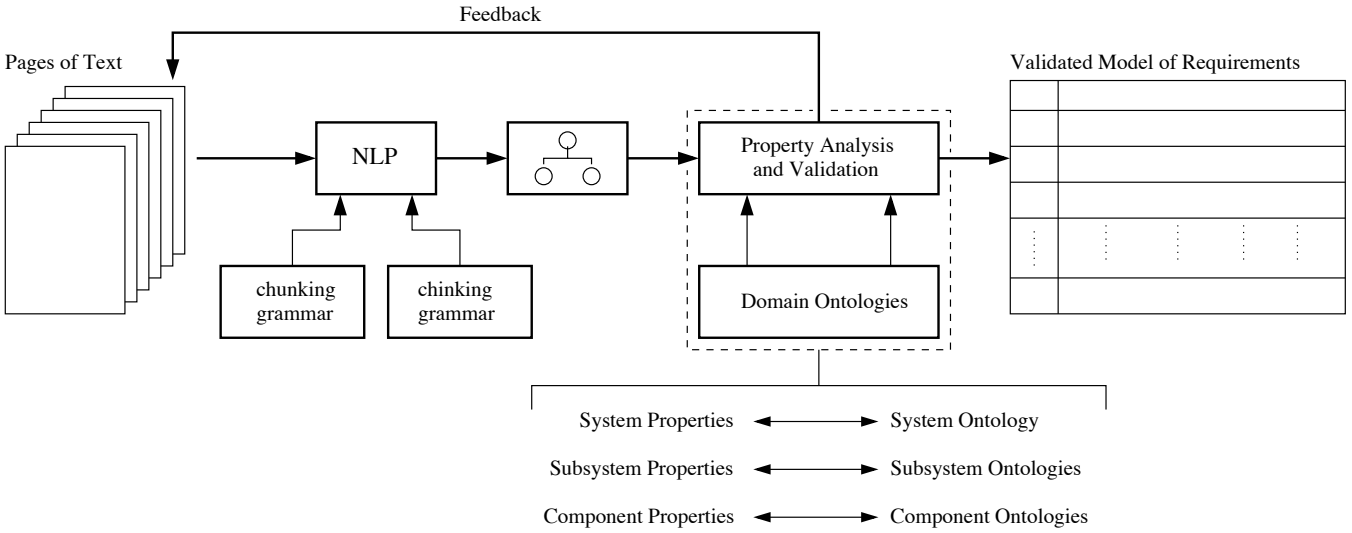

Figure 2. Framework for automated transformation of text (documents) into textual requirements (semi-formal models).

explains the role that semantics can play in modern engineering systems design and management. Its second purpose is to briefly explain state-of-the-art capability in automatic term recognition and automatic indexing. Section III describes two aspects of our work: (1) Working with NLTK, and (2) Chunking and Chinking. The framework for integration of NLP with ontologies and textual requirements is covered in Section IV. Two applications are presented in Section V: (1) Requirements and ontologies for a simple aircraft application, and (2) A framework for the explicit representation of multiple ontologies. Sections VI and VII discuss opportunities for future work and the conclusions of this study.

II. State-of-the-Art Capability

Role of Semantics in Engineering Systems Design and Management. A tenet of our work is that methodologies for strategic approaches to design will employ semantic descriptions of application domains, and use ontologies and rule-based reasoning to enable validation of requirements, automated synthesis of potentially good design solutions, and communication (or mappings) among multiple disciplines [6] [7] [8]. A key capability is the identification and management of requirements during the early phases of the system design process, where errors are cheapest and easiest to correct. The systems architecture for state-of-the-art requirements traceability and the proposed platform model [9], [10] is shown in the upper and lower sections of Figure 3. In state-of-the-art traceability mechanisms design requirements are connected directly to design solutions (i.e., objects in the engineering model). Our contention is that an alternative and potentially better approach is to satisfy a requirement by asking the basic question: What design concept (or group of design concepts) should I apply to satisfy a requirement? Design solutions are the instantiation/implementation of these concepts. The proposed architecture is a platform because it contains collections of domain-specific ontologies and design rules that will be reusable across applications. In the lower half of Figure 3, the textual requirements, ontology, and engineering models provide distinct views of a design: (1) Requirements are a statement of "what is required." (2) Engineering models are a statement of "how the required functionality and performance might be achieved," and (3) Ontologies are a statement of "concepts justifying a tentative design solution." During design, mathematical and logical rules are derived from textual requirements which, in turn, are connected to elements in an engineering model. Evaluation of requirements can include checks for satisfaction of system functionality and performance, as well as identification of conflicts in requirements themselves. A key benefit of our approach is that design rule checking can be applied at the earliest stage possible – as long as sufficient data is available for the evaluation of rules, rule checking can commence; the textual requirements and engineering models need not be complete. During the system operation, key questions to be answered are: What other concepts are involved when a change occurs in the sensing model? What requirement(s) might be violated when those concepts are involved in the change? To understand the inevitable conflicts and opportunities to conduct trade space studies, it is important to be able to trace back and understand cause-and-effect relationships between changes at system-component level, and their effect on stakeholder requirements. Present-day systems engineering methodologies and tools, including those associated with SysML [11] are not designed to handle projects in this way.

Automatic Term Recognition and Automatic Indexing. Strategies for automatic term recognition and automatic indexing fall into the general area of computational linguistics [12]. 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., part-of-speech tagging) and statistical information acquired from the frequency of usage of terms in candidate documents [14] [15]. 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] [17] [18].

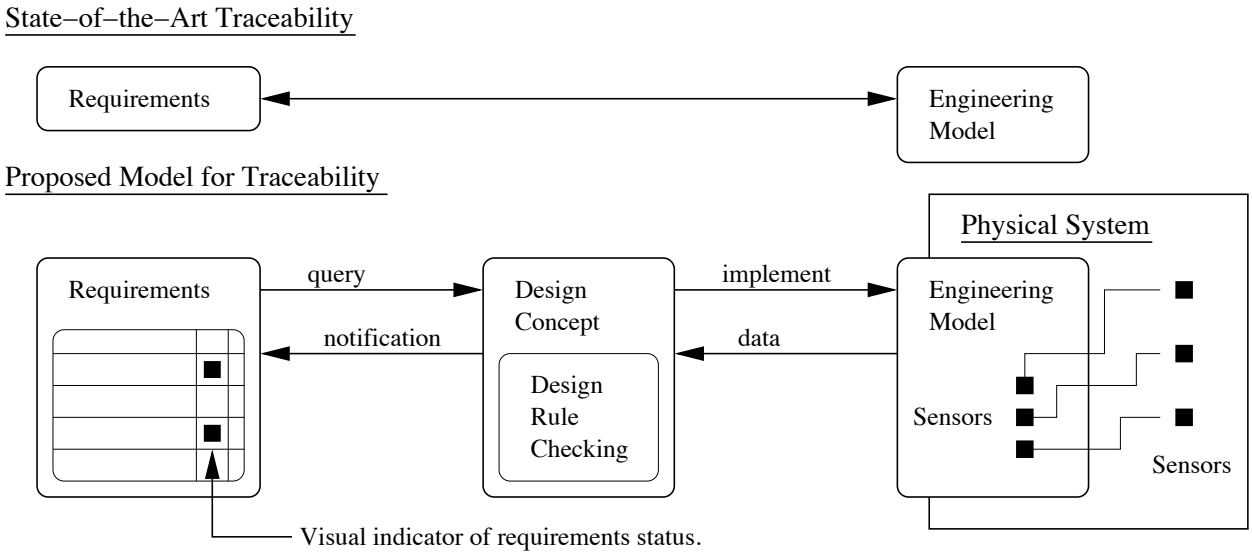

Figure 3. Schematics for: (top) state-of-the-art traceability, and (bottom) proposed model for ontology-enabled traceability for systems design and management.

III. NATURAL LANGUAGE PROCESSING OF REQUIREMENTS

Working with NLTK. The Natural Language Toolkit (NLTK) is a mature open source platform for building Python programs to work with human language data [19].

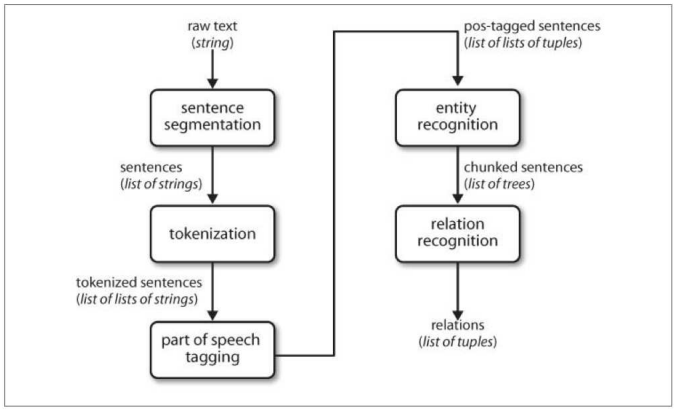

Figure 4. Information extraction system pipeline architecture.

Figures 2 and 4 show the essential details of a pipeline for text (documents) to textual requirements (semi-formal models) transformation. NLTK provides the basic pieces to accomplish those steps, each one with different options and degrees of freedom. Starting with an unstructured body of words (i.e., raw text), we want to obtain sentences (the first step of abstraction on top of simple words) and have access to each word independently (without losing its context or relative positioning to its sentence). This process is known as *tokenization* and it is complicated by the possibility of a single word being associated with multiple token types. Consider, for example, the sentence: "These prerequisites are known as (computer) system requirements and are often used as a guideline as opposed to an absolute rule." The abbreviated script of Python code is as follows:

```
text = "These prerequisites are known as (computer)
        system requirements and are often used as a
        guideline as opposed to an absolute rule."
tokens = nltk.word_tokenize(my_string)
print tokens
=>
['These', 'prerequisites', 'are', 'known', 'as',
 '(', 'computer', ')', 'system', 'requirements',
 'and', 'are', 'often', 'used', 'as', 'a',
 'guideline', 'as', 'opposed', 'to', 'an',
 'absolute', 'rule', '.']
```

The result of this script is an array that contains all the text's tokens, each token being a word or a punctuation character. After we have obtained an array with each token (i.e., word) from the original text, we may want to normalize these tokens. This means: (1) Converting all letters to lower case, (2) Making all plural words singular ones, (3) Removing *ing* endings from verbs, (4) Making all verbs be in present tense, and (5) Other similar actions to remove meaningless differences between words. In NLP jargon, the latter is known as *stemming*, in reference to a process that strips off affixes and leaves you with a stem [20]. NLTK provides us with higher level *stemmers* that incorporate complex rules to deal with the difficult problem of stemming. The Porter stemmer that uses the algorithm presented in [21], the Lancaster stemmer, based on [22], or the built in lemmatizer – Stemming is also known as *lemmatization*, referencing the search of the *lemma* of which one is looking an inflected form [20] – found in WordNet. Wordnet is an open lexical database of English maintained by Princeton University [23]. The latter is considerably slower than all the other ones, since it has to look for the potential stem into its database for each token.

The next step is to identify what role each word plays on the sentence: a noun, a verb, an adjective, a pronoun, preposition, conjunction, numeral, article and interjection [24]. This process is known as *part of speech tagging*, or simply *POS tagging* [25]. On top of POS tagging we can identify the *entities*. We can think of these *entities* as "multiple word nouns" or objects that are present in the text. NLTK provides

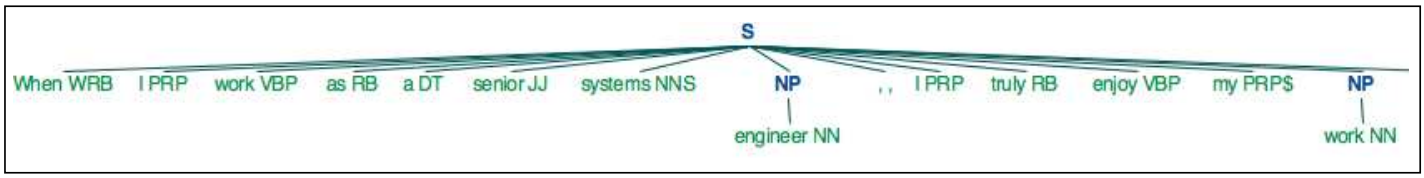

Figure 5. Output from first step on building chunking grammar. Purpose: Simply pick nouns from test sentence.

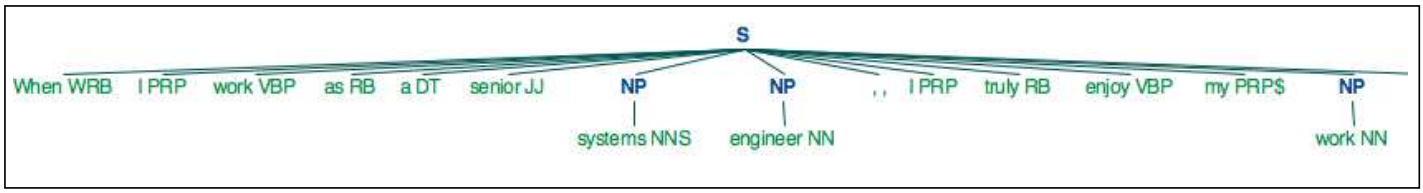

Figure 6. Output from second step on building chunking grammar. Purpose: Identify noun phrases.

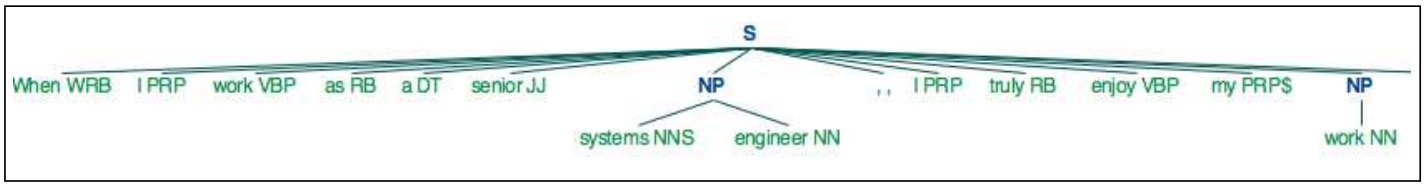

Figure 7. Output from third step on building chunking grammar. Purpose: Form noun phrases.

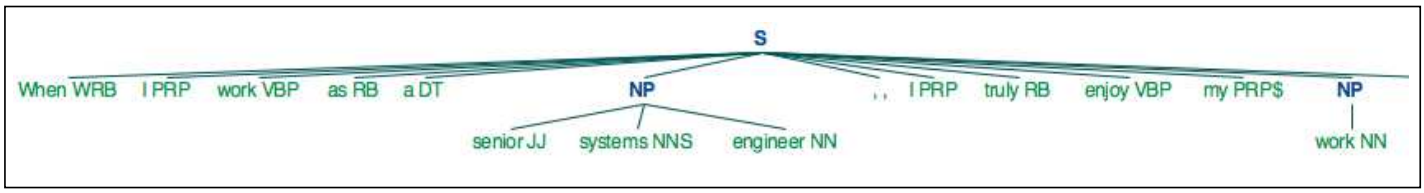

Figure 8. Output from fourth step on building chunking grammar. Purpose: Identify the adjective preceding the first noun phrase.

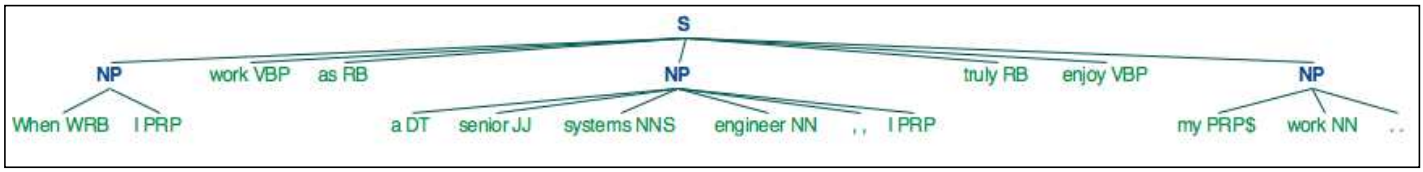

Figure 9. Output from the example on chinking. Purpose: Exclude base verbs and adverbs.

an interface for tagging each token in a sentence with supplementary information such as its part of speech. Several taggers are included, but an *off-the-shelf* one is available, based on the Penn Treebank tagset [26]. The following listing shows how simple is to perform a basic part of speech tagging.

```
my_string = "When I work as a senior systems
            engineer, I truly enjoy my work."
tokens = nltk.word_tokenize(my_string)
print tokens

tagged_tokens = nltk.pos_tag(tokens)
print tagged_tokens
=>
[('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 represent a lexical category or part of speech. For instance, WRB stands for *Wh-adverb*, including *how*, *where*, *why*, etc. PRP stands for *Personal pronoun*; *RB* for *Adverb*; *JJ* for *Adjective*, *VBP* for *Present verb tense*, and so forth [27]. These categories are more detailed than presented in [24], 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. In order 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')*). NLTK offers regular expression processing support for identifying groups of tokens, specifically noun phrases, in the text.

Chunking and Chinking. Chunking and chinking are techniques for extracting information from text. Chunking is a basic technique for segmenting and labeling multi-token sequences, including noun-phrase chunks, word-level tokenization and part-of-speech tagging. To find the chunk structure for a given sentence, a regular expression parser begins with a flat structure in which no tokens are chunked. The chunking rules are applied in turn, successively updating the chunk structure. Once all of the rules have been invoked, the resulting chunk structure is returned. We can also define patterns for what kinds of words should be excluded from a chunk. These unchunked words are known as chinks. In both cases, the rules for the parser are specified defining *grammars*, including patterns, known as *chunking*, or excluding patterns, known as *chinking*.

Figures 5 through 8 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. This is accomplished with the script:

```
grammar     = "NP: {<NN>}"
chunker     = nltk.RegexpParser(grammar)
chunks_tree = chunker.parse(tagged_tokens)
```

Figure 5 is a graphical representation of the results – NLTK identifies "engineer" as a noun. But even this seems not to be correctly done since we are missing the noun systems. The problem is that our grammar is overly simple and cannot even handle noun modifiers, such as NNS for the representation of plural nouns. The second version of our script:

```
grammar = "NP: {<NN.*>}"
chunker = nltk.RegexpParser(grammar)
chunks_tree = chunker.parse(tagged_tokens)
```

aims to include different types of nouns. The output is shown in Figure 6. Now we can see all three nouns properly identified. Unfortunately, the first two are not forming a single noun phrase, but two independent phrases. The refined script:

```
grammar = "NP: {<NN.*>+}"
chunker = nltk.RegexpParser(grammar)
chunks_tree = chunker.parse(tagged_tokens)
```

take care of this problem by adding a match-one-or-more operator `*`. The output is shown in Figure 7. The final script:

```
grammar = "NP: {<JJ.*>*<NN.*>+}"
chunker = nltk.RegexpParser(grammar)
chunks_tree = chunker.parse(tagged_tokens)
```

advances the parsing process a few steps further. We already know that we want to consider any kind of adjectives, so we add the match-one-or-more operator `*` after the adjective code `JJ`. And we use `*` to permit other words to be present between the adjective and the noun(s). Figure 8 shows the output for this last step. We have identified two entities, *senior systems engineer* and *work*, and that is precisely what we want. Incremental development of the chunking grammar is complete.

Chinking is the complementary process of removing tokens from a chunk. The script:

```
grammar = r"""
  NP: {<.*>+}
      }<VB.*>{
      }<RB.*>{
  """
chunker = nltk.RegexpParser(grammar)
chunks_tree = chunker.parse(tagged_tokens)
```

says chunk everything (i.e., `NP: {<.*>+}`, and then remove base verbs (i.e., `VB`) and adverbs (i.e., `RB`) from the chunk. When this script is executed on our test sentence the result is three noun phrase (i.e., `NP`) trees, as shown along the bottom of Figure 9.

IV. Systems Integration

Integration of NLP with Ontologies and Textual Requirements. In order to provide a platform for the integration of natural language processing, ontologies and systems requirements, and to give form to our project, we built *TextReq Validation*, a web based software that serves as a proof of concept for our objectives. The software stores ontology models in a relational database (i.e., tables), as well as a system with its requirements. It can do a basic analysis on these requirements and match them against the model's properties, showing which ones are covered and which ones are not.

The software has two main components: The web application that provides the user interfaces, handles the business logic, and manages the storage of models and systems. This component was built using Ruby on Rails (RoR), a framework to create web applications following the Model View Controller pattern [28]. The views and layouts are supported by the front-end framework Bootstrap [29]; these scripts are written using Python.

Figure 10 is collage of elements in the system architecture and application models and controllers. The model-view-controller software architecture for TextReq is shown in the top left-hand schematic. The interface between the web application and the Python scripts is handled through streams of data at a system level. The content of the streams uses a simple *key/value* data structure, properly documented. The right-hand schematic is a UML diagram of the application models. The *models* corresponding to the MVC architecture of the web application, reveal the simple design used to represent an Ontology and a System. The first one consists of a Model – named after an Ontology Model, and not because it is a MVC model – that has many Entities. The Entities, in turn, have many Properties. The latter is even simpler, consisting of only a *System* that has many *System Requirements*. Most of the business logic resides in the models; notice, in particular, system-level interpretation of results from the natural language processing. And finally, the bottom left schematic is a collection of UML diagrams for the application controllers. Due to TextReq's simplicity, its controllers and views are mostly

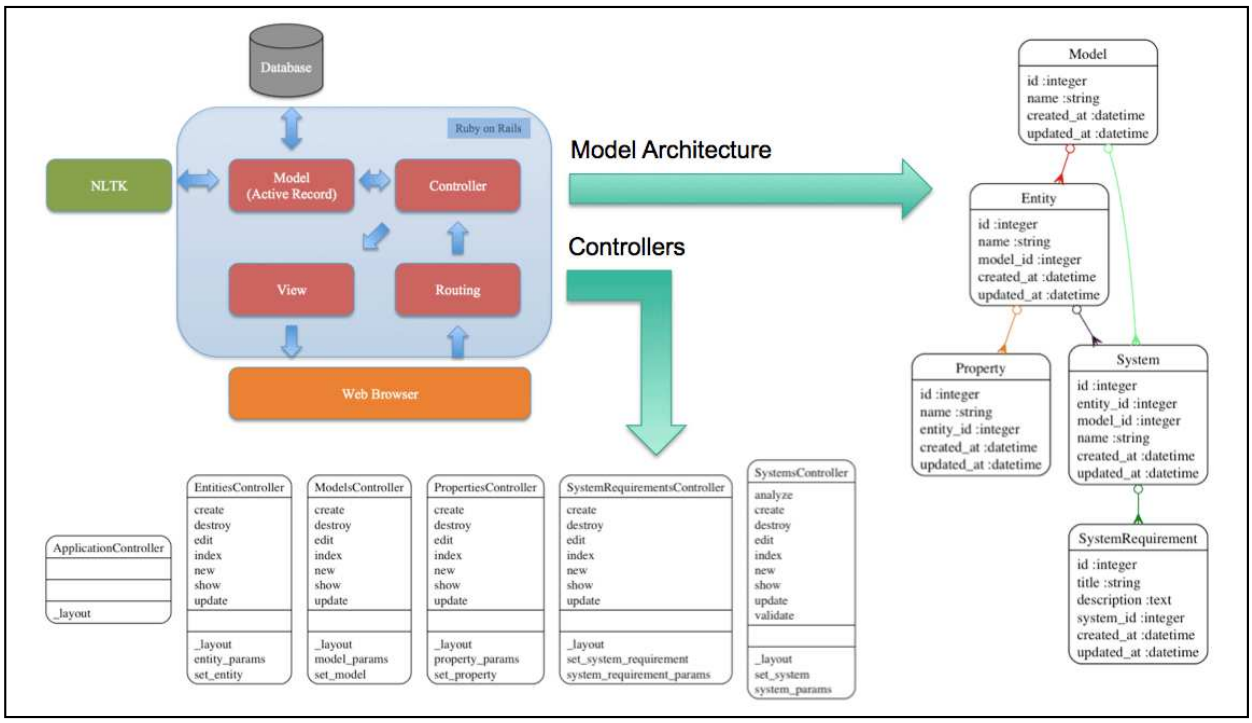

Figure 10. System architecture collage. Top left: Software architecture for TextReq validation. Bottom left: UML diagram of application controllers. Right-hand side: UML diagram of application models.

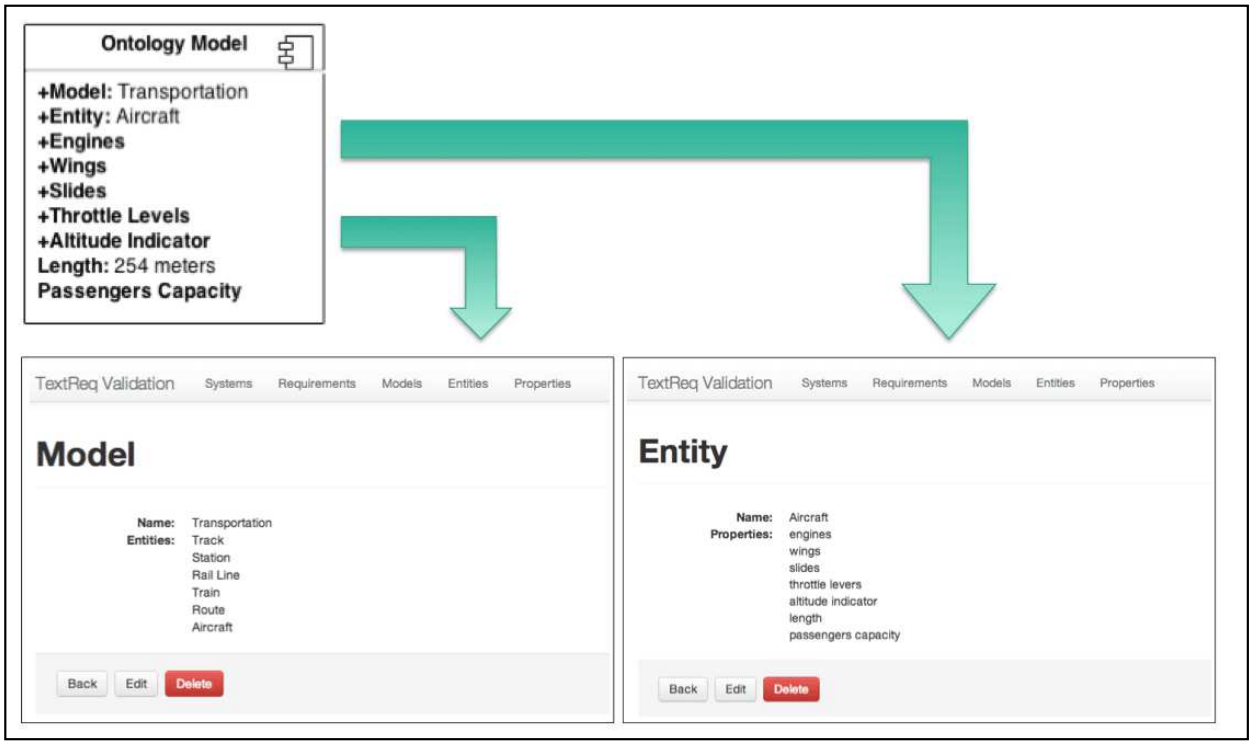

Figure 11. Relationship among aircraft and transportation ontology models, and an aircraft entity model. Top left: Simplified ontology model for an aircraft. Bottom left: Detailed view of the Transportation ontology model. Bottom right: Detailed view for the entity Aircraft.

boilerplate. We have one controller for each part of the model of the application plus an overall "application controller." Each model's controller implements the methods required to handle the client's requests, following a REST (representational, state, transfer) architecture.

The source code for both the web application and the Python scripts are openly hosted in GitHub, in the repository https://github.com/aarellano/textrv.

V. Case Study Problems

We now demonstrate the capabilities of the proposed methodology by working through two case study problems.

Case Study 1: Simple Aircraft Application. We have exercised our ideas in a prototype application, step-by-step development of a simplified aircraft ontology model and a couple of associated textual requirements. The software system requires two inputs: (1) An ontology model that defines what we are designing, and (2) A system defined by its requirements. It is worth noting that while the ontology model and system requirements are unrealistically simple, and deal with only a handful of properties, a key benefit is that we can visualize them easily.

The upper left-hand side of Figure 11 shows the aircraft model we are going to use. We manage a flattened (i.e., tabular) version of a simplified aircraft ontology. This simple ontology suggests usage of a hierarchical model structure, with aircraft properties also being represented by their own specialized ontology models. For instance, an ontology model for the *Wings*, which in turn could have more nested models, along with leaf properties like *length*. Second, it makes sense to include a property in the model even if its value is not set. Naturally, this lacks valuable information, but it does give us the knowledge that that particular property is part of the model, so we can check for its presence.

The step-by-step procedure for using *TextReq Validation* begins with input of the ontology model, then its entities, and finally the properties for each entity. The next step is to create a system model and link it to the ontology. We propose a one-to-one association relationship between the system and an ontology, with more complex relationships handled through hierarchical structures in ontologies. This assumption simplifies development because when we are creating a system we only need to refer to one ontology model and one entity.

The system design is specified through *textual system requirements*. To enter them we need a system, a title and a description. For example, Figure 12 shows all the system Requirements for the system *UMDBus 787*. Notice that each requirement has a title and a description, and it belongs to a specific system. The prototype software has views (details not provided here) to highlight connectivity relationships between the requirements, system model (in this case, a simplified model of a UMDBus 787), and various aircraft ontology models.

Figure 13 is a detailed view of the System UMDBus 787. Besides the usual actions to Edit or Delete a resource, it is important to notice that this view has the *Analyze* and *Validate* actions whose purpose is to trigger the information extraction process described in Section III. The output from these actions is shown in Figures 14 and 15, respectively. The analysis and validation actions match the system's properties taken from its ontology model against information provided in the requirements. In this case study example, the main point to note is that the Aircraft ontology has the property slides (see Figures 11 and 13), but slides is not specified in the textual requirements (see Figure 12). As a result, `slides` shows up as an unverified property in Figure 15.

Case Study 2: Framework for Explicit Representation of Multiple Ontologies. In case study 1, a one-to-one association relationship between the system and an ontology was employed, with more complex relationships handled through hierarchical structures in ontologies. These simplifying assumptions are suitable when we simply want to show that such a simple system setup can work. However, as the number of design requirements and system heterogeneity (i.e., multiple disciplines, multiple physics) increases, the only tractable pathway forward is to make the ontology representations explicit, and to model cause-and-effect dependency relationships among domains in design solutions (i.e., having mixtures of hierarchy and network system structures). While each of the participating disciplines may have a preference toward operating their domain as independently as possible from the other disciplines, achieving target levels of performance and correctness of functionality nearly always requires that disciplines coordinate activities at key points in the system operation. These characteristics are found in a wide range of modern aircraft systems, and they make design a lot more difficult than it used to be.

To see how such an implementation might proceed, Figure 16 illustrates systems validation for requirements covering system-level aircraft specification and detailed wheel system specification. Requirements would be organized into system level requirements (for the main aircraft system) and subsystem level requirements (for the wheels, power systems, and so forth). Full satisfaction of the high-level wheel requirements specification is dependent on lower-level details (e.g., diameter, width, material) being provided for the wheel

VI. Discussion

We have yet to fully test the limits of NLP as applied to requirements engineering. The two case studies presented here demonstrate a framework for using NLP in conjunction with domain ontologies in order to verify requirements coverage. There may be other applications of NLP. A framework for verifying requirements coverage while maintaining consistency by using "requirements templates" has been proposed [30]. For this paradigm, all requirements describing a specific capability must be structured according to a predetermined set of templates. Coverage can then be verified by mapping instances of templates in a set of decomposed requirements to an original list of required capabilities. Figure 17 shows a workflow that combines the requirements template framework with our own. Since the requirements follow templates, it is straightforward for NLP to extract high-level information. Capabilities can then be flowed down for decomposition of each systems requirements. An even further extension of this idea is to use NLP while writing requirements in real time. If an ontology

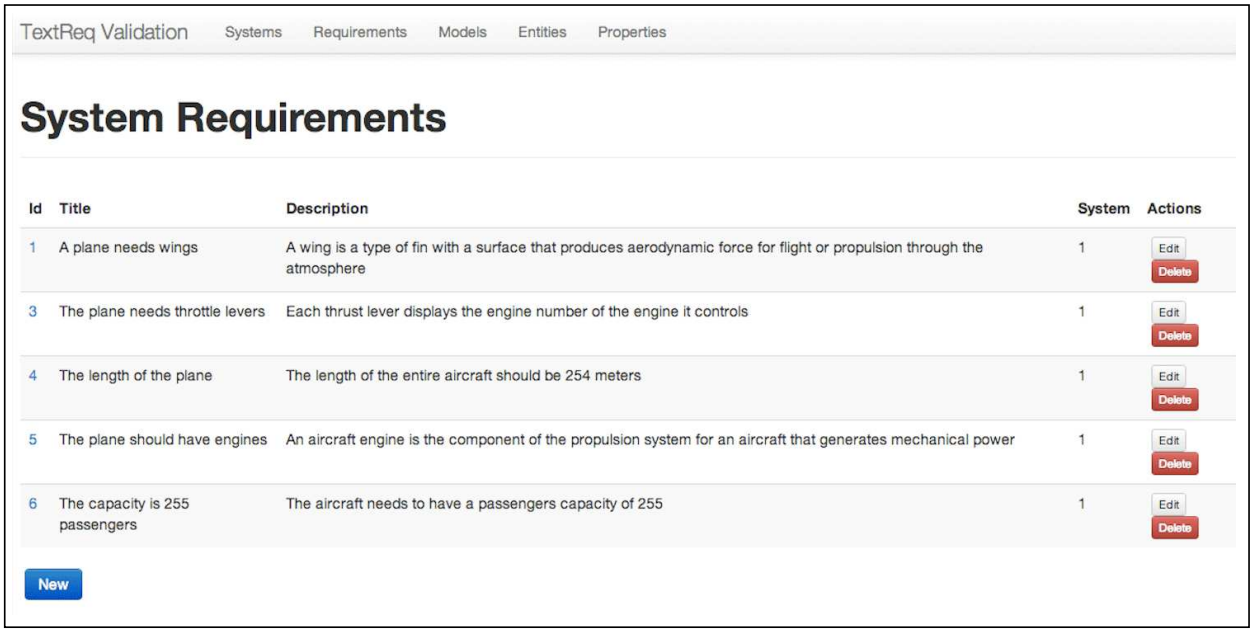

Figure 12. Panel showing all the requirements for the system *UMDBus 787*.

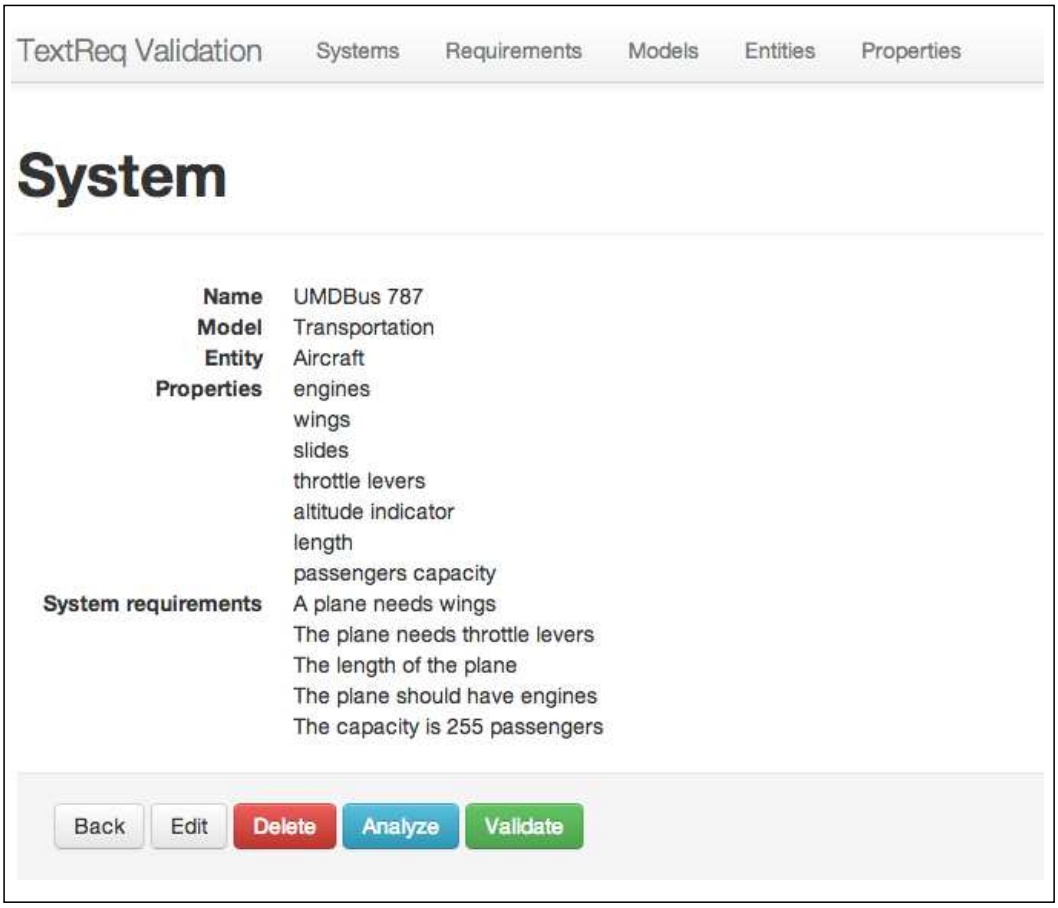

Figure 13. Detailed view for the System *UMDBus 787*.

Basic Properties

Property	Value
Chars	547
Len tokens	94
Sentences	1
Porter stems	94
Lancaster stems	94
Wnl stems	94

Objects

NN	aircraft plane engine capacity length propulsion atmosphere component fin flight force lever number power surface system throttle thrust type
NNS	passengers displays engines generates levers meters wings
NP	aircraft engine engine number generates mechanical power passengers capacity propulsion system throttle levers thrust lever displays

Figure 14. Basic stats from the text, and a list of the entities recognized in it.

System Validation

Verified properties	engines wings throttle levers length passengers capacity
Unverified properties	slides altitude indicator

Figure 15. This is the final output from the application workflow. It shows what properties are verified (i.e., are present in the system requirements) and which ones are not.

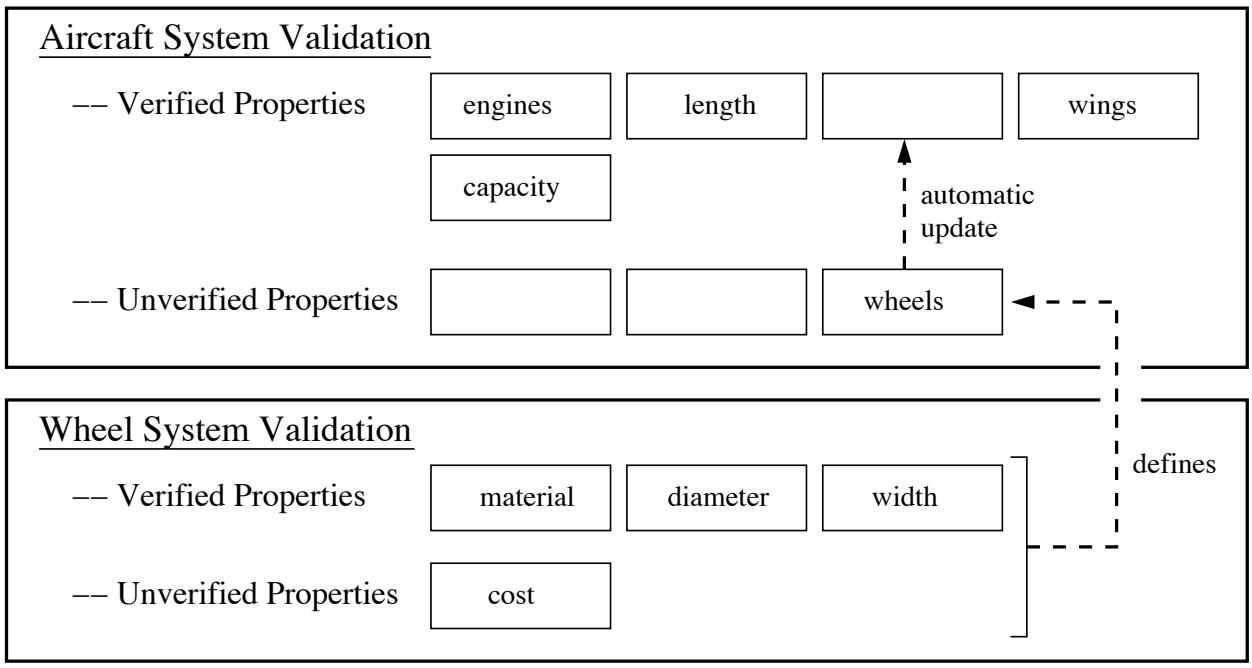

Figure 16. Systems validation for requirements covering system-level aircraft specification and detailed wheel system specification.

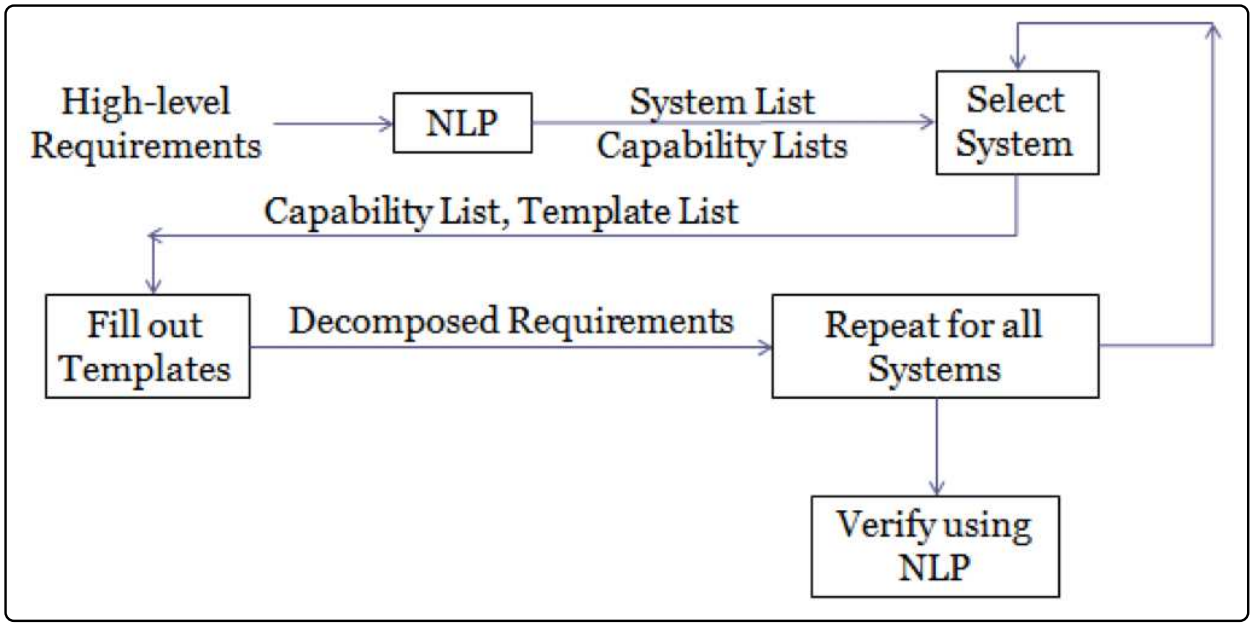

Figure 17. Framework for NLP of textual requirements with templates.

of requirements templates exists, perhaps application-specific NLP could be incorporated into a tool that helps construct and validate requirements as they are written. Some engineers will complain that they are being forced to comply to prescribed standards for writing requirements. Perhaps they will have difficulty in expressing their intent? Our view is that: (1) writing requirements in a manner to satisfy template formats is not much different than being asked to spell check your writing, and (2) the existence of such templates may drastically increase the opportunity for automated transformation of textual requirements into semi-formal models (see Figure 1).

VII. Conclusions and Future Work

When a system is prescribed by a large number of (non formal) textual requirements, the combination of previously defined ontology models and natural language processing techniques can play an important role in validating and verifying a system design. Future work will include formal analysis on the attributes of each property coupled with use of NLP to extract ontology information from a set of requirements. Rigorous automatic domain ontology extraction requires a deep understanding of input text, and so it is fair to say that these techniques are still relatively immature. As noted in Section VI, a second opportunity is the use of NLP techniques in conjunction with a repository of acceptable "template sentence structures" for writing requirements [30]. Finally, there is a strong need for techniques that use the different levels of detail in the requirements specification, and bring ontology models from different domains to validate that the requirements belongs to the supposed domain. This challenge belongs to the NLP area of *classification*.

VIII. Acknowledgments

Financial support to the first author was received from the Fulbright Foundation.

References

[1] A. Arellano, E. Carney, and M.A. Austin, "Natural Language Processing of Textual Requirements," The Tenth International Conference on Systems (ICONS 2015), Barcelona, Spain, April 19–24, 2015, pp. 93–97.

[2] M.A. Austin and J. Baras, "An Introduction to Information-Centric Systems Engineering," Tutorial F06, INCOSE, Toulouse, France, June, 2004.

[3] V. Ambriola and V. Gervasi, "Processing Natural Language Requirements," Proceedings 12th IEEE International Conference Automated Software Engineering, IEEE Computer Society, 1997, pp. 36–45.

[4] C. Rolland and C. Proix, "A Natural Language Approach for Requirements Engineering," Advanced Information Systems Engineering, Springer, 1992, pp. 257–277.

[5] K. Ryan, "The Role of Natural Language in Requirements Engineering," Proceedings of the IEEE International Symposium on Requirements Engineering, IEEE Comput. Soc. Press, 1993, pp. 240–242.

[6] M.A. Austin, V. Mayank, and N. Shmunis, "PaladinRM: Graph-Based Visualization of Requirements Organized for Team-Based Design," Systems Engineering: The Journal of the International Council on Systems Engineering, Vol. 9, No. 2, May, 2006, pp. 129–145.

[7] M.A. Austin, V. Mayank, and N. Shmunis, "Ontology-Based Validation of Connectivity Relationships in a Home Theater System," 21st International Journal of Intelligent Systems, Vol. 21, No. 10, October, 2006, pp. 1111–1125.

[8] N. Nassar and M.A. Austin, "Model-Based Systems Engineering Design and Trade-Off Analysis with RDF Graphs," 11th Annual Conference on Systems Engineering Research (CSER 2013), Georgia Institute of Technology, Atlanta, GA, March 19–22, 2013.

[9] P. Delgoshaei, M.A. Austin, and D. A. Veronica, "A Semantic Platform Infrastructure for Requirements Traceability and System Assessment," The Ninth International Conference on Systems (ICONS2014), Nice, France, February 23–27, 2014, pp. 215–219.

[10] P. Delgoshaei, M.A. Austin, and A. Pertzborn, "A Semantic Framework for Modeling and Simulation of Cyber-Physical Systems," International Journal On Advances in Systems and Measurements, Vol. 7, No. 3-4, December, 2014, pp. 223–238.

[11] S. Fridenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML*. MK-OMG, 2008.

[12] K. Kageura and B. Umino, "Methods of Automatic Term Recognition: A Review," Terminology, Vol. 3, No. 2, 1996, pp. 259-289.

[13] L.L. Earl, "Experiments in Automatic Extracting and Indexing," Information Storage and Retrieval, Vol. 6, No. 6, 1970, pp. 273–288.

[14] S. Ananiadou, "A Methodology for Automatic Term Recognition," Proceedings of 15th International Conference on Computational Linguistics (COLING94), 1994, pp. 1034-1038.

[15] K Frantzi, S. Ananiadou, and H. Mima, "Automatic Recognition of Multi-Word Terms: The C-Value/NC-Value Method," International Journal on Digital Libraries, Vol. 3, No. 2., 2000, pp. 115-130.

[16] D. Fedorenko, N. Astrakhantsev, and D. Turdakov, "Automatic Recognition of Domain-Specific Terms: An Experimental Evaluation," Proceedings of SYRCoDIS 2013, 2013, pp. 15-23.

[17] A. Judea, H. Schutze, and S. Bruegmann, "Unsupervised Training Set Generation for Automatic Acquisition of Technical Terminology in Patents," Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers, Dublin, Ireland: Dublin City University and Association for Computational Linguistics, 2014, pp. 290-300.

[18] L. Kozakov, Y. Park, T. Fin, et al., "Glossary Extraction and Utilization in the Information Search and Delivery System for IBM Technical Support," IBM Systems Journal, Vol. 43, No. 3, 2004, pp. 546-563.

[19] NLTK Project, "Natural Language Toolkit NLTK 3.0 Documentation," See http://www.nltk.org/ (Accessed: December 1, 2015).

[20] C. Manning and H. Schuetze, "Foundations of Statistical Natural Language Processing," The MIT Press, 2012.

[21] M.F. Porter, "An Algorithm for Suffix Stripping," Program: Electronic Library and Information Systems, MCB UP Ltd, Vol. 14, No. 3, 1980, pp. 130–137.

[22] C.D. Paice, "Another Stemmer," ACM SIGIR Forum, ACM, See: http://dl.acm.org/citation.cfm?id=101306.101310, Vol. 24, No. 3, November, 1990, pp. 56–61.

[23] Princeton University, "About WordNet - WordNet - About WordNet," See https://wordnet.princeton.edu/ (Accessed: December 1, 2015).

[24] M. Haspelmath, "Word Classes and Parts of Speech," See http://philpapers.org/rec/HASWCA, (Accessed: December 1, 2015).

[25] S. Bird, E. Klein, and E. Loper, "Natural Language Processing with Python," O'Reilly Media, Inc., 2009.

[26] University of Pennsylvania, Penn Treebank Project, See http://www.cis.upenn.edu/ treebank/ (Accessed, December 1, 2015).

[27] B. Santorini, "Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision)," Technical Reports (CIS), http://repository.upenn.edu/cis_reports/570, 1990.

[28] Ruby on Rails. See http://rubyonrails.org/ (Accessed, December 1, 2015).

[29] Bootstrap. See http://getbootstrap.com/2.3.2/ (Accessed, December 1, 2015).

[30] E. Hull, K. Jackson and J. Dick, Requirements Engineering, Springer, 2002.