Two Rule-Based Natural Language Strategies for Requirements Discovery and Classification in Open Source Software Development Projects

RADU E. VLAS AND WILLIAM N. ROBINSON

RADU E. VLAS is a Ph.D. candidate and instructor in the Department of Computer Information Systems at the J. Mack Robinson College of Business at Georgia State University. His background spans the worlds of technology (he holds a B.S. in electrical engineering and computer science from the Polytechnic University of Bucharest, and an M.S. in computer science from Indiana University) and business (MBA from the University of Nebraska at Omaha). His interdisciplinary research and teaching interests are in the areas of software development, requirements engineering, software quality, open source software, social media and social networks, effects of technology on human behavior, and human perceptions of technology. He was RE’09 Demos Chair and has been session chair and reviewer for various journals and conferences.

WILLIAM N. ROBINSON is an associate professor in the Department of Computer Information Systems and associate of the Center for Process Innovation (CEPRIN) at the J. Mack Robinson College of Business at Georgia State University. He holds degrees from Oregon State University (B.S. in computer science) and the University of Oregon (M.S. and Ph.D. in computer information systems). Requirements and policies of information systems are the focus of his research, which includes analyses of supply chain policies, requirements interactions and negotiations, and runtime requirements monitoring. He directs the Requirements Analysis and Monitoring Lab (RAML), supported by the National Science Foundation and industry. Dr. Robinson has authored over 60 publications, including ACM, IEEE, Springer, and Taylor & Francis transactions and magazines. He serves the Requirements Engineering community as RE’99 program chair, RE’09 general chair, editor-in-chief of Requirements Engineering, and member of the International Federation for Information Processing (IFIP) Working Group 2.9 on Requirements Engineering.

ABSTRACT: Open source projects do have requirements; they are, however, mostly informal text descriptions found in requests, forums, and other correspondence. Understanding such requirements provides insight into the nature of open source projects. Unfortunately, manual analysis of natural language requirements is time-consuming, and for large projects, error prone. Automated analysis of natural language requirements, even partial, will be of great benefit. Toward that end, we describe the design and validation of an automated natural language requirements classifier for open source projects. We compare two strategies for recognizing requirements in open forums of software features. Our results suggest that classifying text at the forum postaggregation and sentence aggregation levels may be effective. Our results suggest that it can reduce the effort required to analyze requirements of open source projects.
The increasing development and use of open source software (OSS) has gained the interest of researchers [27, 43]. OSS development (OSSD) appears to produce high-quality software with fewer resources than more traditional approaches. Analysis of OSS management, membership, processes, and products may lead to improvements in all software development. Studies of OSS requirements provide a means for systematically analyzing OSSD. Therefore, we are creating tools to aid in the automated analysis of OSS requirements. Automation is critical because there are thousands of projects having hundreds to thousands of requirements.

Many OSS projects are successful [10, 54]. In OSSD, the software product is developed, distributed, and supported by users. Common characteristics are (1) many developers, (2) volunteering rather than delegating, (3) limited emphasis on design activities, and (4) few plans, list of deliverables, or timelines [43].

In OSSD, developers are also potential users of the product. They are stakeholders expressing needs that define the system requirements [17]. It may appear that the requirements analysis stage is absent. However, Scacchi has identified software informalisms, which are “the information resources and artifacts that participants use to describe, proscribe, or prescribe what’s happening in an OSS project” [53, p. 473]. Scacchi identifies two dozen types of software informalisms, which include chats, e-mail, forums, and project digests. By analyzing these unstructured, informal, natural language artifacts, one can better understand the requirements, and thus OSSD.

Consider Figure 1, which presents a feature request, a kind of requirement, from the feature tracker of the Password Safe project on SourceForge. The Password Safe project has 630 feature requests, 976 bug reports, and thousands of forum posts. To understand OSSD, researchers need to characterize such communications. Such informalisms are used to manage projects. Their analysis provides insight into the best practices of OSSD.

Requirements traceability is a guiding theory in this line of research [22, 47]. “Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction” [22, p. 94]. By following the life cycle of requirements, we can follow techniques and infer the strategies of development [25]. Ideally, we can discover practices that distinguish successful from unsuccessful projects.

Requirements can be considered from three simple levels:

1. Requirements metrics count individual requirements, including their total number, individual versions, their classifications, and their trace links (types and numbers, including contribution structures [23], e.g., roles, relationships, responsibilities).
2. Requirements management metrics count collections of requirements, including requirements snapshots (the collective versions), their temporal relationships, and trace links (e.g., a snapshot’s association to a code release).

3. Requirements management models specify the associations among artifacts as well as the process model for their creation and management [46].

Advanced developers apply all three levels to development [46]. Our goal is to build successively more capable tools for discovering these concepts from the natural forms found in OSSD. In so doing, we aim to understand the evolving requirements management models (RMMs) of OSSD.

If we had OSSD RMMs now, we could answer questions such as the following:

- What kinds and numbers of requirements are associated with successful projects? quality projects? secure projects? and so forth.
- What are common ratios concerning requirements, resources, and release rates?
- How do the kinds of requirements vary with the size or type of software?
- For a given size or type of software, which RMMs are most successful?

Precise, empirically derived OSSD RMMs describe current practices and suggest best practices for software requirements management.

The research presented here provides an initial step toward automated requirements discovery. Given an OSSD project, our tool applies a natural language parsing approach to
• Identify requirements
• Classify requirements

Our software tool is the Requirements Classifier for Natural Language (RCNL). The following research question defines the objectives of this study: How can the automated requirements discovery and classification be achieved for the natural requirements documents of OSSD projects?

Research Questions

Application of requirements traceability practices improves the likelihood of software project success, specifically through improved quality, functionality, and timely releases. Traceability links record the history of artifact development, from high-level requirements descriptions to the lower-level programming codes. From the artifact traces, one can infer the development tasks used to construct the artifacts. A trace link from source code to binary code implies the task of applying a compiler. A trace link from a feature requirement to a use-case implies the task of use-case specification. Development traces imply the development process model. Sometimes developers apply a process model that varies from their stated process model. Analyzing development traces reveals the actual process model used. A software tool to aid the discovery, modeling, and analysis of development traces will help in analyzing software development [30]. If such a tool were to exist, then it could be applied to existing software projects to aid empirical studies. For example, according to information systems development (ISD) theory, development process models vary in their success—some process models work well and others less so, depending on a variety of factors, such as project type and personnel. A tool for discovery, modeling, and analysis of artifact traces will help to develop ISD process theory. Discovered models can be analyzed according to a variety of success factors, such as quality, functionality, and release periodicity.

This paper is most concerned with the requirements discovery and classification task within traceability theory. Requirements discovery is a prerequisite to developing requirements traceability tools. To date, there is no software tool that can identify and classify requirements from open source requirements documents. Open issues include (1) How is a single requirement recognized and delimited? (2) Given the text of a requirement, how is it classified? and (3) How is a recognized requirement related to another recognized requirement? Solving these problems will provide for a software tool that can automatically review natural text documents and create identified requirements, their classification, and associated traceability links. Such a software tool is a prerequisite to a comprehensive tool for discovery, modeling, and analysis of development traces, which in turn will support empirical analysis of ISD process theory.

With this introduction, we now present a design science hypothesis for natural language requirements discovery:

*Hypothesis 1: The automated discovery and classification of requirements contained within natural requirements documents of OSSD projects can be achieved*
through the design of a requirements analysis process and the development of a software artifact to implement it.

Related research suggests that grammar-based analysis may provide an efficient requirements recognizer. Ideally, the recognizer should work for any natural language document. However, natural language processing (NLP) analyses perform better when they are customized to a document corpus. Thus, our refined design science hypothesis addresses the issues of a language with the specialized sublanguage as expressed by a subculture (such as OSSD):

Hypothesis 1a: A multilayered grammar, varying in domain specificity, can be constructed for the automated requirements discovery and classification of requirements contained within natural requirements documents of OSSD projects.

We apply a design science approach to address these questions. Because it is a design science project, we are also concerned with how well our design works. Therefore, we compare the automated results with the results of an idealized perfect recognizer.

Research Method

We follow the design science approach [26] in developing our RCNL classifier. Design science provides guidelines for research on designed artifacts [26]. We apply the Hevner et al. [26] descriptive approach to design science—developing and then evaluating the design of RCNL using case studies, performance analysis, and argumentation. In addition, we compare the results of automated classification with that of an expert.

Much of information systems (IS) research follows the behavioral science paradigm, in which researchers aim to understand phenomena related to the development and use of IS. In recent years, the design science paradigm has grown, with researchers developing IS artifacts and improving their performance. March and Smith [39] specify four activities (theorize, justify, build, and evaluate) for conducting IS research, with behavioral science addressing the first two activities and design science addressing the last two. The activities occasionally apply the same methods, such as a controlled experiment for behavioral theory justification and for design science artifact evaluation. The two paradigms are mutually supportive in that the results in one approach can provide for new research designs in the other [5].

Multiparadigm research represented only about 20 percent of IS research studies in the 1990s [41]. However, it has been argued that a multiparadigm research provides for broader and more conclusive explanations [5]. This project is design science research in that we build and evaluate a designed artifact. The design stems from the general theories of IS development and traceability management, as well as the technical theories of concept tagging, grammar-based parsing, and classification. This single study is a design science part of a larger multiparadigm research project. Subsequently, the RCNL technology will be used to obtain data for empirical analysis of OSSD. The larger project goals are to extend IS development process theory to explain the unique characteristics of OSSD.
Recognizing Requirements

OSSD requirements take many forms, most of which are represented as natural language text [53]. For each form, there are many requirements. For example, the KeePass project has 1,522 feature requests, 923 bug reports, and thousands of other various forum posts. Cleland-Huang et al. [7] found that forums are filled with thousands of requirements, as well as thousands of lines that are not requirements—for example, social communications, code segments, slang, and typos [53]. Thus, requirements discovery is first about delimiting each requirement within its source. Once requirements are identified, then subsequent processing can begin.

Consider three strategies for recognizing a requirement:

1. Grammar-based strategy: Text is parsed according to a grammar. The grammar defines what text is a requirement. For example, our Subject-Action-Object (SAO) grammar tags each SAO-triple as a requirement. This strategy implements the patterns commonly characterizing formal requirements specifications, in which each requirement is expected to clearly state a subject, an action, and an object. The subject is the actor in the requirements statement. The action determines the feature being described in the requirement. The object is the entity being impacted by the action performed by the actor. For example, let us consider the following requirement statement: “the submit form button should send the form data to the processing component.” In this example, the subject is “the submit form button,” the action that should take place is “should send,” and the object affected by the action is the “form data.” The words following this SAO pattern provide additional explanatory context for the action of the requirement.

2. Delimiter-based strategy: Text is split into segments according to delimiters, which may be key words or phrases. The text between the delimiters is tagged as a requirement. In OSSD feature request forums, posts are very informal. Each post commonly addresses one or a small number of ideas or suggestions. The posts often include a variety of sentences and phrases providing context to the idea(s) presented. Sometimes, the feature being suggested is not clearly specified but, rather, is implied or described without ever being expressed in a concise statement. Given these facts, our delimiter strategy considers each forum posting as a discussion around a limited number of features of interest separated by certain delimiters. Therefore, each feature request post is a requirement if no delimiters are identified within the post. In order to distinguish among topics discussed in a posting, we use a library of requirements separators (key words or short expressions). Such delimiters determine the delimited requirements present in a feature request post.

3. Hybrid strategy: First, the delimiter-based strategy is applied. Then, each requirement text is parsed with the grammar. This allows for the recognition of an aggregate requirement and its supporting subrequirements. (Of course, if the text includes hierarchal requirements numbers, e.g., 1, 1.1, 1.1.1, then such numbers can be used for requirements groupings—unfortunately, this is less common in the open source domain.)
Unrecognizable text affects how each strategy performs. If the grammar completely characterizes the text, then grammar and delimiter strategies will tag the same text segments as requirements. More commonly, the grammar only partially characterizes the text. Thus, the unrecognized text, preceding or following an SAO-triple, for example, will not be tagged as being within a requirement. Using the delimiter-based strategy can provide a more natural tagging, as recognized by analysts. Finally, the hybrid strategy provides the best of both—an entire text segment tagged as a requirement, and its parts characterized according to a grammar. Herein, we report on a comparison between a grammar-based and a delimiter-based strategy for recognizing and classifying requirements from the SourceForge feature tracker.

Figure 2 shows the result of applying a grammar-based parsing strategy using the RCNL. The highlighted text has been parsed as grammar fragments, recognized as requirements, and classified according to a requirements ontology [58]. Notice that some text is not considered to be part of any requirement, and thus the text is not highlighted. The seemingly irrelevant text includes the feature identifier (number), the UNIX date the feature was posted (number), as well as phrases such as ”Thanks in advance for the Developers consideration” (note the presence of typos and poor grammar—one of the prominent challenges in analyzing natural language data from open source requirements).

Classifying Requirements

Requirements engineering theory specifies measures that can guide the analysis of development. For example, one would expect that the specification of a secure operating system would have many kinds of security requirements. Their absence would be a cause for concern. Thus, requirements classification helps requirements management by determining the presence and proportion of various requirement types.
Requirements classifications provide taxonomies of common kinds of requirements. Reliability, efficiency, integrity, and usability are common requirements classes. Quality models, such as those of Boehm et al. [3], IEEE [29], McCall et al. [40], and ISO, specify the characteristics of requirements belonging to a class. These characteristics can be used to classify requirements.

McCall et al.’s [40] software quality model is widely accepted in both researcher and practitioner communities. Their model, like others, specifies words and phrases that are indicative of requirements belonging to a classification. For example, a requirement that includes the word *faster* or *slow* is indicative of a performance requirement. Using such key words, the technique of *key word classification* uses libraries of key words, phrases, and grammar fragments to match against delimited text requirements.

A classification problem arises from the length of delimited requirements:

1. The shorter the word length, the less likely a requirement will be classified, because classification is based on the contained words. This leads to many recognized but unclassified requirements.
2. The longer the word length, the more likely a requirement will be classified; however, it is also more likely that two conceptually unrelated ideas will be commingled and considered as one requirement. This leads to fewer total recognized requirements, each of which is likely to be classified at least once.

The issues of requirements recognition (or delimiting) and classification are interrelated. This raises the issue of what exactly is a natural language requirement? Open source requirements include an unusual amount of extraneous language and symbols and improper syntax. We address the questions of recognition and classification in open source requirements.

We have designed, developed, and applied the RCNL for text-based requirements analysis. A key element of RCNL is its multilevel ontology, in which the lower, specific levels apply generic English grammar-based concepts while the upper, abstract levels apply OSS requirements-based concepts. The experimental application of the RCNL is limited to the text found in work items of SourceForge’s Feature Tracker, which includes a large number of forum posts [35].

This research study provides three contributions:

1. a grammar-based design of software automation for the discovery and classification of natural language requirements,
2. two alternative parsing schemes implemented within the design, and
3. requirements discovery, classification, and analysis of 30 OSS projects.

Together, these contributions provide a path for subsequent empirical studies of OSS requirements and enable subsequent software tools for facilitating automation of requirements traceability analysis in support of IS development process studies.

**Usage Scenarios**

We present two RCNL usage scenarios to provide context. First, consider usage by an academic, Jane, studying OSSD. Jane can apply RCNL to OSS projects to obtain
metrics on the quantity and classification of requirements. Identifying text segments as requirements and their classifications are open to interpretation, even among experts. Thus, Jane may choose to review the requirements and the classifications produced by RCNL. She may even alter RCNL rules to suit her interpretation. Once satisfied with the results, Jane can compare project requirement metrics and correlate those metrics with success or other factors of interest. Second, consider John, an analyst for an OSS company. He can apply RCNL to the thousands of forum postings he receives monthly. By continually monitoring the quantity and classification of requirements posted, John can maintain an overview of the kinds of concerns his users are expressing. In both scenarios, manual analysis of a million words is not practical. It is too time-consuming, and classification requires expertise, which is too costly. Software such as RCNL enables an analyst to get a quick overview of OSS requirements.

Herein we describe the design (in the third section) and engineering (in the fourth section) of RCNL and experiments measuring its capabilities (in the fifth section). These results suggest that the parsing strategies and toolkit may be useful in classifying requirements in OSSD projects.

Related Research

Requirements in OSSD

Scacchi’s [52] study on OSSD requirements shows that informal communication, rather than classical formal modeling, is common. Requirements emerge through a dynamic social process. The informalism concept is strengthened by acknowledgment of the importance of discussion forums as a means of reaching common understanding and acceptance of requirements [35].

In a study of nonfunctional requirements (NFRs), Cleland-Huang et al. [8] use a semiautomated technique for identification and classification of requirements from both structured and unstructured documents. The NFR classification process proposed has three stages: mining phase, classification phase, and application phase. In the mining phase, the authors perform term extraction based on a training set for identification of key words. The classification phase enhances this process by performing a sentence extraction and NFR classification from the available documents based on three resources: the terms extracted during the first phase, a document of unclassified software requirements specifications, and unstructured documents listing potential NFRs. The application phase makes use of the classified requirements in subsequent software engineering activities. This semiautomated method requires a researcher’s intervention and control throughout the entire process.

A number of researchers highlight the benefits and appropriateness of using NLP in requirements engineering. Ryan [50] highlighted, in the early 1990s, the increased need of NLP as a reaction to the increased complexity of software systems. Other studies suggest more specific uses of NLP. Sampaio presents an NLP technique based on Wmatrix for analyzing requirements documents with the intent of identifying aspects specific to Aspect-Oriented Software Development (AOSD). The approach can explore both structured and unstructured documents. The identification process is supervised
and controlled by a researcher and generates a structured document. The identification process is based on frequency analysis and key term extraction [51].

Ambriola and Gervasi [1] confirm the efficiency of using of NLP in requirements analysis. Their requirements modeling and analysis framework (CIRCE) is designed for the analysis of structured documents in which requirements are expressed through natural language text. CIRCE parses input data and applies a number of modelers in order to present the analyst with rendered (UML [Unified Modeling Language]-based) views of the model being analyzed [1].

Fantechi and Spinicci [15] propose a semiautomated process for improving requirements quality through reduction of inconsistencies. Their process also uses NLP; Phrasys is used for phrase and sentence extraction. The proposed technique processes structured requirements documents into SAO-triples in order to analyze interactions between entities [15]. Kof [32] uses SAO patterns for analyzing structured requirements documents also. His goal is to identify associations between terms and construct domain taxonomies.

Classification of Requirements

Requirements have been traditionally classified as either functional (FR) or nonfunctional (NFR), even though some researchers consider this classification to be too broad [2]. When adopting this perspective, researchers refer to FRs as goals (or hard goals, or behavioral requirements), and to NFR as soft goals [45, 57]. FRs are concerned with specifying particular features of the system to be developed. Therefore, a complete set of FRs should comprehensively describe the functionality of the new system. NFRs are concerned with two areas: (1) properties that affect the system as a whole (e.g., usability, portability, maintainability, or flexibility), and (2) quality attributes (e.g., accuracy, response time, reliability, robustness, or security) [6, 44]. Some variations to it include listing of security concerns under FR, adding supportability under NFR, or specifying subcategories of these two [24].

Additional requirements classifications start from an agent-based perspective informing the V-model of requirements and listing them as user-stakeholder, system, subsystem, or component requirements [4, 28, 59]. From a goal-orientation perspective, goals are identified and analyzed based on the agents that can achieve them. From an agent-oriented perspective, the agents are identified and analyzed based on the goals they need to achieve [57].

Requirements Patterns

Toro et al. [56] propose a series of requirements templates that can help capture requirements [9, 19, 49]. They introduce linguistic patterns (L-patterns, natural language commonly used for describing requirements) and requirements patterns (R-patterns, generic requirements templates) [55]. While attempting to bridge between natural language and formal requirements specification, this study reaches a middle ground
between them. Respecification, rather than requirements discovery, is the focus of the technique.

Konrad and Cheng [33] propose a set of requirements patterns for embedded systems. Their work does not address requirements discovery but explores requirements patterns identification from existing project requirements. They validate the initial patterns by applying them to two case studies in order to inform future design decisions [33]. Also for embedded systems, Denger et al. [12] address the problem of requirements imprecision. The study uses a pattern-based analysis of existing requirements in order to identify missing information and to fix inconsistencies. Like the other studies, it explores a specific domain (embedded systems) and does not address the problem of requirements discovery. However, the patterns identified are derived from elements of natural language.

Requirements Analysis Tools

A number of other tools provide automatic or semiautomatic requirements analysis processes that start from a data set of already elicited but not completely or properly specified requirements. Among them, we list the Requirements Elicitation, Capture and Analysis Process (RECAP) [13], the Conceptual Modeling Environment/Process Implementation Methodology (ACME/PRIME) [16], the First Requirements Elucidator Demonstration (FRED) [31], and the Requirements Elicitation Assistance System (REAS) [14].

Despite being a tool that provides minimal requirements elicitation support, the AbstFinder is an interesting example of the way NLP techniques can be used in the analysis of requirements-rich and informal textual data [21]. Another example of the way NLP techniques can be used in the process of requirements discovery is provided by the Requirements Elicitor proposed in 2006 by Mala and Uma [37]. Their tool implements a set of 12 syntactic structures based on the subject-verb-object (SVO) pattern in order to transform normalized sentences into message records. These 12 syntactic structures represent the set of possible versions of the SVO pattern as encountered in natural language. Our RCNL tool implements a slightly wider set of SVO structures.

Classifier Design

OPEN SOURCE DOCUMENTS ARE A PROBLEM FOR CLASSIFICATION. The data sources are real unstructured natural language text, which typically does not conform to standard English grammar. In fact, nearly all the texts are fragments containing many typos, misspellings, and idioms (e.g., text smileys).

Our classifier is a kind of weak ontology-based information extraction (OBIE) system. Such systems work well for parsing and tagging fragments of unstructured natural language text [60]. The ontology is comprised of grammar concepts, except for the last level, which contains the requirements classification concepts.
We validate the classifier using two methods. First, we apply the classifier to a variety of data sources and measure its classification quality. Second, we employ the information retrieval (IR) evaluation process, which consists of comparing the tool’s results with those of a human expert and then interpreting a set of established evaluation metrics. We report on these validation efforts in the fifth section. Next, we present the classification ontology.

Requirements Parsing Ontology

The RCNL ontology for OSSD projects includes six levels, as summarized in Table 1. The first two levels contain common natural language grammar concepts. The next three levels contain concepts of logical statements. The final level, 5, contains the classification concepts. Although the levels are numbered from 0 through 5, each level may be only partially dependent on the lower levels. For example, Level 3-G characterizes subjects, actions, and objects using information from Levels 0 and 1, but it does not depend on Level 2.

The first two levels are common to all natural language parsing systems. Level 0 (L0) concepts include words, punctuation, as well as idioms common to OSSD projects, such as e-mail address, URL, and file reference. Level 1 (L1) concepts are the common English parts of speech (POS).

Level 2 (L2) concepts are qualifiers, such as belief, preference, necessity, quantity, and so forth. These mostly depend on specific words identified in Level 0, but may also depend on the POS of Level 1 (e.g., determiners, deictics, quantifiers, numerals, modals, tenses, negators).

Consider the following text from our data set:

\[\text{I think this is great if awstats html tag can calculate ROI.}\]

The key word “think” indicates the presence of an expression of belief. Any beliefs, preferences, or quantifiers that modify a requirement are tagged at L2. Such phrases usually introduce a requirement.

The parsing strategies rely on Level 3 (L3-G and L3-D). Level 3 (L3) concepts are simply subject, action, and object. Subject is not simply a noun, but an actor (person, an object, or a concept). The actor may execute some action on an object. The action is expressed through a verb or set of verbs defining the desired course of events. The object of this action can be any entity in the environment impacted by the performing of the action.

The analysis of requirements through the lenses of a structured approach built on the SAO-triple is not new. Fantechi and Spinicci [15] use this approach for analyzing interactions among entities in a semiautomated process of reducing requirements inconsistencies in structured requirements documents. Our Level 3 (L3-G) patterns discover all subjects, actions, and objects present in text that can potentially be part of a requirement.
## Table 1. The RCNL Requirements Parsing Ontology

<table>
<thead>
<tr>
<th>RCNL  Level</th>
<th>G-grammar</th>
<th>D-delimiter</th>
<th>Level name</th>
<th>Description</th>
<th>Elements covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0-G,D</td>
<td>G,D</td>
<td></td>
<td>Token</td>
<td>Defines basic elements of text commonly included in all types of communication.</td>
<td>Word, punctuation, symbol, list, filename, sentence, e-mail address, URL, phrase, syntactical separator</td>
</tr>
<tr>
<td>L1-G,D</td>
<td>POS</td>
<td></td>
<td>POS</td>
<td>Defines most common parts-of-speech (POS) elements.</td>
<td>Adjective, adverb, verb, conjunction, preposition, determiner, negator, noun</td>
</tr>
<tr>
<td>L2-G,D</td>
<td>Qualification</td>
<td></td>
<td>Qualification</td>
<td>Identifies expressions defining a context that can indicate a requirement.</td>
<td>Belief, certainty, necessity, preference, qualifier, quantifier, qualifying phrase</td>
</tr>
<tr>
<td>L3-G</td>
<td>Entities</td>
<td></td>
<td>Entities</td>
<td>Identifies the three basic elements of a requirement.</td>
<td>Subject (S)/actor, action (A)/verb, object (O)</td>
</tr>
<tr>
<td>L3-D</td>
<td>Concept separators</td>
<td></td>
<td>Concept separators</td>
<td>Identifies pieces of text separating logical concepts.</td>
<td>Semantic separators</td>
</tr>
<tr>
<td>L4-G</td>
<td>Microrequirement</td>
<td></td>
<td>Microrequirement</td>
<td>Discovers parts of text identified as micro-requirements.</td>
<td>SAO-triples, SAO extensions, SAO atomic elements</td>
</tr>
<tr>
<td>L4-D</td>
<td>Macrorequirement</td>
<td></td>
<td>Macrorequirement</td>
<td>Discovers parts of text identified as macro-requirements.</td>
<td>Sets of statements delimited by concept separators (L3-D)</td>
</tr>
<tr>
<td>L5-G,D</td>
<td>Classification</td>
<td></td>
<td>Classification</td>
<td>Classifies pieces of text identified at previous level and elements of lists.</td>
<td>SAO-triples, SAO extensions, list items and introductory phrases, macrorequirements</td>
</tr>
</tbody>
</table>
An SAO assertion is the concept of Level 4 (L4-G). Adjectives, adverbs, and other elements may be involved, but Level 4 represents the central requirements statement. Often, a Level 4 requirement is qualified by a Level 2 expression.

In the L4-G patterns, the existence of a subject and object in text is optional. For example, the following text (from our data set) is tagged as two L4-G requirements (separated by the “,” but”):

Keep the current view of top keywords, but add a new option to display the following information.

The L4-G patterns can reference the other levels. In fact, the qualifier tags of L2 often introduce the L4-G requirement. This example (from our data set) illustrates:

I want to see when I get more visitors, and be able to compare my traffic to other days.

The expression “I want” (L2) qualifies the requirement (L4-G) that it introduces.

Level 4 represents each text requirement. This level is used to specify our two parsing strategies:

1. The SAO-based recognizer combines a subject, an action, and an object into an SAO-triple and enhances it with an optional qualifier or a sequence of objects ending the triple. We call SAO-recognized requirements microrequirements.
2. The delimiter-based recognizer relies on tokens and parts of speech to recognize macrorequirement delimiters. (A delimiter-based requirement can include multiple microrequirements.) If the preceding requirement (“I want to see . . .”) was all the text in a posting, then it would be recognized as a requirement. However, if it were included with other statements with no intervening delimiters, then the set of statements would be recognized as the requirement. The delimiter-based recognizer mostly considers each posting to the SourceForge feature tracker as a requirement. There are a few exceptions, in which key words (e.g., first, second, third, . . . , conversely, in addition to) separate a post into more than one requirement.

Finally, Level 5 (L5) concepts are the domain-specific classification of the Level 4 statements. We have designed two L5 classifiers: (1) standard McCall’s quality model, and (2) extended McCall’s (McCall+) for OSSD projects.

McCall’s quality model specifies 23 quality criteria for software. These concepts are represented in L5. In particular, we specify rules for recognizing the 23 quality criteria using the annotations of Level 4. We aimed to represent accurately the quality model as specified by McCall et al. [40].

In addition to McCall’s classification rules, we specified our own classification rules for McCall’s 23 quality criteria. In particular, these extend McCall’s classification rules to recognize concepts and terms unmentioned in the McCall specification. We call this the OSSD extension of McCall’s model, or McCall+.

The McCall OSSD extensions are based on two sources. First, the NLP literature for requirements parsing suggests key words and parsing strategies—in particular,
Cleland-Huang et al.’s [8] study on NFRs. Second, analysis of natural language associated with OSSD projects suggests further key words and parsing strategies. In particular, we iteratively extended the RCNL parser and tested it on sample data until we reached a consistent level of correct classification. In addition, we made use of the SensAgent online dictionary (www.sensagent.com) for gathering all synonyms that are properly describing the same meaning as the original key word. Here, we report on only the McCall+ classifications because of their significantly better classification efficiency over the McCall classifications.

Classifier Engineering

The RCNL classifier is implemented in GATE (General Architecture for Text Engineering) [11]. Next, we describe the engineering involved in realizing the RCNL framework in GATE. In particular, we describe rules for tagging text according to the ontology, additional text processing, and the overall text processing activity.

Our parser implements the RCNL ontology to recognize and classify natural language requirements. For each level, rules developed with the Java Annotation Pattern Engine (JAPE) specify how GATE tags text with concepts of that level. The rules are processed in a pipeline, from Level 0 to Level 5. The final output tags qualified (L2) requirements (L4) according to their classification (L5). Any text may have multiple tags from multiple levels.

GATE supports Levels 0 and 1 directly, identifying tokens and some parts of speech. The RCNL classifier rules augment and extend the native GATE tags to aid processing for OSSD projects.

Parsing Pipeline

The process begins with a natural language resource, which is processed through a series of special-purpose programs, or plug-ins. Preprocessing includes five stages:

1. Language resources are converted into GATE format.
2. An English Tokenizer plug-in identifies tokens within the text provided. These tokens are basic elements of text, such as words, punctuation, or spaces.
3. A Sentence Splitter plug-in identifies and annotates pieces of text corresponding to sentence and paragraph structures.
4. A POS Tagger plug-in identifies POS in the text. POS are elements such as adjectives, adverbs, nouns, verbs, conjunctions, and so forth.
5. A Morphological Analyzer plug-in identifies each token’s lemma and affix. We use this plug-in in our analysis because it provides stemming capabilities to our NLP analysis.

Named entity transducers follow preprocessing. This process runs the RCNL JAPE rules on the preprocessed data. Discovery and classification in the SAO-based analysis consists of 199 JAPE rules, while the delimiter-based analysis consists of 76 JAPE rules.
We developed separate sets of JAPE rules for discovery, classification, and evaluation. In the SAO-based analysis, discovery rules include Levels 0 through 4 from the RCNL requirements parsing ontology (Table 1). In the delimiter-based analysis, discovery rules include Levels 2 through 4. Classification rules are the same in both approaches. Finally, a third set of rules postprocesses the analysis results preparing measures of interest for the evaluation process.

Rule-Based Tagging

GATE provides JAPE, a rule-based text-engineering engine that supports text parsing. GATE also provides an annotation indexing and search engine with an advanced graphical user interface called ANNIC (Annotations in Context). Our analyses use ANNIC for development of rules and inspection of results, and JAPE for rule design and implementation.

JAPE rules specify a left-hand side (LHS) describing the pattern to be matched and a right-hand side (RHS) defining the annotation and the features to be created for all the discovered instances of the pattern.

The current implementation of the RCNL classifier consists of a total of 252 distinct JAPE rules. To illustrate how the RCNL ontology is recognized through JAPE rules, we present simple rules from Levels 3 and 5.

A Level 3 Rule

To illustrate our rule usage, here is a rule from L3-G:

```
Rule: PotentialSubjectFinder
{
    {Token.category==PP} | {Token.category==PRP} | {Token.category=="PRPR$"} | {Token.category=="PRP$"} |
    {L1.category == "Noun"} | {L0.category == "Filename"} |
    {L0.category == "email"} | {L0.category == "url"} |
    // . . .
    ({{L1.category == "Determiner"} {L1.category == "Noun"}}
    ) [1,5]
} :SubjectFound
-> :SubjectFound.L3 = {category = "Subject"}
```

The LHS part of the rule describes a pattern searching for nouns (as defined in L1), or pronouns (as defined in predefined rules in GATE), or filenames, URLs, e-mail, (etc., as defined in L0), or a determiner followed by a noun (up to five instances of
this entire pattern). When one of these is found, the text matching the pattern is annotated as an L3 subject.

A Level 5 Rule

Here is a (slightly simplified) 5-G classification rule:

```
Rule: L5_Comunicativeness
( {L4.valid == "Yes,"L4.Requirement contains KW_F5C12} )
:L5_ComunicativenessFired
--> :L5_ComunicativenessFired.Comunicativeness = {category = "F5C12"}
```

The LHS matches text annotated as L4 (requirement) that contains key words associated with Factor 5 and Criteria 12 of McCall’s model. These key words are previously defined in another JAPE rule that annotates each instance of them as KW_F5C12. The matched text is annotated as Communicativeness, which is the name for Factor 5, Criteria 12.

Auxiliary Text Processing

Three auxiliary kinds of text processing are noteworthy. First, list processing presents an interesting problem. Project texts include technical yet informal communication containing numerous examples of specifications expressed with lists. A list typically has an introductory phrase followed by one or more list items:

```
<Introductory phrase> [<list item>]+
```

Sometimes the introduction phrase and each list item are complete requirements. Often, however, the introductory phrase can be classified as a requirement, while the list items are examples or phrases that extend the meaning of the introductory phrase. To address such issues, the 5-G tag associated with the introductory phrase is propagated to the list items. As such, a list item can have two tags: a tag from parsing the list item and a tag propagated from the introductory phrase. List classification processing occurs at Level 5-G.

A second auxiliary text processing applies to check requirement containership. It is possible, but rare, that a (5-G) requirement is fully contained inside another requirement. When found, a final finishing rule reannotates requirements to indicate that only the larger requirement should be considered for classification. This avoids double annotating and double classifying same piece of text.

The last auxiliary text processing generates metrics for evaluating the rules. These include tokens covered, sentences covered, requirements tagged, classifications created, and requirements classified.

All rules are controlled by a configuration. Our experiments, described next, enable or disable various rules to determine their contribution to the classifier’s performance.
Classifier Experiments

Having created a requirements classifier for the unstructured and typo-rich natural language text, we applied two validation methods to assess it. In addition, we implemented two strategies for OSSD requirements classification and assessed their characteristics.

- Which strategy classifies the most text?
- Which strategy is most consistent with the way a human analyst would mark requirements within the text?
- Is there any value in running both strategies, that is, the hybrid approach?

To answer these questions, we applied two validation methods. First, we generated metrics from the classification of sampled OSSD project data. Second, we compared the automated classification to that of an expert. These two experiments are presented next.

SourceForge Data Set

Like many researchers in OSSD, we selected SourceForge projects for our data set [35]. SourceForge provides access to over 260,000 OSSD projects and over 2.7 million registered users’ activities, as of November 2011. We took advantage of the enhanced online access offered to the SourceForge data set by the SourceForge Research Data archive (SRDA) [20, 36]. In particular, we processed the May 2011 data from SourceForge.

We narrowed our data set to substantial projects that actively used requirements. We define this as (1) having more than four developers, (2) more than 5,000 downloads, and (3) more than 600 feature requests. After review of the project data, Biet-O-Matic was removed because it is entirely German natural language text, and our classifier’s rules are designed for processing exclusively English natural language data. The result is a data set comprising 30 projects with averages of 4,962 sentences consisting of 110,517 tokens, 1,035 feature requests, 763,984 downloads, and 18 contributors.1 While the project with the longest history and the most active community is SourceForge.net (over 5,200 feature requests posts), the total number of downloads indicates phpMyAdmin as the most popular (almost 6 million downloads). In terms of project contributors, Tiki Wiki CMS Groupware is the project with the largest set of contributing members (over 130). The Matplotlib project had many spam posts, perhaps the result of a virus on a contributor’s computer. In order to avoid a significant skewing effect on the results of the analysis, we manually removed all spam postings and analyzed the remaining data of the project.

Experiment Configurations

Our analysis uses experimental configurations to process and evaluate each project from the data set:
1. Requirements recognition based on (a) the SAO-based parser and (b) the delimiter-based parser
2. Requirements classification based on our extensions to McCall’s quality model, called McCall+ (see the section Requirements Parsing Ontology)

Our performance analysis of the RCNL classifier considers the time to recognize and classify requirements using the two strategies:

- SAO-based averages 463.5 tokens/second, which amounts to about 3.97 minutes for an average-size project in our data set. These rules are many, but very simple. Consequently, they process quickly due to limited memory and computational demands.
- Delimiter-based averages 211.0 tokens/second, which amounts to about 8.73 minutes for an average-size project in our data set. These rules are few and complex. Consequently, the processing time is longer due to the greater memory and computational demands.

Note that neither grammar is a simple context-free grammar as found in most programming languages. The parser applies knowledge of English-language terms and grammar. (See the fourth section.) Naturally, classification takes the most time. The results were obtained on a 3.2 GHz computer running Windows 7. Database retrieval of the features, storage of the results, and evaluation processing are not considered in these numbers. The performance analysis shows that both strategies offer reasonable computational times to project leaders interested in applying our automated requirements discovery and classification tool.

Data Analysis

The results show that the SAO-based strategy discovers an average of 7,304 microrequirements compared to the 1,607 macrorequirements discovered using the delimiter-based strategy. The SAO-based strategy identifies more requirements per project because it discovers all SAO-triples within text. In contrast, the delimiter-based strategy identifies, on average, a little more than one requirement delimiter for every other feature tracker post. On average, the delimiter-based strategy identifies 1.62 requirements per feature tracker post. The delimiter-based strategy includes all the text of each post; thus, text coverage is 100 percent. In contrast, the SAO-based strategy excludes text that does not conform to the SAO parser; thus, its sentence coverage averages 84.3 percent.

Consider the Compiere ERP (enterprise research planning) project as an example. The analyzed text has 64,421 tokens. The tokens comprise 2,937 sentences, 86.1 percent of which are recognized by RCNL as including one or more requirements statements (microrequirements), according to the SAO parser. The remaining text is unrecognized by the SAO parser, often because it is code segments, social tags, and so forth. In contrast, the delimiter parser considers all the text within a post to belong to one or more requirements.
Once a segment of text is recognized as a requirement, it is passed to the classifier, which attempts to classify it according to a specified ontology. The same classifier applies to both SAO-based (micro) requirements and delimiter-based (macro) requirements. The difference is that the SAO-based requirements are shorter segments with length varying between SAO-length and sentence-length, while the delimiter-based requirements are mostly post-length requirements, usually including few sentences. McCall+ provides classifications per requirement at an average rate of 2.4 and 4.2 for SAO-based and delimited-based requirements, respectively. This is expected because the longer-length delimiter-based requirements have more words, and thus a higher likelihood of matching more than one classification. Conversely, the SAO-based requirements have fewer words and thus, on average, they match fewer classifications.

Our analysis also includes a number of additional metrics associated with the delimiter-based requirements (macrorequirements) and with the SAO-based requirements (microrequirements). We can draw some interesting inferences from the metrics:

1. A feature request posting for the selected projects averages 1.62 macrorequirements. (There is an average of more than 0.5 delimiters that split a post into more than one requirement.)
2. On average, there are 4.45 times more microrequirements than macrorequirements. (An aggregate macrorequirement contains 4.45 subrequirements.)
3. On average, there are 4.2 classifications per macrorequirement and 1.4 per microrequirement. This means that feature tracker posts refer to more than one of McCall’s factors, but each subrequirement addresses (slightly more than) one factor.
4. On average, there are three more classifications per requirement using the delimiter-based (macro) analysis than the SAO-based (micro) analysis. (The greater word length in an aggregate requirement increases the likelihood of matching classification key words and phrases.)
5. The amount of text recognized as a requirement is 100 percent for the delimiter-based (macro) analysis and averages 66 percent for the SAO-based (micro) analysis (see [58]). (Delimiter-based requirements cover, by definition, all the text of a feature tracker post.)

A hybrid parsing strategy may be best. It can characterize an aggregate requirement and its supporting subrequirements.

Expert Analysis

To evaluate the two parsing strategies, we employ established techniques for performance evaluation of NLP tools in the fields of information extraction (IE) and information retrieval (IR) [18, 38, 48]. The evaluation measures we use are precision, recall, and $F$-measure. Precision is expressed as a percentage value and represents the ratio of the correctly identified elements to the total number of identified elements; precision measures how many of the items identified by the automated classifier are
correct items. A tool capable of identifying mostly correct items has high precision. Recall is expressed as a percentage value and represents the ratio of correctly identified elements to the total number of correct elements; recall measures how many of the items in the data set are identified by the automated classifier. A tool capable of identifying most of the correct items in the available data set has high recall.

Given the definitions for precision and recall, it is clear that achieving either 100 percent precision or 100 percent recall is easy. A tool that identifies no items will have perfect precision. Similarly, a tool can have 100 percent recall when it identifies everything from a data set. The common way to evaluate a tool’s efficiency is by using precision, recall, and a weighted measure of them. $F$-measure (sometimes called $F$-score) provides this weighted average [48]:

$$F\text{-measure} = \frac{\left(\beta^2 + 1\right) \cdot P \cdot R}{\left(\beta^2 \cdot P\right) + R},$$

where $\beta$ indicates the weighting of precision and recall, $P$ represents precision, and $R$ represents recall. In our evaluation, precision and recall receive equal weighting; thus $\beta = 1$.

Precision, recall, and $F$-measure can only be calculated if a key is available for providing an example of the ideal discovery and classification result. This key is called a gold (or golden) standard in the IR literature and represents the benchmark against which a tool’s output is compared. A gold standard is normally created by experts in the field and is the result of successive passes through a sample data in order to improve its quality. Ideally, a gold standard provides a perfect, error-free sample result. The gold standard we use for evaluation was created by a requirements expert, one of the authors, and provides the output of an ideal, error-free requirements discovery and classification process. Producing a gold standard requires an expert and time, and thus is very costly in practice. It is impractical to have an expert identify and classify thousands of requirements. However, a gold standard for sampled data allows for the accurate evaluation of an automated tool.

We calculate precision, recall, and $F$-measure as a result of comparing a gold standard to the output of the RCNL tool. For the SAO-based strategy, we randomly select 25 data segments from the 515 data segments in the data set. Then, we randomly extract a short text sample (between 1 and 4 postings long) from each selected data segment. These 25 randomly selected sample texts are manually tagged with annotations corresponding to microrequirements and the 23 classification types from McCall+. The expert tagged and classified at an average rate of 515.22 tokens per hour. At this rate, the RCNL is about 3,239 times faster than the human expert. The expert tagging process for the SAO-based strategy is designed to annotate sentence-level requirements defined as an SAO pattern and the associated extending patterns. The delimiter-based gold standard is developed in a similar manner and is created at an average of 1,695.93 tokens per hour. At this rate, the RCNL is about 448 times faster than the human expert. These processing results are explained by two factors. First, as indicated in the Experiment Configurations section, the RCNL processing time for an SAO-based
analysis is shorter than the one for a delimiter-based analysis due to the complexity and memory requirements of the processing rules. Second, the human effort required for the identification of requirements at the forum post level is naturally less than the one required for the identification of requirements at the sentence level, due to the level of detail associated with these two tasks. Consequently, the automated requirements discovery and classification with our RCNL is significantly faster than a human expert, especially in the case of an SAO approach.

The results of the gold standard analysis are summarized in Table 2. The last three columns show recall, precision, and \( F \)-measure. The middle four columns show how the match occurs between the standard and the automation. The Process column indicates which process’s resulting annotations are being considered in the computing of the evaluation measures: (1) requirements discovery (identification of microrequirements or macrorequirements in the data set), (2) requirements classification (the classification of the identified requirements), or (3) requirements discovery and classification (the identification and classification of microrequirements or of macrorequirements). The Annotations column clarifies this context by indicating specifically which annotations are used to compute the evaluation measures. The results we obtain depict an efficient discovery process (\( F \)-measure of 83 percent for grammar-based requirements discovery). They also indicate a less efficient classification process (\( F \)-measure of 29 percent for classification). A number of factors explain these evaluation results.

First, the evaluation scores for the classification process are calculated according to McCall’s model. McCall’s quality model was developed in 1977 when information systems were different from today’s information systems. This model might not reflect current open source users’ perceptions of quality accurately. Second, when there is a mismatch between a gold standard microrequirement and the output of the automated tool, cascading mismatches occur—a missed gold standard requirement also misses the associated gold standard classifications (averaging 2.4). Third, the semantic content is significantly determined by the overall context provided in the entire posting. The SAO strategy performs the analysis at a sentence level. Thus, much of this context is not available, and therefore some classifications are absent.

When combining discovery and classification into one comprehensive evaluation effort, we have 24 annotations grouped together as one composite annotation. (See Discovery and Classification in Table 2.) The instances within text of these 24 annotations represent the sum of the instances corresponding to microrequirement and to the 23 classification criteria. For example, there are 104 matches for discovery and classification (second row), which corresponds to the sum of 69 matches from discovery (first row) and the 35 matches from classification (third row). Because discovery and classification (second row) lists the combined values of discovery (first row) and classification (third row), it is also expected that the effectiveness (expressed through recall, precision, and \( F \)-measure) of it would be between those of the two processes it consists of (discovery and classification). More precisely, it is closer to the effectiveness of classification because in the combined evaluation the classification process contributes 23 annotations, whereas the discovery process contributes only one annotation.
Table 2. Expert Comparison Measures

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Process</th>
<th>Evaluated</th>
<th>Matching</th>
<th>Only in gold standard</th>
<th>Only in output</th>
<th>Overlapping</th>
<th>Recall</th>
<th>Precision</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAO</td>
<td>Discovery</td>
<td>Microrequirement</td>
<td>69</td>
<td>22</td>
<td>74</td>
<td>168</td>
<td>0.92</td>
<td>0.76</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td>Discovery and Classification</td>
<td>Microrequirement and 23 classification criteria</td>
<td>104</td>
<td>342</td>
<td>455</td>
<td>274</td>
<td>0.52</td>
<td>0.45</td>
<td>0.49</td>
</tr>
<tr>
<td></td>
<td>Classification</td>
<td>23 classification criteria</td>
<td>35</td>
<td>320</td>
<td>381</td>
<td>106</td>
<td>0.31</td>
<td>0.27</td>
<td>0.29</td>
</tr>
<tr>
<td>Delimiter</td>
<td>Discovery</td>
<td>Macrorequirement</td>
<td>189</td>
<td>14</td>
<td>72</td>
<td>136</td>
<td>0.96</td>
<td>0.82</td>
<td>0.88</td>
</tr>
</tbody>
</table>
Overall, the results are encouraging. Although there are no other tools to provide the same type of analyses, we note the similarity between our results and those from a 2006 study by Cleland-Huang et al. [8]. In their study, Cleland-Huang et al. explore approaches to NFRs’ discovery and classification. They start with exploring whether a predefined fixed set of key words can be efficient in classifying NFRs. Their findings include recall between 61 percent and 80 percent and precision between 40 percent and 57 percent but also highlight a difficulty of this approach—the problem of finding accepted sources of key words for specific types of NFRs. Consequently, they develop an NFR-Classifier that includes a mining phase for key word extraction from a training set. The extracted indicator terms are ranked and determine the two alternate extraction methods considered: (1) top K terms selected and (2) all terms selected for each NFR type. When the top 15 terms are used, the classification of different NFR types exhibits different efficiency levels. Recall ranges between 51 percent and 98 percent (with an overall recall of 77 percent), while precision ranges between 19 percent and 37 percent (with an overall precision of 25 percent). Our approach to classification is also key word based, but we use a different, pattern-based approach to requirements discovery. While we acknowledge the distinct characteristics of the two studies, we also note the similarity of the evaluation values. Our recall ranges between 31 percent and 96 percent, and our precision ranges between 27 percent and 82 percent. We conclude that the two studies implement distinct approaches and complement each other while proposing similarly efficient tools.

Discussion

This study contributes to research and practice of OSSD. A systematic method for discovery and classification of requirements in OSSD projects is currently not available. When available, such a method would enable important improvements, such as (1) better understanding of requirements, their kinds, and life cycles, and (2) better understanding of project scope, goals, and overall project direction. Such understanding in turn leads to better understanding and improvement of both OSSD projects and more traditional software development.

This research study provides three specific contributions:

1. A grammar-based design of software automation for the discovery and classification of natural language requirements
2. Two alternative parsing schemes implemented within the design
3. Requirements discovery, classification, and analysis of 30 OSS projects

Together, 1 and 2 above affirm H1a: A multilayered grammar, varying in domain specificity, can be constructed for the automated requirements discovery and classification of requirements contained within natural requirements documents of OSSD projects. In total, these contributions provide a path for subsequent empirical studies of OSS requirements and enable subsequent software tools for facilitating automation of requirements traceability analysis in support of IS development process studies. The RCNL classifier provides an alternative to existing methods that require substantial
input from the researcher (second section). The RCNL classifier runs autonomously. However, users may choose to customize rules from the topmost layers of the six-layer ontology to work most effectively with new data sets. Although we did develop and test it on a large SourceForge data set, it may be that other OSSD data or traditional software artifacts require changes to the lower levels of the parser (Levels L0 through L2 in Table 1). To adapt the RCNL classifier to another quality model (other than McCall), the Level 5 rules must be modified.

The RCNL classifier can be applied to traditional requirements documents. Most such documents have clearly delineated requirements, with few classifications. The RCNL classifier can be used to extend existing requirements classifications or provide new classifications where they do not exist.

Our future work has two main directions. First, we will continually refine the parsing rules to improve the quality of the recognition and classification. This will be achieved largely through detailed analysis of partially correct and missing tags on our data sets. Another improvement direction we consider is represented by capturing part of the context surrounding a grammar-based requirement. This will be achieved through the implementation of reference resolution techniques [34, 37, 42]. Second, we will extend the data to include structured text. Feature requests, bug reports, and other tracked work items have a variety of structured attributes, including author, data, version, references (links), and so forth. We believe such structured data can be used to increase the recognition and classification quality. With access to the structured data, we plan to extend the work to analyze traceability relationships, such as contributions, evolution, and the interrelation between requirements and code.

Acknowledgments: The authors thank the National Science Foundation for its support via NSF grant no. 0613698.

Notes
1. The 30 projects are SourceForge.net, Gallery, KeePass Password Safe, FileZilla, phpMyAdmin, PhpGedView, WinMerge, POPFile—Automatic Email Classification, Ariane RPG, MegaMek, Tiki Wiki CMS Groupware, AWStats, Scintilla, Matplotlib (spammed), floAt’s Mobile Agent, Compiere ERP + CRM Business Solution, TortoiseCVS, JabRef, Gutenprint—Top Quality Printer Drivers, Fire, SW Test Automation Framework, EGroupware Enterprise Collaboration, more.groupware, MediaWiki, Windows Installer XML (WiX) toolset, ScummVM, Slash, Password Safe, Tcl, and OSCARMcMaster.
2. The results of the automated analysis provided many metrics, but space limitations preclude their presentation. Among them are the counts of requirements for each classification and the total number of classifications created per project and per project interval (183 days long). The projects averaged 6,788 total classifications and 395 new classifications each 183 days. Such analysis is enabled by the automated classification. Interested readers may contact the authors for a more complete report of the experiment.

References


