Using Syntactic and Semantic Analyses to Improve the Quality of Requirements Documentation

Editor(s): Giancarlo Guizzardi, Federal University of Espírito Santo, Brazil
Solicited review(s): Ivan Jureta, University of Namur, Belgium; Renata Guizzardi, Federal University of Espírito Santo, Brazil

Kunal Verma*, Alex Kass, Reymonrod Vasquez
Accenture Technology Labs, 50 W. San Fernando Street, Suite 1200, San Jose, CA, USA

Abstract. We discuss our experiences with deploying a tool called the Requirements Analysis Tool (RAT), which automatically reviews requirements documents for clarity and content based issues using a variety of syntactic and semantic techniques. The tool has been deployed at over 500 large software projects. We provide an overview of our syntactic approach, which is based on enforcing restrictions on both sentence structure and vocabulary in a way that is carefully chosen to align with best practices. We discuss how RAT analyzes natural language text to find defects such as terminological inconsistencies and missing contextual information. Structured content from requirements is then represented as a semantic graph and RAT performs semantic analysis to help users perform interaction analysis. We present a number of case studies based on real world deployments of RAT which demonstrate number of improvements in the projects' requirements ranging from clearer sentence structure to more complete requirements.

Keywords: Requirements Analysis Tool, Semantic Analysis, Syntactic Analysis, Requirements Ontology

1. Introduction

This paper describes the successful results we have deploying a tool we have built, called Requirements Analysis Tool (RAT), which helps promote a set of best practices in requirements documentation. It does this through an automated review that flags portions of the requirements document that fail to follow these best practices, and produces reports that make it easier for business analysts to spot other violations. This paper provides an overview of the tool and also presents 7 case studies and outlines lessons learnt from the deployment of RAT at over 500 enterprise projects.

The past few years has seen a trend in the research world toward promoting alternative techniques for specifying software requirements, such as: prototyping and simulation, user stories [6] and use case analysis [9]. There have also been approaches based on formal (typically logic-based) specifications of requirements that reason about the specifications. Key examples of such approaches include goal based analysis [2][13][22] and model checking [4]. However, our experience working with practicing developers of large-scale enterprise systems convinced us that - at least for those kinds of systems – these techniques *augment* traditional requirements documents, rather than replace them. For various reasons, such as: the skill-set of business analysts, preferences of stakeholders who must sign-off the requirements, concerns of scalability with most formal modeling approaches, the bulk of the specification of most large software systems is still contained in a document containing long lists of requirements written in natural language.

Because incomplete, unclear, or ambiguous requirements in such documents frequently drive up the cost of software development projects, there has been a longstanding interest in improving the quality of requirements documentation with the help of automated review to detect various kinds of defects. However, attempts to automate the review of re-

* Corresponding Author. E-mail: k.verma@accenture.com

quirements have been challenged by the fact that requirements are typically written in natural language. Automated systems which require requirements to be expressed in a formal notation have run into adoption problems because analysts typically find these formalisms too difficult to adopt, especially for large, enterprise computing projects. On the other hand, if analysts are allowed to use unrestricted natural language, the language-processing problems become intractable, and the reliability of any automated analysis is greatly reduced.

A practical approach to automating the review of requirements documents therefore requires finding just the right compromise to ensure that requirements can be documented in a way that is convenient for the writer, clear for the reader, and tractable for the automated review system. One of the key contributions of our work is the identification of an approach that places restrictions on the syntax and vocabulary of natural language requirements that - (1) are feasible for analysts to adopt, (2) align with accepted best practices for writing clear requirements, and (3) make it feasible to produce a useful set of automated analyses to detect clarity-based issues. A second contribution is a system that (1) leverages our restricted syntaxes and user defined glossaries to extract structured content from the requirements documents; and (2) uses a semantic framework to detect content-based issues by automating certain aspects of manual tasks such as interaction analysis. Our semantic analysis framework consists of a core, extensible requirements ontology, domain specific ontologies that extend the core requirements ontology and a reasoning engine.

The rest of the paper is organized as follows. Section 2 provides an overview of some common issues in requirements documents that RAT can detect, and an explanation of the best practices that it encourages. Section 3 discusses how we leverage user-defined glossaries and controlled syntaxes to parse requirements documents and detect defects. An overview of our approach for detecting syntactic problems is found in section 4. Section 5 presents our approach for semantic analysis. Section 6 briefly discusses our approach for creating reusable checklists and requirements. Section 7 presents case studies that describe the usage of RAT at large, enterprise projects and section 8 outlines some of the lessons that we have learned. We contrast our approach with related work in Section 9. Implementation details are presented in Section 10. Our conclusions and future work are presented in Section 11.

In one previous paper [10], we described our natural language processing (NLP) approach for automatically detecting and flagging clarity-based issues. In another [23], we presented our broad vision of semantic analysis for detecting content-based issues. Sections 3 and 4 of this paper, and parts of Section 2 briefly review material covered in [10]. The rest of the paper contains unpublished content: Section 5 provides a more detailed explanation of our semantic approach than that discussed in [23], especially with regards to interaction analysis and the structure of the ontology. The case studies and lessons learned are also presented for the first time.

2. Common Issues in Requirements

In this section, we will discuss some common requirements issues. We'll also discuss the best-practices that are widely promoted to avoid each problem type. Finally, we will discuss how RAT helps users deal with those problems. We categorize the issues into two categories – 1) clarity-based and 2) content-based.

2.1. Clarity-based Issues

Clarity issues are, in most cases, independent of the domain of the project and they can lead to confusion and misinterpretation. They are often caused by three kinds of syntactic and lexical problems – 1) Use of problematic phrases, 2) terminological inconsistency and 3) missing contextual information.

2.1.1. Use of Problematic Phrases

There are a number of problematic phrases that lead to vagueness or ambiguity in requirements such as *quick*, *easy to use* and *flexible*. Since, this issue has been dealt extensively by previous work (for e.g., [12], [26]), we will not cover it in this paper.

2.1.2. Terminological Inconsistency

Terminological inconsistency refers to using different terms to refer to the same entity/verb, leading to potential confusion. For example, consider the following two requirements: *A1: The Order Processing System shall generate profit reports. A2: The Order Entry System shall allow users to view all orders placed in a day.* If the requirements analyst uses two different terms to refer to the same system, this can lead to confusion. Generally accepted best practice is to create a glossary of standard terms, and to only use terms that are defined in that glossary. However, because enforcement of the glossary discipline

has generally been manual, it is hard to leverage glossaries systematically, or even to motivate teams to create them. RAT addresses this issue by checking whether the entities and actions that occur in the requirements document have been defined in the glossary and generates error messages, if that is not the case.

If "Order Processing System" is in the Entity Glossary but "Order Entry System" is not, RAT will generate the following warning message: *"This requirement contains 'Order Entry System ' where an agent is expected, but 'Order Entry System ' is not in the entity glossary."* The message should help the user consider whether a new agent ("Order Entry System") should be added to the entity glossary or the second requirement (A2) should be rewritten by replacing "Order Entry System" with "Order Processing System" as the agent.

2.1.3. Missing contextual information

Users often leave out key contextual information such as which agent/actor should perform an action. It may be obvious to them but it can lead to confusion and potentially expensive rework. This kind of problem is often associated with certain sentence structures, such as incomplete sentences or passive voice sentences. Therefore, best practices for requirements documentation [25] specify that each requirement should be stand-alone and explicitly state who is responsible for performing which function. But without automated review against these practices, they are often violated. There are two common forms this can take:

Sentence fragments: Consider: *"Report1: must generate profit reports"*. RAT generates a warning message stating: *"This requirement lacks an agent before 'must'. It can be confusing to leave the agent implicit*". The attempt is to guide the user towards a better written requirement that includes an agent. For example: *"Report1: The SAP system must generate profit reports."*

Requirements that are written in passive voice: *Discount1: If the person's age is greater than 60, the age factor discount shall be applied."* RAT generates the following warning message: *"This requirement contains 'Age factor discount' where an agent is expected but 'Age factor discount' is in the entity glossary but is not designated as an agent."* In this case, RAT is encouraging the user to write the following requirement: *"Discount1: If the person's age is greater than 60, the claim processing system shall apply the age factor discount."*

The comments that are automatically generated by RAT for the clarity-based issues discussed in this section are shown in Figure 1.

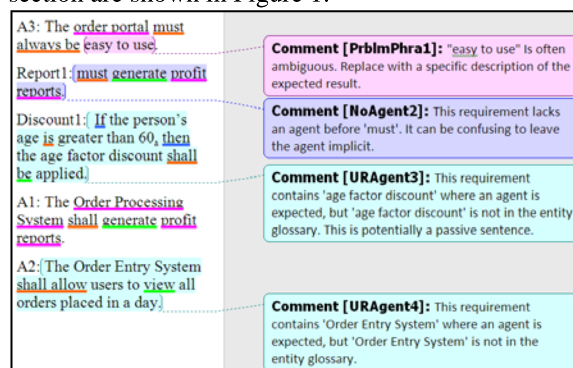

Figure 1: RAT generated comments for clarity-based issues

2.2. Content-based Issues

The issues discussed above are all about requirements that are not written clearly. In addition to these clarity problems, there can be content issues in requirements documents, such as conflicting requirements or goals [22], system behavioral inconsistencies [4] and missing interaction or non-functional requirements. Since detection of these issues generally involves a lot of domain expertise, we refer to them as content-based issues. In this paper, we will discuss one issue that typically occurs in large enterprise projects – missing interaction requirements.

2.2.1. Missing interaction requirements

Consider a large software project, where a newly developed enterprise portal has to be integrated with a number of existing systems. Often times, an interaction with another system, say the "customer management system" may be overlooked, either because the stakeholders forgot about it or the business analyst did not note it down. Since, the requirements documents can often span hundreds of pages, the missing interaction may not be discovered until much later, where it is much more expensive to fix. RAT helps deal with this issue by generating a textual report and a visual representation of all the system-system and system-user interactions in a requirements document. This allows users to quickly check all the interactions and validate them with the stakeholders.

3. Overview of NLP Approach

Existing work (for e.g., [12][26]) handled problematic phrases well, so we simply adopted those techniques in RAT. The research challenge was to be able to identify other kinds of defects such as terminological inconsistencies and missing contextual information. For the other problems, we needed an approach that would allow us to automatically detect the nouns, the verbs, and their relationships in textual requirements. While our initial approach was to investigate the use of general-purpose NL parsers, we found two very interesting aspects of requirement documentation best practices recommended in the requirements engineering literature (e.g., [25]), that we determined could be exploited to support a simpler, yet more reliable approach:

1. Requirements should systematically be written in a stylized subset of English, with specific sentence structures corresponding to various categories of requirements. For example, simple, active voice declaratives are the most common recommendation for functional or solution requirements.
2. Requirements should be written in a carefully controlled vocabulary. Analysts should maintain glossaries defining the actions and entities that will be used, and should use only those throughout a requirements set.

RAT *enforces* the above recommendations by flagging violations, and then *leverages* the restrictions to greatly reduce the complexity of the parsing process. For example, to aid automated analysis of requirements, users are required to create glossaries that define all the entities and actions to be used in the requirements document. Glossaries serve as a reference for valid entities and actions. The fact that RAT finds instances where undefined terms are used makes RAT very useful to users, as it helps them adhere to the glossary discipline. From a RAT-centric point of view, the glossary entries help with tokenization and make the analysis of natural language requirements tractable.

3.1. User Defined Glossaries

RAT uses two kinds of glossaries to process requirements sentences. 1) *The entity glossary* (shown in Table 1) is used to capture all entities in a solution such as systems, sub-systems, interfaces, users, processes, records, and data fields. In the entity glossary, users are required to define and classify each entity. Entities must be classified as agents or passive entities. An agent entity is any entity that can perform an action. Users can further classify the agents as systems, users or any user-defined sub-class of an agent. 2) The *action glossary* lists the valid actions that can be specified for users and systems in the requirements.

Table 1: Snapshot of Entity Glossary

Agent Descriptor	Explanation	Type	Parent
Order Proc. System	Used to place orders	System	-
Billing module	Handles billing for order processing systems.	System	Order Proc. System
Drop Ship Process	Process that controls the supply chain.	Agent	-
Administrator	Admin for the system	Person	-
Order details	Details of the order	Passive Entity	-

3.2. Overview of restricted syntaxes

RAT requires users to follow a set of controlled requirements syntaxes, drawn from previously defined best-practices (e.g., [25]), each of which corresponds to a conceptual requirement category. This discipline includes such practices as writing in active voice and clearly indicating which entity is providing which function. RAT supports 7 requirements categories; three of which are described in Table 2. (Note that agent, entity and action tokens refers to glossary entries and 'rest' token allows any text.) The syntaxes for each category were carefully designed such that each of them is uniquely identified using identifier phrases (shown underlined in Table 2).

Table 2: Requirement categories and examples.

<table>
<tr><th>Requirement Category</th><th>Syntax and Example(s)</th></tr>
<tr><td>Solution Requirement: Describes a function the solution must perform</td><td><agentPhrase> <"shall" | "must"> <action><rest>
A-1: The Billing System <u>shall</u> produce invoices with rolling rates.</td></tr>
<tr><td>Enablement Requirement: Expresses a capability that the solution must provide to the user.</td><td><agent> <"shall" | "must"> <"allow"> <agent> <"to"> <action><rest>
A-2: The billing system <u>shall allow</u> the user to determine how much each client owes.</td></tr>
<tr><td>Attribute Constraint: Describes constraints on attributes and values.</td><td><entity> <"shall" | "must"> <"always be"> <constraint>
A-3: Customer standing <u>must always be</u> one of the following: 1) Red 2) Green 3) Blue.</td></tr>
</table>

4. Syntactical Analysis Approach

In this section, we summarize our syntactical analysis approach, which was previously described in detail in [10]. RAT leverages the controlled syntaxes and user-defined glossaries to find syntactic problems in requirements. Our first approach was to look at compiler-compiler tools such as YACC [27], ANTLR [1], which provide a rich framework for analyzing a set of controlled syntaxes. However, these tools are aimed at programming languages, where the syntaxes are less flexible than natural language. We have implemented a two-phased, syntactical analysis approach, which has been heavily influenced by these tools, and allows us to have the flexibility required to analyze a controlled natural language, and leverage user-defined glossaries. The first phase consists of tokenization, which converts a requirement statement into a set of tokens with the help of the user-defined glossaries. The second phase uses state machines for analysis of the requirement statement's syntax. There is a different machine for each syntax type and indicator phrases for the syntax (underlined in Table 2) are used to decide which state machine to use.

Table 3 Phrases of requirements statement

Phrase	Token Type	Reason
SA1:	Label	Valid syntax for label
The SAP System	Agent phrase	Present in entity glossary and classified as an agent
Shall	Modal phrase	Keyword that identifies solution requirement.
Send	Action phrase	Present in action glossary
vendor data	Entity phrase	Present in entity glossary and classified as a non-agent entity
To	Unknown	Not present anywhere
The order processing system	Agent phrase	Present in entity glossary and classified as an agent

We will provide a brief overview with the help of an example of a solution requirement. Consider the following requirement: *SA1: The SAP System shall send vendor data to the order processing system.* In the first phase, the requirement statement is tokenized. Table 3 shows the phrases of this requirement statement, the token types associated to them and a reason for choosing the particular token type. The algorithm uses information from the glossaries and a fixed set of modal words for the tokenization.

In the second phase, a requirement is classified into one of the requirements categories based on the modal phrase and then analyzed using the appropriate state machine for the requirement category. Our grammar is designed such that each syntax type can be identified by its modal phrase. For the example requirement, the keyword "shall" calls for validation using the solution requirement state machine (Figure 2), which represents the following syntax: *<Agent Phrase> <"shall" | "must"> <Action Phrase>*

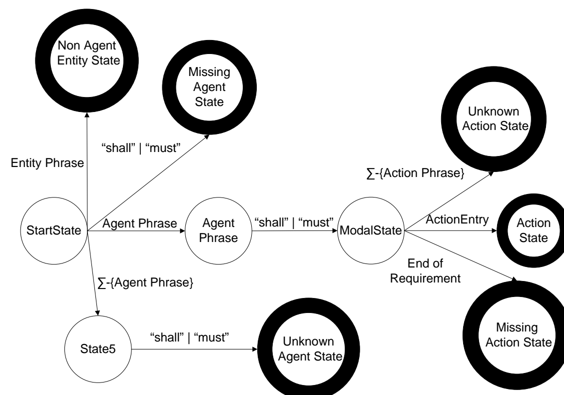

Figure 2: Solution Requirement Syntax State Machine

A requirement is treated as syntactically correct when the state machine successfully transitions from start state to a valid final state, using the token stream for that requirement. However if the state machine enters an error state, then a warning message is generated. For every error state, there is a pre-defined warning message that is displayed to the user. The statement of the warning message also explains why the requirement deviates from best practices. Using the state machine transitions in Figure 1, the token stream (from Table 3) for the requirement will end up in "Action State" and it will be treated as a valid requirement.

Let us also see how this approach generates the error messages that were discussed in Section for terminological inconsistency and missing contextual information. Consider an example of missing contextual information from section 2 - *Report1: must generate profit reports.* For this requirement, the state machine will halt in Missing Agent State (as it lacks an agent phrase in the beginning) and generate the following message "*This requirement lacks an agent before 'must'. It can be confusing to leave the agent implicit.*"

Consider the example of terminological inconsistency from Section 2: *A2: The Order Entry System shall allow users to view all orders placed in a day.* For this requirement, the state machine will halt in the Unknown Agent State (as "Order Entry System" is not in the entity glossary) and generate the following message *"This requirement contains 'Order Entry System' where an agent is expected, but 'Order Entry System ' is not in the entity glossary."*

In addition to analyzing requirements, the syntactical analysis generates structured content which contains information about each requirement such as the requirement category, the primary entity and secondary entities and the action and model phrases.

Extracting Structured Content

For each requirements sentence, structured content is extracted for semantic processing. The information extracted can be illustrated with the following requirement: *C1: The order processing system shall send user data to the order portal.* The following information is extracted:

- The primary entity and its classification (<*Order processing system*, system>)
- All other entities or secondary entities and their classifications (<*user dat*a, passive entity>, <*order portal*, system>
- All the actions in the requirement (*send*)
- The category of the requirement (solution requirement)

5. Semantic Analysis overview

Experts use a lot of domain knowledge to study requirements documents for various problems, such as dependencies and conflicts between requirements, and finding missing requirements. Our approach involves using ontologies and rules to represent domain knowledge and then to leverage them to analyze the requirements for content-based issues. We create a semantic graph for a requirements document by using extracted structured content generated from syntactic analysis. Each requirement is modeled as an instance in the core requirements ontology that we discuss later in this section. In addition to the requirements themselves, domain specific ontologies, and the glossaries are also leveraged to create this graph. In essence, a textual document is converted into a semantic representation, which can be queried and reasoned upon. While we presented a vision of this broad approach in [23], we will present one specific application of this approach in this section - supporting interaction analysis.

5.1. Core requirements ontology

The core requirements ontology is an extensible ontology that models requirements and related artifacts.

A snapshot of the requirements ontology is shown in Figure 3. For brevity, only one requirement category (Solution requirement) is shown. Classes are denoted with a circle and the instances are denoted using a diamond. Unnamed arcs between 2 classes represent the subclass relationships. Here are some of the important concepts and relationships in the ontology:

Each requirement is an instance of one of the subclasses of the class *Requirement*. All categories of requirements (*Enablement*, *Solution*, etc.) are subclasses of the *Requirement* class. As shown in Figure 3, “r1” is an instance of the *Solution* (requirement) class.

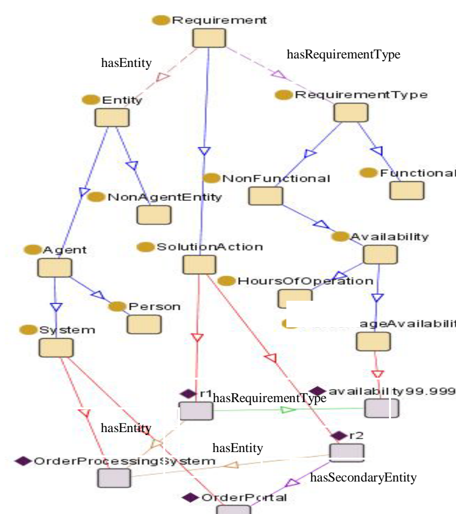

Figure 3: Core requirements ontology that shows many of the core classes and relationships

Each entity defined in the entity glossary is an instance of one of the sub-classes of *Entity* class. Different types of entities (*Agent, System, Person and NonAgentEntity*) are all subclasses of the entity class. As shown in Figure 3, “OrderProcessingSystem” is an instance of the *System* class.

The primary entity of a requirement is related to it by the *hasEntity* property. As shown in Figure 3, the entity “OrderProcessingSystem” is related to requirement “r1” using the *hasEntity* property.

All other entities of a requirement are related to it using the *hasSecondaryEntity* property. As shown in Figure 3, “OrderPortal” is the secondary entity for “r2”.

RequirementType: This class represents the type of the requirement. It has two main sub-classes – *Functional* and *Non-Functional*, which can be sub-classed by users to create their own types. For example three user-created types – *Availability*, *PercentageAvailability* and *HourOfOperation* have been created as sub-classes of *NonFunctional* class in

Figure 3. It is related to a requirement using the *hasRequirementType* property. As shown in Figure 3, "PercentageAvailability" is the requirement type for requirement "r1".

There are also some other properties and classes which have not been discussed for brevity. The current implementation of RAT uses Web Ontology Language (OWL) [16] as the ontology language and (SPARQL Protocol and RDF Query Language) SPARQL [19] as query/rule language. Jena [11] is used as the reasoning engine. OWL already provides the capability to extend ontologies with the help of URI references. Users can leverage this extensibility feature of OWL to add new requirement types by creating sub-classes of the *Functional* and *NonFunctional* classes. Users can also create their own domain-specific sub-classes of the *Entity* class. This can be done by either creating using RAT's user interface to create domain specific checklists or by just creating new domain-specific OWL ontologies that extend the core requirements ontology.

5.2. Supporting interaction analysis

Large enterprise software projects often involve creating a complex eco-system of new systems that interact with each other and existing legacy systems. After eliciting textual requirements, then it is often useful to generate high-level interaction/integration diagrams, and validate the diagrams with the stakeholders. These diagrams are also critical inputs to the design process. The manual, time consuming process of deriving the diagrams can cause key interactions to be missed. RAT leverages the core requirements ontology, extracted requirements content and a set of rules to deduce which requirements describe interactions, and then generates interaction diagrams.

We define interaction requirements as those that have a primary entity that is a system or a person and a secondary entity that is a system or a person. Formally,

Requirement (R) & hasEntity (R, A) & (System(A) or Person (A)) & hasSecondaryEntity(R,B) & (System(B) or Person (B)) => InteractionRequirement(R)

Based on this definition, consider the following requirement ("r2" in Figure 3): *A1: The order processing system shall send user data to the order portal*. If both order processing system and order portal are defined as systems in the entity glossary, this requirement will be treated as an interaction requirement. Conversely, consider another requirement: *R2: The order processing system shall generate profit reports.* Since, the secondary entity in this requirement, "profit report" is not an agent entity; it will not be treated as an interaction requirement. Identified interaction requirements are used to detect which system and users are involved in an interaction. We define an interacting agent as: 1) any system or user that is the primary or secondary entity of an interaction requirement, or 2) any system or user that is the parent of an interacting agent. Formally:

I (System(A) or Person (A)) & InteractionRequirement (R) & (hasEntity (R, A) or hasSecondaryEntity(R, A)) => InteractingAgent(A)

II (System(A) or Person (A)) & child (A, B) & InteractingAgent(B) => InteractingAgent(A)

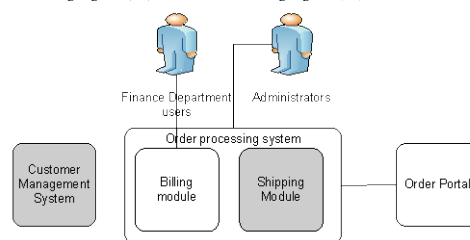

Figure 4: Interaction diagram fragment generated by RAT. Systems without interactions are shaded in grey.

Based on the first definition for interacting requirements whose child is an interacting agent, both order processing system and Web server will be defined as interacting agents. As an example of interaction analysis, consider the following requirements where all the entities are appropriately classified as systems, users and passive entities: *I1: The order processing system shall retrieve order details from the order portal. I2: The billing module shall allow finance department users to view billing information. I3: The order processing system shall allow administrators to generate usage reports.* Based on these rules, RAT will automatically generate interaction diagrams such as the one shown Figure 4. Systems without interactions are shown in grey. We generate three types of diagrams: 1) overall integration diagram: which shows all the interactions between all the systems and users; 2) per system integration diagram: which shows all the interactions with respect to a particular system; and 3) use case integration diagram: which allows all the interactions for a particular use case.

6. Creating Reusable checklists and requirements

From the context of our organization, we often elicit similar requirements in a number of domains across different clients. For example, most enterprises have requirements for number of non-functional areas

such as availability, performance and security. We have worked with practitioners, who deal with non-functional requirements to create checklists and reusable requirements. These are available with the standard download of RAT.

6.1. Checklist of non-functional requirements

This covers a number of areas including performance, availability and delivery channels. Each area has a number of attributes. A snapshot of the non-functional checklist is shown in Table 4. The first column represents an attribute, such as *"Hour of operation"* or *"scheduled maintenance"*. The second column represents a broader area, such as "*Availability*" and "*Performance*". For purposes of semantic reasoning, all areas must be a sub-class of either to "*Functional*" or "*NonFunctional*" requirement types. The third column contains indicator phrases, which are used by RAT to classify the requirements. Finally the forth column contains binary relationships between the attributes.

Table 4: Snapshot of non-functional requirements checklist

Attribute	Area	Indicator phrase	Relationships
Scheduled maintenance	Availability	Scheduled maintenance …	Affects Percentage Availability
Percentage availability	Availability	5 nines, 99.99, …	
Response Time	Performance	Response, ms …	
Encryption	Security	encrypt, SSH, RSA, DSA	Increases Response Time

The checklist can be used either to guide elicitation or by RAT to automatically detect and classify which requirements have been specified for which systems and also detect dependencies and conflicts between them. This is done using a combination of information retrieval and semantic reasoning techniques (some of which were described at a high level in [23]), description of which is beyond the scope of this paper.

6.2. Reusable Security Requirements

We have worked with our security practitioners to convert security specifications from the NIST 800-53 [14] security specification. Our process involved going through different control statements in that specification and converting it into requirements in RAT syntax. The intent for projects to reuse these requirements and glossaries and customize them using RAT. We will be creating a checklist from this specification in the future. We will also be doing the same for similar specifications.

7. Case Studies

In this section, we describe the experiences of seven projects that have utilized RAT to improve the quality of their requirements. Based on the maturity of team's requirements expertise, the teams saw benefits such as:

- On job training for new analysts and improvement of clarity based issues (case study 1).
- Reduced cycles with developer team due to clearer requirement (case study 2).
- Removal of inconsistencies between requirements text and manually general interaction diagrams (case studies 3 and 4).
- Removal of terminological inconsistencies (case study 5)
- Reduction in defects found in peer review phase (case study 6)
- Sharing of glossaries and guidance in functional decomposition (case study 7).

In the remainder of this section, we will present the case studies. Note that the first three case studies are more detailed than the last four, because the project teams provided more data to us.

Case Study 1 - Improving requirements quality at a project with inexperienced business analysts: This case study describes the impact of using RAT in the context of a risk management company. The project was an ERP implementation of a new system that automates some manual processes. One of the issues that the team faced was that some of the business analysts were inexperienced. As a result, senior personnel on the project spent time reviewing the requirements for clarity-based issues. Once the team started using RAT to mark-up requirements, the analysts started to produce deliverables with consistent sentence structure and terminology.

Here are some examples. One analyst initially wrote as requirement "*Feet to hydrant*". Obviously, it was marked up by RAT for not having enough contextual information. The analyst changed it to "*The system must be able to derive protection class based on feet to hydrant*". This was a classic case of a requirement with missing contextual information, where the context was clear to the analyst, but would have led to downstream confusion, especially since the development team was remote. Another effect of

using the tool was that it made the project teams become more specific in referring to details. Consider the following requirement "*Ability to store commission rates by line of business product line*". As the team was defining different terms in the glossary, they also added "*business product line*" from this requirement. However, they realized that they actually had many lines of business, which were also added to the glossary. Based on this insight, this single requirement evolved into the following five requirements: "*A1.1: The system must store commission rates by line of business product line. A1.2: The system must store commission rates for the Homeowners line of business. A1.3: The system must store commission rates for the Collections line of business. A1.4: The system must store commission rates for the Watercraft line of business A1.5: The system must store commission rates for the Excess Liability line of business.*" While, it does make the specification more verbose, having a requirement for each "*business product line*", ensures that the testing team will be explicitly checking for each individual department.

Table 5: Comparison of original and modified requirements

No. of requirements or problems / Document type	Original Requirements Document	Modified Requirements Document
Total No. of req.	164	236
No. of req. with Missing Contextual Information	151	8

The team shared one of their requirements documents with us. Some details of both the original and modified documents are available in Table 5. In the original document there were 164 requirements, which increased to 236 requirements, due to the fact that the team had a number of combined requirements using conjunctions and disjunctions, and had been using high level entities such as "*lines of business*". In the initial document, most of the requirements (151 out of 164) were marked up by RAT as having missing contextual information. This is because many of the requirements had initially been documented as phrases (for e.g., "*feet to hydrant*" or "*ability to …*"). The team rewrote practically all of them as complete sentences. All the newly added requirements were also written as complete sentences, mainly in the solution and enablement requirements syntax.

Case Study 2 – Bringing clarity and completeness to deliverables: This case study describes the impact of using RAT at a project for an auto-insurance company. Four new systems were being added to a complex eco-system which already had sixteen existing systems. Many of the systems had similar functionality and previous iterations of the project had suffered because it was not always clear to the downstream developers which requirement was meant for which system. Before using RAT, many of the requirements were written in passive form. For example, consider the following requirement: "*State Accident Prevention Course Discount applies if principle operator is 55 or older, and has a certificate for an accident prevention course.*" After this requirement was marked up by RAT, it was changed to "*If principle operator is 55 or older, and has a certificate for an accident prevention course, then ICO system shall apply the State Accident Prevention Course Discount*." Once the team started using RAT, the business analysts started writing more consistent sentences which led to less confusion among the development team.

Table 6: Comparison of original and modified requirements

No. of requirements or problems / Document type	Original Requirements Document	Modified Requirements Document
Total No. of req.	33	58
No. of req. with Missing Contextual Information	22	1

The team shared one of their requirements documents with us. Some details of both the documents are available in Table 6. In the original document there were 33 requirements, which increased to 58 requirements, due to similar reasons as those in the previous case study. In the initial document, majority of the requirements (22 out of 33) were marked up by RAT as having missing contextual information. This is because many of the requirements had initially been written in passive voice. The team rewrote practically all of them into active-voice sentences, which clearly specified which system was responsible for which requirement. Here is a quote from the project's Requirements Lead regarding their experience with RAT:

"By having the tool critique the structure of the requirements, our deliverables were becoming clearer to the end users, resulting in a drop in requests for information from the development teams concerning our deliverables."

Another piece of feedback from the project team is that the tool forced team members from various locations to create a project-wide glossary that brought a "*sense of the appropriate terminology*" that should be used across the project. Once the glossary was created, it became easier for new team members to under-

stand the project and the goals the project was trying to accomplish.

Case Study 3 - Detecting faults in manually-generated interaction diagrams: Case study 3 is based on a single release cycle of medium-sized telecommunications project, containing 50 use cases. Before we introduced RAT, the project team had created an interaction diagram for each use case. The interaction diagrams were created because of two reasons – 1) the visual representation made it easy to discuss with business-minded stakeholders and 2) each diagram was used in the downstream design stage to make decisions on design of the systems such as bandwidth requirements needed between systems. Hence, it was critical that diagrams accurately represented use case text. The project team wanted us to verify how consistent the diagrams were with the use-case text. We followed a three-step process for this:

1. First, we ran the syntactic analysis of RAT on the use cases and corrected clarity-based issues. (While, we didn't discuss this earlier. RAT does support syntaxes for use cases.)
2. Then, we used diagram generation feature of RAT to generate interaction diagrams for each use case.
3. We compared the manually-created diagrams with those that RAT produced automatically.

The comparison highlighted error-prone nature of the manual process. The project team left out interactions from their manual diagrams that were specified in the requirements text (and therefore did appear in the RAT-generated diagram). We looked at 15 use cases and associated diagrams, and the manual diagrams approximately missed 23% of the valid interactions which showed up in the RAT diagrams. While, it had taken the team of 3 people 5 days to create 15 diagrams, it took one of the co-authors only 1 day to make the syntactic changes and generate the diagrams using RAT.

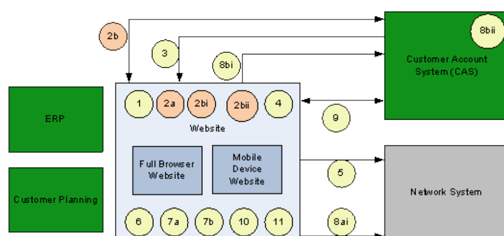

Figure 5: Manually Created Use Case Diagram. Interactions of a system with customer are depicted as circles in that system's boundary (for e.g., 1, 4).

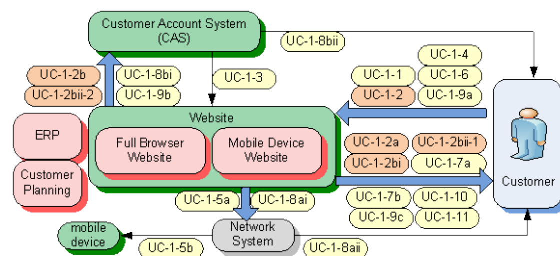

Figure 6: Automatically Generated Use Case Diagram by RAT

In some cases manual diagrams missed interactions because the diagram creator simply overlooked a requirement in the text. This is inevitable with large projects. In other cases, the requirements text was poorly written, obscuring the interaction for the diagram drawer; those requirements were improved with the help of RAT before diagrams were automatically generated. We do not know whether the manual diagrams would have been improved if they had been created from the improved requirements. But clearly, the combination of using RAT to improve requirements, and the automatic generation of diagrams from those requirements results in interaction diagrams and text that are more consistent with each other – and with the intentions of the stakeholders.

Let's illustrate the above in context of a specific piece of one of the use cases from this project in detail. Here's step 2 of a 9-step use case:

Step2: (main) <u>The customer selects the registration screen on the Website and enters his/her carrier customer account number</u>. The Website, which contains a repository of customer account numbers, will check the number.

2a) Number not found: If the customer account number is not found by the Website, then the Website alerts the customer via an on-screen message that the carrier customer account number entered is invalid.

2b) Number found: If the customer account number is found, then the Website validates that the customer does not have an enabled customer account by checking the customer information in the CAS.

2bi) If a customer account already exists, then the customer is alerted via an on-screen message indicating that the customer is already registered. The online message instructs the customer to return to the login page and log in using the customer account number and password.

2bii) If no customer account exists, then the Website requires the customer to enter in customer personal information. <u>The new customer account information is then sent to CAS for validation.</u>

The manually created interaction diagram for the entire use case is shown in

Figure 5. Each interaction in the figure is labeled using the identifier for the use case step. Based on recommendations made by RAT, a number of changes were made in the use case text, and then the use case interaction diagram was automatically generated by RAT (shown in Figure 6). The integration diagram generated by RAT had 6 interactions that were missed in the manual diagram. In step 2, in particular, there were two missed interactions in the manual diagram. The first missed interaction is shown in Figure 6 as UC-1-2 (from the customer to the Website). This represents the main branch of step 2 (shown underlined). The business analyst who manually created the diagram (

Figure 5) apparently missed drawing this interaction. The second missed interaction was UC-2b-ii-2 in Figure 6. This represents the second sentence (shown underlined) in Step 2bii. Since the original use case step was written in a passive form, the business analyst had missed the interaction between the Website and the CAS. RAT was able to flag this passive sentence. It was then rewritten by the analysts as: "*The Website sends the new customer account information to CAS for validation*", which is much clearer.

Case Study 4 - Finding defects in overall architecture diagram for a large project: Case study 4 involved a large government project, which had 5000 requirements, and involved integration of 24 existing systems with a new system. Before we introduced RAT, the project team had created an overall interaction diagram for the entire project. We followed the same three steps, as the previous case study) to study how consistent the requirements were with the text. As in the previous case study, the comparison also highlighted the error-prone nature of the manual process: a number of interactions were omitted from their manual diagrams. Specifically, in the manually created diagram, the project team simply missed depicting interactions of 5 systems with the new system. In addition, in this case study, the converse also occurred: 3 interactions showed up in the manual diagrams that weren't actually specified in the text. (Of course, these interactions did not show up in the RAT diagram.) This can happen when the diagram creator depicts an interaction that he/she thinks makes sense even though it has been left out of the text. It could be that the text is incorrect or the diagram is incorrect. The auto-generated diagrams also helped clarify that in some cases the text was defective because it specified no interactions at all for some of the systems defined in the glossary. (RAT shades the icon for systems in a special color when they are not participating in any interactions, since this generally corresponds to missing requirements).

Case Study 5 – Using RAT to detect terminological inconsistencies: The large project in the retail industry involved integrating 6 new Java-based systems (with about 70 sub-systems) with 19 existing systems. The requirements (about 3000) were written by different people and contained a number of terminological inconsistencies. Here are some examples. Requirements from one person referred to the main system as "*SWC System*", while other referred to it as just the system. Some requirements just had a generic "*user*" as the entity, while others had more granular users such as "*administrator*" and "*customer*". The team created a glossary, ran syntactic analysis to view undefined entities, and generated interaction diagrams to visualize the relationships between the entities. Based on the analysis results, the requirements were updated to make the entities consistent.

Case Study 6 - Using syntactic analysis to reduce requirements defects: The multi-year project consists of custom development of a software system for a mobile company with short 2 month release cycles. In the previous release (July-Aug 09), RAT was used by the business analysts to develop requirements and only 17 defects were manually detected in the peer review phase. This was much lower compared to the prior release, where 58 defects were detected in the peer review phase.

Case Study 7 - Glossary sharing and functional decomposition at large health benefits company: This project had a team of 30 experienced business analysts for documenting requirements, so RAT was used in a more sophisticated manner. A glossary of entities was initially created and shared by different team members to ensure consistent usage of terms. RAT was also leveraged for functional decomposition. There were 19 types of accumulators. Initial requirements were written with "accumulator" as the agent, so that the focus could be on the capability and not the specific accumulator. Later on, "accumulator" was removed from the glossary and RAT was used to mark-up where "accumulator" was used and replace it with more specific type of accumulator.

8. Lessons Learned

Based on our experiences of deploying RAT at over 500 real world projects, studying over 500 requirements documents varying in size from 25 to 5000 requirements, and talking to various practition-

ers who used our tools, here are seven important lessons learned:

Large textual requirements documents are still prevalent in practice for enterprise systems development: While newer approaches such as Agile development or Extreme programming propose doing away with large textual requirements specifications, it is hard to apply those approaches to large enterprise software development projects. A majority of the projects in this domain start with request for proposals, which can contain hundreds of requirements. After software development on a project is started, a significant amount of effort in expended into reviewing, rewriting and expanding such requirements. Given the aggressive timelines, practitioners welcomed a tool like RAT which quickly points them to issues in the documents and helps them generate visualizations. In fact, we have observed a more tailored methodology in practice, which combines the best principles of Agile and Waterfall models. While the requirements acquisition is done upfront, many principles of agile such as deeper stakeholder involvement, organization of scrums, rapid iterations and stakeholder reviews are often applied in design and development of the software, across small portions of the requirements documents.

Just training or providing general checklists is not enough, to make an impact in practice, automation is needed: Often times practitioners are required to review or write a large number of requirements in a short amount of time. Arming practitioners, with checklists of clarity and content based issues to look for (as proposed in inspection based approaches), is not a viable option. This is because it is not often feasible for practitioners to manually review large requirements documents either because of time constraints or the size of the documents. Most of the practitioners had undergone basic requirements training and were armed with check-lists of basic issues to detect. However RAT still found a number of issues in their documents, which they then corrected. We do believe that domain-specific checklists, such as the non-functional requirements checklist that we have created, can be valuable.

Practitioners will follow restrictions placed by a tool, only if they also make sense outside the context of the tool: RAT placed a number of restrictions on practitioners – use of controlled syntaxes and use of terms defined in glossaries. We have had over 500 projects (with at least 5 users per project) use this tool and most of the practitioners did not mind using the syntaxes proposed by us. This is because the restrictions are consistent with the best practices for documenting requirements. While we did propose specific syntaxes for different categories of requirements, the idea of having specific syntaxes rather broad guidelines that were present in previous training and books, resonated with the practitioners. Note that the initial version of RAT only supported the solution requirement syntax. We worked collaboratively with end-users and experts to develop the current syntax set.

RAT's analysis features are useful for requirements clarity-based issues, but more support for content-based issues is needed: As our case studies show, many of the problems that we discussed in the paper such as the use of ambiguous phrases, missing contextual information, terminological inconsistency and missing interaction requirements were detected and corrected because of RAT. A number of practitioners have also asked for automated support for judging the completeness of functional requirements documents. To that effect, we are working on a new tool called Process Model Requirements Gap Analyzer (ProcGap) [21] that compares requirements documents to domain-specific process models using natural language processing and semantic matching techniques.

There is some overhead in creating project-specific glossaries: For entity glossaries, projects generally reported that there was some overhead in creating the entity glossaries, especially for large projects where the glossaries could have hundreds of entities. However, they saw the value in creating such glossaries because of the fact that RAT helped them in maintaining terminological consistency and aided in functional decomposition. We have added a feature to the latest version of RAT that automatically detects noun phrases from the requirements documents using OpenNLP [15] and suggests them as potential entries to the entity glossary. For action glossaries, projects reported that there was an added burden. However, since many of the same actions are used across projects, it was possible to create a re-usable glossary of about 400 common actions, which we pre-package with RAT.

Extra effort needed to adhere to controlled syntaxes is offset by other benefits: While it does take a bit longer to draft requirements into syntaxes that align with the best practices suggested by RAT, projects reported that the overall time spent in requirements phase did not increase. This presumably because the extra time spent drafting the requirements is offset by reduced time in review/signoff phases.

Leveraging controlled syntaxes and use of glossaries makes it feasible to explore generation of down-

stream artifacts: Another benefit of using controlled natural language to write requirements is that it becomes easier to generate downstream artifacts such as high-level class diagrams or test cases. In [18], we explored generating high-level UML class diagrams based on structured information extracted from the requirements documents and the glossaries and applying additional heuristics. Another recent work [7] demonstrated how writing requirements using controlled syntaxes and following other principles espoused by RAT, such as writing requirements in controlled natural language, not using passive voice and maintaining glossaries, can lead to more automation in test planning.

9. Related work

There are three bodies of work that are relevant to this work -1) work on defining a language for requirements and 2) tools that analyze requirements written using a formal specification and 3) tools that automatically analyze natural language requirements.

Work on defining a language for requirements: IEEE/ANSI Standard 830-1998 [8] provides a number of best practices for writing software requirements and states that functional requirements typically start with "system shall". RAT provides a comprehensive set of syntaxes for writing requirements. Three of the requirements categories can be started with "system shall". Simplified Technical English (STE) [3] provides a set of grammar rules and a controlled dictionary for writing technical documentation and is applicable to both technical documentation and requirements. We believe that RAT has compatible and stricter restrictions specifically aimed towards documenting software requirements. RAT provides a set of controlled syntaxes and a number of grammar rules (for example, checking that the agents are specified and detecting passive voice) are enforced on the syntaxes. The use of user-defined glossaries helps users in restricting their terminology in the same vein as the STE controlled vocabulary.

Tools that automatically analyze requirements written in a formal specification: A number of tools focus on analyzing requirements written using a formal specification. Atlee and Buckley [4] present an approach to use model checking to verify requirements modeled using CTL (Computation Tree Logic). KaOS [22] leverages a temporal variant of first-order logic to formally represent goals, agents and actions. The formal definitions are used to find different kinds of inconsistencies, mainly in the behavior of the desired solution. Liaskos et al [17] have presented an approach for constructing high variability goal models. RAT, on the other hand, takes natural language as input and focuses on the quality of the documentation for a variety of issues ranging from terminological inconsistency, missing contextual information to missing requirements with the help domain ontologies and visualizations. One approach worth considering might be to enhance natural language parsing capabilities and the set of syntaxes of RAT that would allow the generation of a template for a formal specification like KaOS or CTL (to enable model checking discussed in [4]).

Table 7: Comparing RAT to other tools

Problem type/ Tool	RAT	QUARS	ARM	Raven
Find ambiguous phrases	Y	Y	Y	N
Readability tests (e.g. Coleman-Liau)	Partial (simple syntaxes)	Y	Y	N
Missing contextual information	Y	N	N	Partial (missing agents in use cases)
Terminological inconsistency	Y	N	N	N
Interaction analysis with help of visualization	Y	N	N	Partial (for use cases only)

Tools that analyze requirements written in natural language: This includes tools such as QUARS [12], ARM [26], Raven [20] and QUARCC[5]. A comparison between RAT and some of the other tools is shown in Table 7. Most of the tools provide support for phrasal analysis. Raven [20] provides some support for interaction analysis, but it is limited to use cases. QUARCC [5] provides support for finding dependencies between non-functional requirements. RAT builds on the relationships provided in QUARCC, to create a semantic model and provides a more general approach that allows us to detect both dependencies and missing requirements using a general purpose semantic reasoning engine. The reason that RAT is able to provide the broadest range of syntactic and semantic analyses is because of our assumptions of using controlled syntaxes and user-defined glossaries. These assumptions have been validated by deployment results.

Previous industrial experience in requirements engineering field: Our motivation for doing this work has been heavily influenced by previous industrial experience reports. Wever and Maiden [24] presented a study that reports survey results from 127 business analysts. One of their key findings is that

training for business analysts is ad-hoc and often not reinforced. As a result, the analysts often are not able to use the training at client implementations. We believe that is because requirements documentation, in practice, has been a softer discipline with more general guidelines and best practices and not enough prescriptive guidance. One of our key motivations in creating RAT was to ensure that practitioners have a concrete approach to follow to improve requirements documentation. As our case studies demonstrate, having some concrete set of syntaxes and analyses, has helped industrialize the requirements documentation and review process at Accenture. All business analysts are trained on the tool and using the tool at projects acts as reinforcement of what they learn.

Weigers [25] provides a case study of an industry project where the use of a glossary and separating out different types of requirements helped the project. That experience report was a key influencer of our design decision of having user-defined glossaries and different syntaxes for different types of requirements.

Wilson et al. [26] presented a set of findings based on their experience at NASA. They developed nine quality indicators for requirements. Some of the key ones that affected our work were:

- Imperatives: Their studies showed that "*shall*" and "*must*" are the top two words that explicate that something must be provided. All the functional requirement syntaxes supported by RAT require using "*shall*" or "*must*" as the modal word.
- Weak phrases and Options: Their studies showed that weak phases such as "*adequate*" and "*as appropriate*" and optional words such as "*may*" and "*can*" lead to ambiguity and misinterpretation .The words and phrases identified by then and other works such as [12], were used to see the problematic phrase glossary.

10. Implementation details

Architecture: RAT has been implemented as an extension to Microsoft Office can be installed as an "add-in" for Word and Excel. We use the office libraries in .NET to access the textual requirements and analyze them. The semantic engine is implemented using Jena and we use XML to transfer data back and forth from .NET to Jena, which is implemented in Java.

Execution times: Execution times for RAT on Intel Dual Core 2.13Ghz CPU with 2GB RAM for different sizes of requirements documents are shown in Table 8. These times were found reasonable by all RAT users.

Table 8: Execution times for RAT

No. of Req.	Syntactic Analysis	Semantic Analysis
200	43 sec.	67 sec.
1000	223 sec.	322 sec.
2000	543 sec.	1003 sec.
3000	771 sec.	1470 sec.

11. Conclusions and Future work

In this paper, we presented a tool called RAT, which automates the analysis of requirements against a set of best practices. Since we introduced the tool in practice within Accenture, it has been deployed at over 500 projects. We presented case studies of using RAT at seven enterprise software projects across different domains, which showed how using RAT led to better documented and more complete requirements. Projects with less experienced business analysts mainly focused on structuring requirements better, while more mature teams were able to use more sophisticated features such as functional decomposition and interaction analysis.

We also compared RAT to some other tools that have previously been developed to analyze natural language requirements. Our conclusion was that the restrictions placed by controlled syntaxes and user-defined glossaries, which have been validated by successful deployment of this tool, allowed us to provide a much broader set of analyses than any other tool.

We have adopted an iterative approach for introducing RAT in the practice. Earlier versions focused on just clarity-based issues. We received requests from users for automating analysis based on content of the requirements. To that extent, we created a semantic framework to help automate some manual tasks such as interaction analysis. Our future work includes focusing on automatically mapping requirements documents to domain specific process models to judge their completeness [21]. In addition, we are also looking at generating downstream design artifacts such as architecture diagrams and test cases.

12. References

[1] ANTLR, http://www.antlr.org/

[2] Antón, A.I., Goal-Based Requirements Analysis, 2nd International Conference on Requirements Engineering, 1996, 136-144.

[3] ASD STE, http://www.asd-ste100.org/pagina1.htm
[4] Atlee, J.M. and Buckley, M.A., A Logic-Model Semantics for SCR Software Requirements. ISSTA, 1996, 280-292.
[5] Boehm, B. and In, H. 1996. Identifying Quality-Requirement Conflicts. *IEEE Softw.* 13(2), 1996, 25-35.
[6] Mike Cohn, "User Stories Applied", 2004, Addison Wesley
[7] Güldali, B., Funke, H., Jahnich, M., Sauer, S., and Engels, G., Semi-automated Test Planning for e-ID Systems by Using Requirements Clustering, Automated Software Engineering Conference, 2009, 29-39.
[8] IEEE Recommended Practice for Software Requirements Specifications. IEEE/ANSI Standard 830-1998,
[9] Jacobson, I., Booch , G. and Rumbaugh, J. 1999. The Unified Software Development Process, Addison-Wesley Professional.
[10] Jain, P., Verma, K., Kass, A., and Vasquez, R.G. 2009. Automated review of natural language requirements documents, Indian Software Engineering Conference, 2009, 37-46.
[11] Jena - Semantic Web work, http://jena.sourceforge.net/
[12] Lami G. 2005. QuARS: A tool for analyzing requirements. CMU Software Engineering Technical Report.
[13] Letier, E., and Lamsweerde, A., Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering Proceedings of 12th ACM International Symposium on the Foundations of Software Engineering, 2004, 53-62
[14] NIST 800-53, http://csrc.nist.gov/publications/nistpubs/800-53-Rev2/sp800-53-rev2-final.pdf
[15] OpenNLP, http://opennlp.sourceforge.net/
[16] OWL, http://www.w3.org/TR/owl-features/
[17] Liaskos, S., Lapouchnian, A., Yu, Y., Yu, E., and Mylopoulos, J., On Goal-based Variability Acquisition and Analysis, the 12th International Conference on Requirements Engineering, 2006, 76-85.
[18] Sharma, V., Sarkar, S., Verma, K., Panayappan, A., and Kass A. 2009. Extracting High-Level Functional Design from Software Requirements. APSEC 2009, 35-42.
[19] SPARQL, http://www.w3.org/TR/rdf-sparql-query/
[20] Raven Software, www.ravensoft.com
[21] Thayasivam, U., Verma, K., Kass, A., and Vasquez, R., Automatically Mapping Natural Language Requirements to Domain-Specific Process Models, IAAI 2011.
[22] van Lamsweerde, A., Letier, E., and Darimont, R. 1998. Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Trans. Softw. Eng. 24(11).
[23] Verma, K., and Kass, A., Requirements Analysis Tool: A Tool for Automatically Analyzing Software Requirements Documents, 7th International Semantic Web Conference, 2008, 751-763.
[24] Wever, A., and Maiden, Neil: What are the day-to-day factors that are preventing business analysts from effective business analysis? RE 2011: 293-298
[25] Wiegers, K. E. 2003 Software Requirements, Microsoft Press.
[26] Wilson, W., Rosenberg, L., and Hyatt, L. 1997. Automated Analysis of Requirement Specifications. ICSE 1997.
[27] YACC, http://dinosaur.compilertools.net/yacc/