Requirements Analysis Tool: A Tool for Automatically Analyzing Software Requirements Documents

Kunal Verma and Alex Kass

Accenture Technology Labs, San Jose, CA, USA
{k.verma,alex.kass}@accenture.com

Abstract. We present a tool, called the *Requirements Analysis Tool* that performs a wide range of best practice analyses on software requirements documents. The novelty of our approach is the use of user-defined glossaries to extract structured content, and thus support a broad range of syntactic and semantic analyses, while allowing users to write requirements in the stylized natural language advocated by expert requirements writers. Semantic Web technologies are then leveraged for deeper semantic analysis of the extracted structured content to find various kinds of problems in requirements documents.

Keywords: Requirements analysis, Domain Ontologies, Semantic Analysis, SPARQL.

1 Introduction

Requirements documents for large software systems are very lengthy and complex. Moreover, gathering correct and accurate requirements from customers requires a deep understanding of both the customer's business needs and the technical issues involved, which are often not communicated clearly or at all. These challenges often result in poorly written requirements that are unclear, verbose, and even inconsistent. These poorly written requirements result in extensive rework, which in turn drives up project costs, extends timelines, and decreases customer satisfaction (see, for example, [1]). Many organizations have processes and best practices in place for reviewing requirements documents at various stages of the project lifecycle to detect problems that can lead to extensive rework. This manual review process, however, is painstaking, time consuming and often fails to uncover even the commonly occurring problems in requirements documents. Hence, more automated solutions to assist in the review of requirements documents are needed.

A number of researchers have recognized this need and have proposed a number of tools and techniques to automatically analyze requirements (see, for e.g.,QUARCC [1], QuARS [4], KaOS [3]). There also exist commercial products (e.g. Raven [7]) that provide assistance of this nature. However, these solutions have not achieved a high level of adoption for two reasons. 1) Many solutions (e.g., KaOS [3]) require users to write requirements in formal notations. This restriction is not feasible in practice because requirements documents are written by semi-technical analysts and have

A. Sheth et al. (Eds.): ISWC 2008, LNCS 5318, pp. 751–763, 2008.

to be signed-off by business executives. Hence, the communication medium is still natural language. 2) Solutions (e.g. QuARS [4], Raven [7]) that allow users to write requirements in natural language are limited in the kinds of analysis they can perform. For example, QuARS [4] only supports phrasal analysis and Raven [7] is restricted to supporting only use cases. For broad adoption, there is need for a tool that takes a more holistic approach and allows a broad range of analyses over natural language requirements.

In this paper, we report on a tool, called the *Requirements Analysis Tool* (RAT) that automatically performs a wide range of syntactic and semantic analyses on requirements documents based on industry best practices, while allowing the user to write these documents in natural language. RAT encourages users to write in a standardized syntax, a best practice, which results in requirements documents that are easier to read and understand. RAT performs a syntactic analysis by using a set of glossaries to identify syntactic constituents and flag problematic phrases. An experimental version of RAT also performs semantic analysis using domain ontologies and structured content extracted from requirements documents during syntactic analysis. The semantic analysis, which uses semantic Web technologies, detects conflicts, gaps, and inter-dependencies between different sections (corresponding to different subsystems and modules within an overall software system) in a requirements document. Hence, the key contributions of this paper are as follows:

1. We present a tool for enforcing requirements best practices, while allowing users to write requirements in natural language with the help of controlled syntaxes and user-defined glossaries.
2. We show how the structured content extracted from requirements can be used for a deeper semantic analysis with the help of a semantic engine and domain ontologies.

The rest of the paper is organized as follows. Section 2 presents an overview of common problems that occur in requirements documents, and also presents some best practices that RAT supports for avoiding those problems. Section 2 also presents an overview of what RAT does. Sections 3 and 4 present the details of syntactic and semantic analysis performed by RAT. A discussion of the related work is presented in Section 5. Implementation details and an early user evaluation are presented in Section 6. Finally, our conclusions are presented in Section 7.

2 Overview of Common Requirements Problem and Best Practices for Writing Requirements

Despite the challenges associated with writing and analyzing requirements documents, there is general agreement about the kinds of problems that plague these documents. We give an overview of these problems followed by an overview of best practices for writing good requirements.

2.1 Common Requirements Problems

Here is an overview of common problems that occur in a requirements document:

Requirements that are easy to misunderstand. This problem is caused by the use of ambiguous terms, terminological inconsistencies or the use of non-standard syntaxes for documenting requirements.

- *Use of ambiguous terms.* Consider the requirement "The Payroll Database shall respond to all queries quickly". It is unclear what "quickly" means in this context (1 ms, 1 sec, 10 sec, etc.) and this may lead to a mismatch between the expectations of the stakeholder and the interpretation of the developer, ultimately leading to rework and/or customer dissatisfaction.
- *Terminological inconsistency.* Consider the following requirements: 1) "The Order Entry System shall allow users to view all orders placed in a day" and 2) "The Order Processing System shall generate daily reports". In this case, the requirements analyst uses two different terms to refer to the same system, leading to confusion.
- *Use of non-standard syntaxes such as missing agent.* Consider the requirement "Shall have the ability to generate profit reports". It is unclear which agent (e.g., which system or sub-system) shall have the ability to generate the profit reports. In this case, the designers/developers may try to guess which system should have that functionality, potentially leading to rework at a later phase.

Requirements that are inconsistent. This problem is caused by requirements either conflicting with each other or with some policy or business rule. Consider the following two requirements: "The Payroll database shall authenticate all requests using LDAP" and "The payroll database must respond to all requests within 1 millisecond". In this case, it may not be feasible to satisfy this requirement if the LDAP server takes more than 1 millisecond to process each request. If such a defect is not detected early on, then it is possible that the infeasibility is only realized when the LDAP server and the Payroll database have been designed and implemented, leading to large costs in rework.

Requirements that are incomplete. This problem is caused by missing some requirements of a certain type. For example, it is often the case that performance requirements for a system are omitted either due to the lack of knowledge among the stakeholders or because the requirements analysts fail to elicit them. This leaves the technical designers and developers to make design choices about the software system, which may or may not meet the stakeholder's approval.

2.2 Best Practices and Overview of Approach

One of best-known and authoritative sources for software requirements [10] suggests the following best practices for writing requirements to avoid to some of the problems mentioned in the previous sub-section:

1. Write complete sentences that have proper spelling and grammar.
2. Use active voice (For e.g., " The System shall send an e-mail" instead of "E-mail will be sent")

3. Use terms consistently and as defined in the glossary. Do not use different phrases to refer to the same thing. (For e.g., Do not use Order Processing System and Order Entry System to refer to the same system)
4. Write sentences in a consistent fashion starting with the agent/actor, followed by an action verb, followed by an observable result. (For e.g., "The System (agent) shall generate (action verb) a report")
5. Clearly specify the trigger condition that causes the system to perform a certain behavior. (For e.g., "If the user enters the wrongs password , the system shall generate an error message")
6. Avoid the use of ambiguous phrases

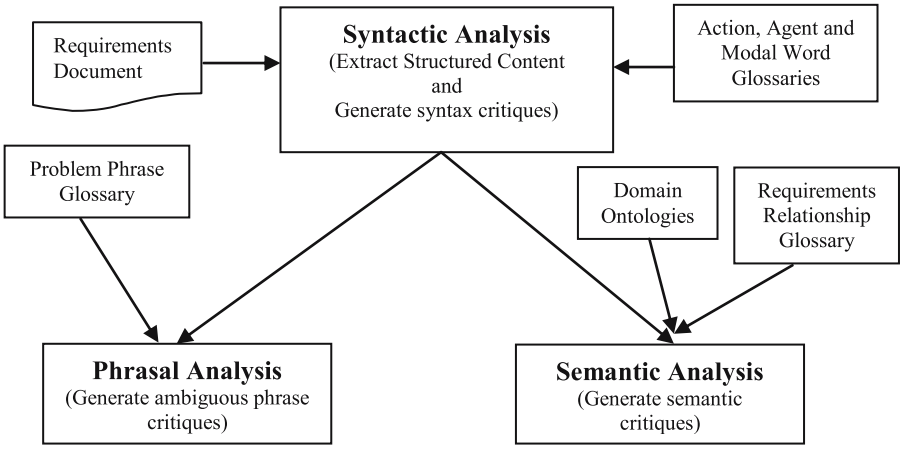

Fig. 1. Analysis Overview

These best practices are designed to deal with syntactic problems such as those caused by the use of ambiguous terms, terminological inconsistencies or the use non-standard syntaxes and can be detected by syntactic analysis of the requirements. However, other problems caused due to inconsistent and incomplete requirements require a more domain specific best practices, where domain knowledge needs to be applied to find the occurrence of such problems.

As shown in Figure 1, our approach starts by syntactically analyzing requirements documents and extracting structured content from the requirements document about each requirement. We leverage a set of controlled syntaxes and user defined glossaries for the syntactic analysis. Then the structured content is leveraged for phrasal and semantic analysis. For the phrasal analysis, RAT uses a problem phrase glossary. For semantic analysis another glossary called the requirements relationship glossary is used. We will discuss the different analysis techniques and the different glossaries in the ensuing sections.

3 Syntactic and Phrasal Analysis

In this section, we will describe our approach for syntactic analysis. We will first cover the set of controlled syntaxes supported by RAT, and then we will discuss the user-defined glossaries, followed by our approach for extracting structured content

from requirements documents. Finally we will provide a brief overview of phrasal analysis.

3.1 Controlled Syntaxes for Writing Requirements

RAT supports a set of controlled syntaxes for writing requirements. This set includes the following:

Standard Requirements Syntax. This is the most commonly used syntax for writing requirements and is of the form:

StandardRequirement: <agent> <modal word> <action> <rest>

Where, <agent>, <action>, <modal word> are phrases in their respective glossaries and <rest> is the remainder of the sentence and can consist of agents, actions or any other words and is defined as:

<rest>: [<anyword> | <agent> | <action>]*

Here is an example of an standard requirement: “The system shall generate profit reports.” Here “System” is the agent, “shall” is the modal word and “generate” is the action and “profit reports” is the rest of the requirement.

Conditional Requirements Syntax. There are a number of conditional requirements supported by RAT. For brevity, we will only discuss the most common condition syntax:

<“if”> <condition> <“then”> <StandardRequirement>

For example, consider the following requirement: “If the user enters the wrong password, then the system shall send an error message to the user.” In the case, “user enters the wrong password” is the condition. The part after “then” is treated like a standard requirement.

Business Rules Syntax. RAT treats all requirements that start with “all”, “only” and “exactly” as business rules. An example is: “Only the members of payroll department will be able to access the payroll database”.

3.2 User-Defined Glossaries to Parse Documents

RAT uses three types of user glossaries to parse requirements documents: agent glossary, action glossary and modal word glossary. An agent entity is a broad term used to denote systems, sub-systems, interfaces, actors and processes in a requirements document. The agent glossary contains all the valid agent entities for the requirements document. It captures the following information about each agent: name of the agent, immediate class and super-class of the agent and a description of the agent. The class and parent of the class field of the glossary are used to load the glossary into the semantic engine. Table 1 shows some sample entries for an agent glossary.

Table 1. Agent Glossary

Agent Name	Description	Class	Super-Class
Web Server	The Web Server for the project.	System	Agent
Payroll Employee	The payroll dept employee	User	Agent
SAP System	The SAP system for the project	System	Agent
Order Parts Process	The ordering parts process	Process	Agent

The action glossary lists all the valid actions for the requirements document. It has a similar structure to the agent glossary. Examples of actions are "generate", "send" and "allow". The modal word glossary lists all the valid modal words in the requirements document. Examples of modal words are "must" and "shall".

3.3 Parsing and Extracting Structured Content

In this section, we will provide a high level overview of how controlled syntaxes and glossaries are collectively used to parse the requirement. We have a deterministic finite automata based approach to parse the requirements, extract structured content and generate error messages. We shall define the approach informally:

- *S* is the set of states, and each state denotes a certain stage in the parsing of requirements. *s* is the start state denoting the start of requirement.
- The entire English language is considered as the alphabet. However only certain members of the alphabet (for e.g., "if", "when", "all", <agent>, <action>, <modal word>) lead to transitions between states.
- *G* is the set of goal states. Only a transition sequence (for e.g. <agent><modal word><action><".">) that denotes a controlled syntax can lead to a goal state from the start.
- *E* is the set of error states and each error state has an associated error message. Consider the following requirement: The Messaging System shall allow users to view previously sent messages. If the agent "Messaging System" is not present in the agent glossary, the transition will end in an error state corresponding to missing agent and RAT will mark it with a "Missing Agent" critique.

For brevity, we will not provide more details of the automata here. As a requirement is parsed, structured content is extracted for each requirement. This structured content is used for both the syntactic and semantic analyses. The extracted structured content contains:

- Type of requirement syntax (standard, conditional, business rule)
- All agents, actions and modal words for all the requirements
- Different constituents of conditional requirements.

3.4 Finding the Use of Problematic Phrases

Certain phrases frequently result in requirements that are ambiguous, vague or misleading. The problematic use of such phrases has been well documented in the requirements literature. A classification of problematic phrases is presented in [4]. [10] provides a list of such words and how to correct requirements that use them. We have

collected an extensive list of problematic phrases from a number of sources and stored it in a user-extensible glossary called the problem phrase glossary. We believe that the real power of the tool is in creating user-defined, domain-specific glossaries. We have been working with experts in a number of domains to find problematic phrases in their respective domains. Consider this requirement from a financial domain: The Reporting System shall generate new account reports daily. While this requirement seems correct and precise, it turns out that "daily" is ambiguous, since it does not specify what time the reports should be generated (end of day, start of the day, mid-day).

3.5 User Interfaces

The current version of RAT makes the analyses described above available to the user through two distinct user interfaces: First, there is an interactive interface, called the *Requirements Checker*, which is depicted in Figure 2. This interface is similar to Microsoft Word's built-in spelling and grammar checker. The Checker provides an explanation of each problem, one by one, and suggestions for fixing it. For instance, when a problematic phrase is detected, the checker presents the context, explains why the phrase causes problems, and provides a number of suggestions for improving the requirement.

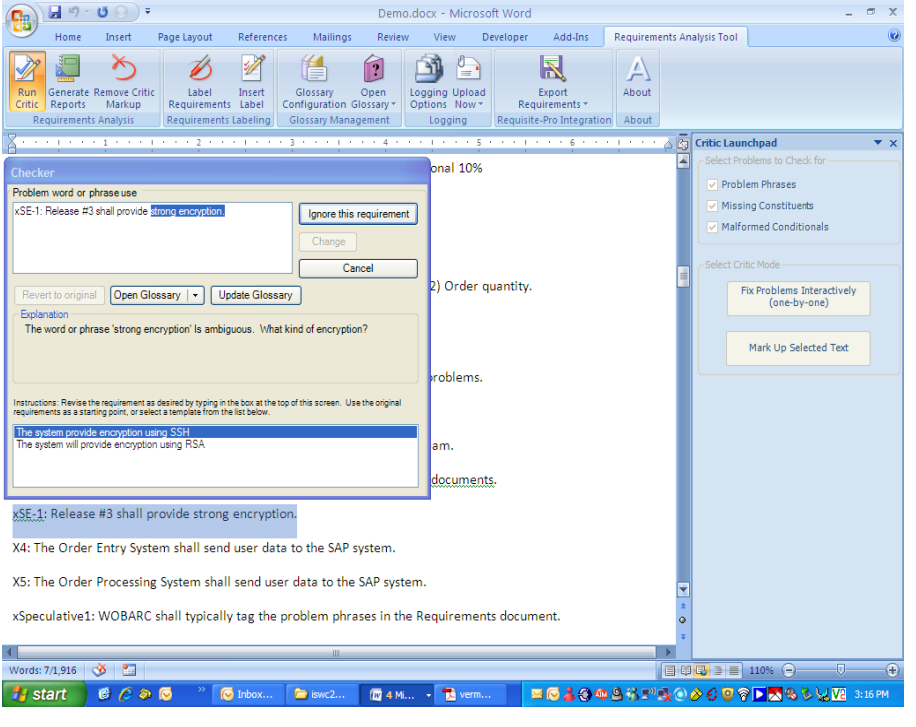

Fig. 2. Interactive Requirements Checker

A second interface, called the *Requirements Tagger*, which is shown in Figure 3, runs through the entire document, and marks it up using Word's margin-note comment feature. It tags all problems it can find with comments that are color-coded to represent the kind of problem detected. The Tagger also highlights the requirements text, underlining the agent and action in each requirement through the use of the corresponding glossaries.

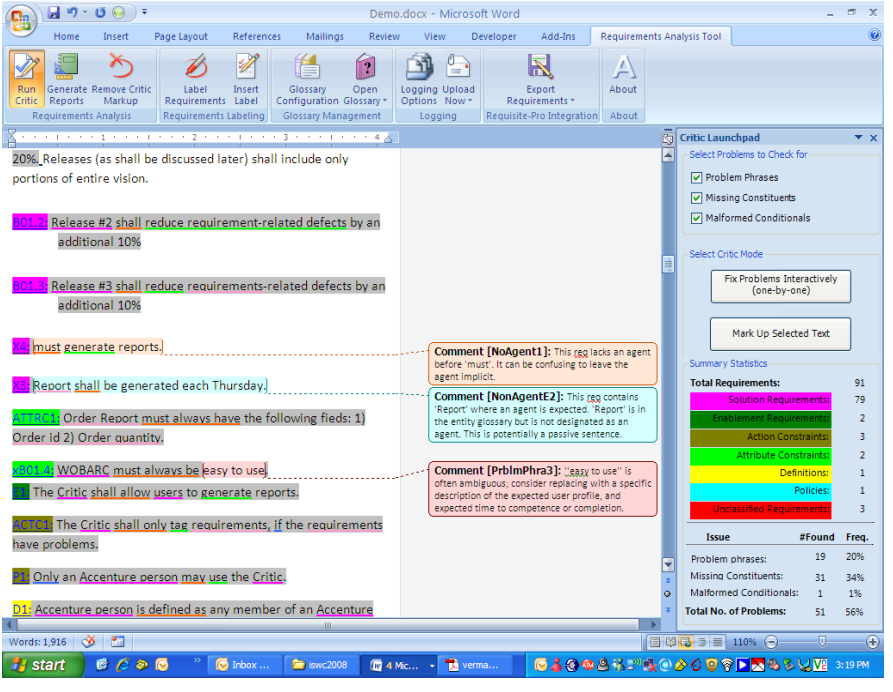

Fig. 3. Comments generated using Requirements Tagger

4 Semantic Analysis of Requirements Documents

Experts use a lot of domain knowledge to study requirements documents for various problems, such as dependencies and conflicts between requirements. Much of this knowledge can be captured using domain-specific ontologies to provide a deeper analysis of requirements document. The crux of our approach is to create a semantic graph for all requirements in the document based on the extracted content. In RAT, users can use the requirements relationship glossary, which is shown in Table 2 to enter domain specific knowledge. The requirements relationship glossary contains a set of requirement classification classes, its super-class, keywords to identify that class and the relationships between the classes. Some samples requirement classes for non-functional requirements and their relationships are shown in Table 2, which essentially states that security based requirements (encryption, authentication) affect time based requirements (query time and response time).

Table 2. Requirements Relationship Glossary

Class	Super-Class	Keywords	Relationships
Security	NonFunctional		Affects:Time
Authentication	Security	Password, token, authentication, Kerberos	Increases:ResponseTime
Encryption	Security	encrypt, SSH, RSA, DSA	Increases: Response Time
Time	NonFunctional		
Query Time	Time	Query time, querytime	
Response Time	Time	response, respond	

For creating the requirements relationship glossary, we have leveraged some of the non-functional classes and relationships presented in QUARCC [1]. For functional requirements, we have been working with experts in various domains such as finance. As is the case with all other glossaries, this glossary is also user editable. Our fundamental belief is that once a requirements document is transformed into a semantic graph (represented as an OWL ontology), users can query for different kinds of relationships that are important to them. Here are the steps that RAT uses to create the semantic graph (graphically depicted in Figure 4) from a requirements document:

1. Load the core requirements ontology in the Semantic Engine. The core requirements ontology is basic requirements ontology with different types of requirements formats (standard, business rule and conditional) and the information that each of them contain.
2. Using the agent and action glossaries to create the agent and action classes and instances of agents and actions.
3. Using the requirement relationship glossary to create requirements classification classes and their relationships.
4. Using the extracted structured content (enhanced by requirement classification information from Table 2) to create instances of requirements.

4.1 Requirements Dependency Analysis

There are many dependencies between the requirements in a document. However, due to the large size of the documents and the fact that different people may have written different sections of the document, it is difficult to identify all these dependencies. Not being able to uncover some of these dependencies can lead to a number of problems such as: 1) Conflicting requirements in the requirements documents that may lead to flawed design and development decisions and 2) Not scheduling some lower priority requirements for development, especially those that may be required for implementing higher priority requirements.

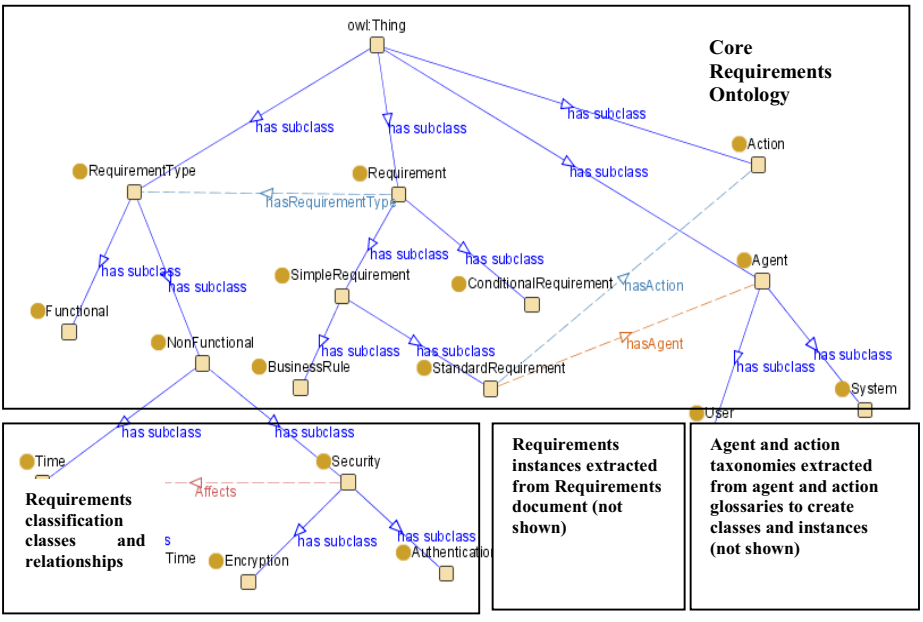

Fig. 4. Core Requirements Ontology and how it is built from different glossaries and requirements document

Let us illustrate a dependency with the help of an example. It is common knowledge among technical architects that increasing the security profile of a system affects it performance. This is modeled in the ontology in Figure 5. Let us see how RAT can leverage this knowledge. Consider the following two requirements: 1) The Web Server shall encrypt all its responses using SSH and 2) The Web Server shall have a response time of 5 milliseconds or less. Once these requirements are entered into Jena, then we can then issue the following SPARQL query to check for any dependencies between them.

```
select ?req1, ?req2 where
{ ?req1  hasRequirementType ?type1 .  ?req2  hasRequirementType  ?type2 .
 Affects domain ?type1 .  Affects range ?type2 .
?req2 hasAgent ?agent2 .
?req1 hasAgent ?agent1  filter( ?agent1 = ?agent2)})
```

This query returns all requirements (for the same agent) that have requirement-types, which "affect" each other. This is illustrated using Figure 5. In this case, the first requirement (WebServer_EncReq) has requirement-type Encryption, which is a sub-class of Security. Similarly, the second requirement (Web-Server_ResTimeReq) has requirement-type Response Time, which is a sub-class of Time. There is a relationship in the ontology that says that Security affects Time. Thus, the query returns the fact that requirements 1 affects requirement 2.

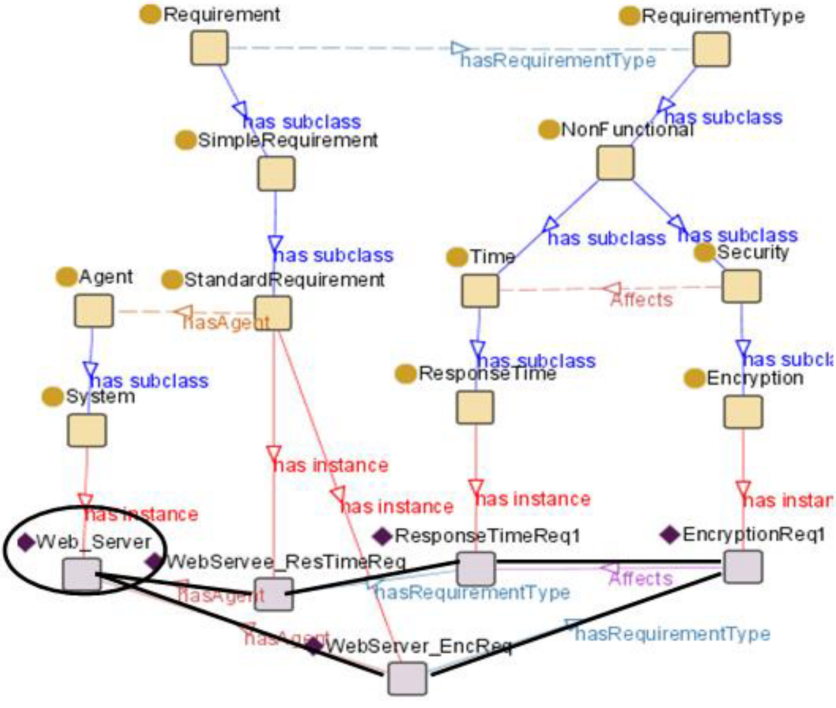

Fig. 5. An ontology snippet (drawn using the Jambalaya plug-in) showing the dependency (using thick lines) between the encryption and response time requirements for the agent Web_Server (circled in figure)

4.2 System and Agent Role Based Analysis

Given the large size of requirements documents, it is useful request reports that highlight various properties and relationships. Using the agent glossary, where we require the users to classify the different agents either as systems, users or any other category, RAT supports the following queries:

- *Systems that are interacting with each other.* We have created a query which returns all requirements with a system as the primary and secondary agent. Consider the following example: "The Web Server shall send the vendor data to the SAP System." In this case, the Web Server is the primary agent and the SAP System is the secondary agent and both of them are classified as systems in the agent glossary, so RAT can deduce that they are interacting with each other.
- *Systems that are missing certain kinds of non-functional requirements.* Non-functional requirements are often overlooked in requirements documents. Hence, we have created a query to identify and return all systems that are missing a non-functional requirement in one or more of following categories: security, performance, reliability, usability, integration and data requirements.

- *Interacting systems that do not have compatible security profile.* This query checks if a system has similar security protocol requirements with another system that it interacts with. Consider the case where one system has a requirement for supporting a certain kind of encryption, while an interacting system does not have any requirement for the same kind of encryption. In this case, RAT points out that there might be a security based incompatibility.

Another important type of analyses is based on the role of the agent. While we do not have any pre-defined queries for this category, we believe that users can enter queries that may be useful in their domains. One way to proceed would be to capture information in the domain ontologies about which agents are allowed to perform which actions (e.g. only purchasing manager may approve new suppliers). Another variation of a similar analysis is "Separation of duty", as outlined in Sarbanes Oxley.

5 Related Work

One approach of analyzing requirements is to treat them as a form of pseudo-code, or even a very high-level language, through which the requirements analyst is essentially beginning to program the solution envisioned by the stakeholders. In such a case, it would be possible to build tools that analyze requirements, just like compilers analyze software programs. This observation has been shared by a number of researchers in this space and a number of tools have been proposed (comprehensive survey of tools is available at [8]). For these tools, the representation of requirements has ranged from free text, use of structured notation, to formal representation using logic. The analyses provided by these tools range from phrasal analysis, use of custom code to analyze small domain models to formal analysis general purpose reasoners. Most of the tools that don't restrict the syntax of requirements are limited to phrasal analysis, while the tools that require formal representation of requirements use general purpose reasoning engines for analyzing the requirements. QuARS [4] presents an approach for phrasal analysis of natural language requirements documents. QUARCC [1] uses a specialized model for identifying conflicts between quality (non-functional) requirements. While we have leveraged both these works (QuARS for classification of problem phrases and QUARCC for creating relationships between non-functional requirements), neither of them provide the broad range of semantic and syntactic analyses that we discussed in this paper. KaOS [3] presents an approach for using Semantic nets and temporal logic for formal analysis of requirements using a goal based approach. While they can perform a broad range of reasoning, their system needs the requirements to be inputted in a formal language.

6 Implementation Details and Early User Evaluation

RAT has currently been created as a plug-in for Microsoft Word 2003/7 using Visual Basic for Applications. The Glossaries are currently also Word documents. The Semantic Engine leverages the reasoning capabilities of Jena [5] and is implemented in Java. We use Protégé to create the OWL [6] ontologies. The Jena semantic engine is used for the reasoning and SPARQL [9] is used for the query language.

Currently, we are piloting an early version of this tool without the full semantic analysis engine, at four client teams. In these pilots, 15 requirements analysts have used RAT to process more than 10,000 requirements. Users have reported that the tool is helping them catch many real world problems. They have reacted positively to the controlled syntaxes and critiques, as the tool helps them follow best practices and comply with a number of industry standards, resulting in much clearer requirements. Some early users requested support for more syntaxes. Users also requested more advanced semantic analysis to detect conflicts, dependencies and missing requirements. The syntactic extensions have been incorporated in the latest release. The semantic analysis discussed in this paper has been implemented in an experimental prototype, and will be made available to users in the next release of RAT.

7 Conclusions and Future Work

In this paper, we have presented RAT: a tool for automatically analyzing requirements documents. We have presented an approach that allows users to write requirement using a set of controlled syntaxes advocated by requirements experts, and user-defined glossaries. This approach allows RAT to extract structured content from the requirements text, which is used for syntactic and semantic analysis using semantic Web technologies. A valuable feature of our approach is that users can extend the analysis by creating user-defined glossaries, creating domain ontologies or writing their own queries. Our future work includes creating a collaborative framework for users to enter domain knowledge.

References

1. Boehm, B., In, H.: Identifying Quality-Requirement Conflicts. IEEE Software 13(2), 25–35 (1996)
2. Boehm, B., Papaccio, P.: Understanding and Controlling Software Costs. IEEE Trans. on Software Eng. 14(10), 1462–1477 (1988)
3. Lamsweerde, A.V., Darimont, R., Letier, E.: Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Trans. Software Eng. 24(11) (1998)
4. Gnesi, S., Lami, G., Trentanni, G.: An automatic tool for the analysis of natural language requirements. CSSE Journal 20(1), 53–62 (2005)
5. Jena – A Semantic Web Framework for Java, http://jena.sourceforge.net/
6. OWL – Web Ontology Language, http://www.w3.org/TR/owl-features/
7. Raven: Requirements Authoring and Validation Environment, http://www.ravenflow.com
8. Robinson, W.N., Pawlowski, S.D., Volkov, V.: Requirements interaction management. ACM Comput. Surv. 35(2), 132–190 (2003)
9. SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query/
10. Wiegers, K.: Software Requirements. Microsoft Press (2003)