

Automated Review of Natural Language Requirements Documents: Generating Useful Warnings with User-extensible Glossaries Driving a Simple State Machine

Prateek Jain¹, Kunal Verma², Alex Kass², Reymonrod G. Vasquez²

¹Kno.e.sis
Wright State University,
Dayton, OH 45435
prateek@knoesis.org

²Accenture Technology Labs,
50 West San Fernando,
San Jose, CA 95113
{k.verma,alex.kass,
reymonrod.g.vasquez}@accenture.com

ABSTRACT

We present an approach to automating some of the quality assurance review of software requirements documents, and promoting best practices for requirements documentation. The system we describe – the Requirements Analysis Tool (RAT) - has been deployed and is currently being used in pilot projects with large and complex requirements documents. Preliminary results indicate a reduction in time needed to review documents and reduction in requirements defects as well as a change in the way users think about writing requirements. Our approach allows users to write requirements in natural language instead of an artificial formalism. RAT enforces requirements documentation best practices such as using standardized syntaxes and internally-consistent use of terminology. It supports the use of user-extensible glossaries to define terms. The formalism driving RAT is a state-machine, which is used to classify requirements into types based on keywords and then verify that the requirements follow one of the best practice syntaxes supported by the tool. It generates helpful warning messages explaining where requirements are not following best practices and suggests ways to rectify.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications – *tools*.

General Terms

Design, Standardization

Keywords

Requirements Engineering, Requirements Analysis, Requirements Analysis Tool, State Machine.

1. INTRODUCTION

Requirements are the foundation of any software development

effort. Without clear, high quality requirements the effort is unlikely to satisfy the customer's needs. This is particularly true for complex, enterprise software projects. In such projects, the requirements documents are typically lengthy and play especially challenging yet important communication role: they are written by one team, and then after customer approval, are utilized by a number of other teams in downstream roles such as architects, developers, and testers, who provide technical designs and working software but may have limited opportunity to talk directly with the customer.

To support the quality of this communications process, a number of best practices have been proposed by experts (see, for example, [2]), which expound a number of principles such as:

- Writing each requirement as a single, separate sentence [11]
- Associating a unique identifier with each requirement.
- Writing complete active-voice sentences which clearly specify the actor/agent and the action.
- Maintaining terminological consistency and clarity by restricting action and actor descriptions to terms that are clearly defined in a glossary.

While the practices, which we'll discuss in detail later, are relatively easy to state and understand, it seems fairly difficult for teams of analysts to consistently apply them throughout requirements documents with thousands of requirements. We have developed a tool called Requirements Analysis Tool (RAT) that helps users adhere to best practices in requirements documentation. RAT processes a document, identifies the requirements sentences, and generates warning messages informing the user when the requirements do not conform to best practices. RAT is designed to handle documents that mix requirements text with supporting text and figures. It only applies its analysis to sentences that are preceded by identifiers in a specific format (using such identifiers to label requirements is itself a best practice). It ignores all other text and figures.

Let us illustrate some common issues that occur in requirements documents and how RAT helps users deal with them using two examples.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISEC '09, February 23–26, 2009, Pune, India.

Copyright 2009 ACM 978-1-60558-426-3/09/02...\$5.00.

Example 1 - Leaving out key contextual information

Consider the following requirement:

- "xReport1: Must generate profit reports." (Note: In this paper, we use the convention of marking problematic requirements with an identifier beginning with 'x').

In order to reduce confusion about who should generate the profit reports, best practices dictate the requirement be written as:

- "Report1: The order processing system must generate profit reports."

For the requirement above (*xReport1*), RAT will give the following warning message to the user: "*This requirement lacks an agent before 'must'. It can be confusing to leave the agent implicit.*" This prompts users to employ the best practice of explicitly mentioning the agent/actor in each requirement.

Example 2 - Failure to standardize terms within a requirements document:

Consider the following two requirements:

- "UserData1: The Order Processing System shall provide an interface to enter user data."
- "xUserData2: Order Entry System shall send user data to the SAP System."

In a large requirements document written by different people, it becomes hard to deduce whether both the requirements are referring to the same system or two different systems. To deal with this issue, requirements documentation best practices encourage practitioners to define all entities in the project glossary and to only use the defined entities in the requirements. In the scenario where "*Order Processing System*" is in the project glossary, but "*Order Entry System*" is not, RAT will generate the following error message for the user: "*This requirement contains 'Order Entry System' where an agent is expected, but 'Order Entry System' is not in the entity glossary.*" This warning will warn the user that they are using a non-standard or undefined entity and they can make the appropriate changes.

The value of automatically analyzing requirements to detect potential quality problems has been recognized as an important one for quite some time. A number of research projects ([1], [4] and [5]) and even some commercial products [6] have addressed aspects of the problem, as we shall discuss later in the paper. However, the problem has proven to be very challenging and none of the existing systems have resolved enough of those challenges to achieve broad adoption:

One of the key challenges is that formats which are easiest for analysts to generate (natural language text) are not the easiest to automatically process. Automating the understanding of requirements written in completely unrestricted natural language remains intractably difficult. On the other hand, requiring analysts, who are more business savvy than trained in formal modelling, to write requirements in an artificial formalism has proven unfeasibly burdensome for most real-world development processes. Maintaining this balance between giving flexibility to users to write in natural language, while still being able to analyze the requirements, was a key design issue for us. Our approach to the problem of processing requirements has been to restrict both syntactic structure and lexical choices *in a manner that is driven by requirements engineering best practices*. By imposing only

restrictions that make sense independent of our tool (because they make requirements clearer for *human* readers too) we believe (and our earlier pilots have validated this) that we have produced an approach that is feasible in real-world software-development contexts. Through a combination of 1) requiring that phrases used to describe entities and actions be defined in explicit glossaries, and 2) restricting syntaxes to relatively simple forms recommended by requirements experts, we have been able to reduce the problem of processing natural-language requirements to one that can be handled by analysis techniques similar to programming languages. While the vagaries of natural language prevent us from using compiler-compiler tools which are used to analyze programming languages such as ANTLR [8] and Yacc [7], we use an extensible state machine approach which is heavily influenced by them.

Our key contributions are:

1. We have identified a restricted form of natural language that corresponds to accepted best practices for documenting requirements. This restricted natural language is rich enough to clearly express a full range of requirements types, yet is regular enough to facilitate automatic analysis.
2. We have developed an approach to automatically analyze these requirements, using techniques adapted from compiler-compiler tools.

RAT is currently being piloted by approximately 11 projects engaged in commercial software-development work. Preliminary results suggest a reduction in the time to review requirements, a reduction in requirements defects, and an improvement in analyst's perception of clear requirements.

In the rest of the paper, we will describe our approach, which allows RAT to generate warning messages for requirements that do not conform to best practices. We will discuss related work in Section 2. The best practices supported by RAT are discussed in Section 3. We will describe the structure of user-defined glossaries used by RAT in Section 4. Then, we will describe the different best practice syntaxes supported by RAT in Section 5. We will then describe our approach for analyzing the requirements in Section 6 and the different kinds of warning messages that are generated by RAT in Section 7. Section 8 presents results from early project deployments of RAT. Finally, the conclusion and future work is outlined in section 9.

2. Comparison with Related Work

In this section, we will summarize the functionality provided by RAT and then we will compare how the functionality compares with previous work in this area. RAT provides the following functionality to the users:

- RAT helps users write requirements in a set of standardized natural language syntaxes based on best practices. For example, RAT requires most types of requirements to be written in active voice, explicitly specifying what needs to be done, and by whom. If the users deviate from the set of syntaxes, RAT generates helpful warning messages to help them rectify the problem.
- RAT helps users maintain terminological consistency by automating the checking of phrases that represent entities and

actions in each requirement against user-defined glossaries for those entities and actions.

- In the spirit of [4], RAT also helps users avoid terminology known to be frequent sources of ambiguity or misinterpretation.
- Finally, it generates a structured representation of the natural-language requirements text, which can serve as the basis for more advanced, semantic processing, such as conflict detection and impact analysis. Semantic processing is not the focus of this paper. For interested readers, our ideas on semantic processing of requirements documents are outlined in [3].

Two bodies of work are relevant to this work – 1) tools that automatically analyze requirements and 2) compiler-compiler tools. We will distinguish this work from both those bodies of work.

Tools that automatically analyze requirements: A number of tools such as QUARCC [1], QUARS [4], KaOS [5] and Raven [6] have been developed for automatically analyzing requirements. QuARS [4] performs only the phrasal analysis of requirements. QuARCC [1] uses a special model for conflict detection between quality (non-functional) requirements. KaOS [5] performs a formal analysis of requirements using a goal based approach, but requires the input to be in formal language. Other tools such as those discussed in [9] and [10], perform analysis of requirements using formal techniques and models. None of these tools critique requirements written using a controlled natural language.

The tool that is closest to our approach is Raven [6], which allows users to write requirements in controlled form of natural language and critiques them. However, its analysis is restricted to use cases and it cannot support the broad variety of syntaxes that are supported by RAT.

Compiler-compilers tools: Compiler-compiler tools such as YACC [7], ANTLR [8] provide a rich framework for analyzing a set of controlled syntaxes. These tools are aimed at programming languages, where the syntaxes are less flexible than natural language. Our state-machine based approach, which was heavily influenced by these tools, allowed us to have the flexibility required to analyze a controlled natural language and leverage user-defined glossaries.

3. Best Practices Supported by RAT

To support the quality of this communications process, a number of best practices have been proposed by experts [1] [11] [12] which expound a number of requirements documentation principles such as:

- Write complete sentences rather than bulleted buzz phrases.
- Use entities and action consistently and as defined in the project’s glossary.
 - Do not use different phrases to refer to the same entity. (For example, do not use *Order Processing System* and *Order Entry System* to refer to the same system)
- Avoid using phrases, such as “easy to use”, whose meaning is subjective and leads to ambiguity.

- Write requirements sentences in a consistent fashion using a standard set of syntaxes with each syntax-type corresponding to and signaling different kinds of requirements.
 - For example, write solution requirements as simple active sentences, preferably with the syntax: <agent/actor> “shall” <action>: “*The System shall generate a report.*”
 - Another example: Enablement requirements should follow another standard syntax: <user type> “shall be able to” <capability to be provided>. Example: “*The finance department users shall be able to generate reports.*”

In the rest of the paper, we will discuss how RAT helps users adhere to these best practices.

4. Structure of User-Defined Glossaries

The corner-stone of our approach is utilizing user-defined glossaries that define all the entities and actions to be used in the requirements document. The glossaries serve two main purposes:

1. Glossaries serve as a reference for valid entities and actions. The fact that RAT finds all instances where undefined terms are used makes RAT very attractive to users as it helps them adhere to best practices.
2. Secondly, from a RAT-centric point of view, the terms from the glossaries are used as placeholders in the syntax set and help make the analysis of natural language requirements tractable.

In this section, we will briefly describe the two types of user-defined glossaries used by RAT for syntactic analysis – 1) Entity Glossary and 2) Action Glossary.

The entity glossary (Table 1) is used to capture all entities in a solution such as systems, sub-systems, interfaces, users, processes, records, and data fields in a requirements document. We differentiate between agent and non-agent entities by defining agents as entities that can perform actions. For example, “Order Proc. System” is an agent since it can perform an action such as processing an order, but “Total Sales Value” is not.

Table 1 Snapshot of Entity Glossary

Agent Descriptor	Explanation	Is Agent
Order Proc. System	System for processing orders	Yes
Web Server	HTTP Web Server	Yes
Finance Dept User	User from finance dept	Yes
Chemical Containers	Containers that store acids	No
Customer Standing	The status of the customer	No
Total Sales Value	Item value inclusive of sales tax	No

The action glossary (Table 2) lists the valid actions for the requirements document.

Table 2 Snapshot of Action Glossary

Action Descriptor	Descriptor
process payroll	Action for processing of payroll.

inform administrator	Action for sending e-mail notification to administrator
send contracts data	Action for transfer of contract data.
display	Rendering an item on screen.

For the rest of this paper, the following terms will be used to refer to glossary entries:

- Agent phrase: Entry in entity glossary that is marked as an agent
- Entity phrase: Non-agent entry in entity glossary
- Action phrase: Entry in action glossary

Please note that RAT also has a user-editable problematic and vague phrase glossary which finds the use of ambiguous phrases in requirements. For more details of the problematic phrase glossary, please refer to [3].

5. Enforcing Best Practice Syntaxes

As we discussed in Section 3, a number of experts have put forward best practices for requirements documentation. Our aim was to synthesize the syntactic recommendations from this literature, into a set of controlled syntaxes that our tool could enforce. Enforcing the use of these syntaxes serves two purposes. First, it directly promotes use of best practice syntaxes, and second, it makes the problem of analyzing the natural language requirements, to promote the other documentation best practices, tractable. In this section we present the set of controlled syntaxes that the tool promotes. To create this set of syntaxes, we followed these design principles:

1. Align each syntax-type with a conceptual category of requirements. For example, we have a different syntax-type for solution (requirements that define the capability of a system) and enablement (requirements that define a capability that a system must provide to a user) requirements. Aligning a syntax-type with a conceptual category makes it easier for the reader to quickly understand the intent of each requirement.
2. To cover as broad a range of real-world requirements as possible. We chose these syntaxes based on a number of academic and industry sources such as [2] and feedback from practitioners who used an earlier version of RAT [3], which only supported the solution requirement syntax. Sometimes adopting RAT's controlled syntaxes involves transforming a requirement that an expert author may already believe to be clear. However, in almost all cases, experts have agreed that the transformation improves clarity. And all the requirements we've seen to date can be converted to one of these syntaxes without distorting the intent.
3. Design each syntax-type in such a way that it can be easily distinguished from others. This is done by using a unique set of keywords as part of each syntax-type. For example, the use of "shall" or "must" denotes a solution requirement, whereas "shall be able to" denotes an enablement requirement.

In this section, we will discuss all the six syntaxes that are supported by RAT. We will provide details on the conceptual category of each syntax-type and also give some examples.

5.1.1 Solution Requirement

This is probably the most common requirement category. Put simply, this kind of requirement communicates something that the proposed system (or some sub-component) must do. The solution requirement is expressing that someone (or, often, some system) is responsible for carrying out some action. The recommended syntax is a simple, *active voice* sentence, which includes the words "shall" or "must" or "will" in between the exact phrase defined to describe the agent, and the exact phrase defined to describe the action that this agent is required to perform.

The formal representation for the syntax is as follows.

Solution Requirement: <Agent Phrase> <"shall" | "must" | "will"> <Action Phrase>

The phrases "shall" or "must" are included to enforce the operations to be performed by the Agent Phrase.

Examples of Solution Requirements

- SA1: The Order Processing System shall process orders every 2 hours.
- SA2: The Web Server must inform administrator of failed login attempts.

5.1.2 Enablement Requirement

These requirements express a capability that the proposed system must provide, but do not specify what/who will provide this capability. There are two kinds of enablement requirements.

Enablement requirements that do not mention the system: That is appropriate for user requirements, where it is premature to specify what will provide the capability.

- The recommended syntax is a simple, active voice sentence, which includes the words "shall be able to" or "must be able to" in between the exact phrase defined to describe the agent, and the exact phrase defined to describe the action that this agent is required to perform

Enablement Requirement: <Agent Phrase> <"shall" | "must" | "will"> <"be able to"> <Action Phrase>

Examples

- ER1: The user must be able to display the PDF rendition of associated documents.
- ER2: The payroll system shall be able to deduct loan amounts from paychecks.

Enablement requirements that mention a high level capability provided by a system to a user: Such requirements are appropriate for situations where stakeholder knows a capability that a certain system/agent must provide to certain type of user.

- The recommended syntax is a simple, active voice sentence in the form

Enablement Requirement: <Agent Phrase> <“shall” “must” “will”> <“allow” | “permit”> <Agent Phrase> <“to”> <Action Phrase>

Examples

- ER3: Inventory management system shall allow users to add items.
- ER4: Payroll system shall permit users to change direct deposit profiles.
- ER5: Order Processing System must permit administrator to view daily transactions.

Ultimately, every capability that a solution provides must be supported by actions that components of the solution take to provide the capability. An enablement requirement commits to provide users with a certain capability, and it also commits the requirement writers to eventually provide the more specific requirements that specify which actions will be performed (and by which system) in order to provide that capability.

5.1.3 Action Constraint

This syntax is used to express a constraint on how the solution or component of the solution is expected to behave. There are two kinds of action constraints

Action constraints that express a constraint on how the solution (or a component of it) is allowed to behave: These requirements are used to impose conditions or constraints on actions performed by agents

- The recommended syntax is a simple, active voice sentence in the form:

Action Constraint: Agent Phrase <“shall” | “will” | “may”> <“only” | “not”> Action Phrase <“when” | “if”> condition.

Examples

- AC1: The account management system shall only close an account if the current balance is zero
- AC2: The authentication system shall not grant access when identity-verification level is less than 8.9.

Action constraints that express business rules that constrain how agents in the business take actions: These requirements are used to constraint the actions that an agent may or may not perform.

- The recommended syntax is a simple, active voice sentence in the form:

Action Constraint: “Only” <Agent Phrase> <“may” | “may be”> <Action Phrase>.

Examples

- AC3: Only child-friendly pets may be placed in old age homes.
- AC4: Only payroll employees may access the payroll database.

5.1.4 Attribute Constraint

This kind of requirement is used to express constraints on attributes and attribute values.

- The recommended syntax is a simple, active voice sentence in the form:

Attribute Constraint: <Entity Phrase | Agent Phrase> <“must” <“always” | “never” | “not”> <“be” | “have”> <Value Phrase>.

Examples

- ATR1: Customer standing must always be one of the following: 1) Gold 2) Silver 3) Bronze.
- ATR2: Chemical containers must not be stored in subzero temperature.
- ATR3: The customer must never have non US address in records.

5.1.5 Definition

This kind of requirement is used to define non-agent entities.

- The recommended syntax is a simple, active voice sentence in the form:

Definition: <Entity Phrase | Agent Phrase> <“is” | “will be”> <“defined as” | “classified as”> <Entity Phrase> .

Examples

- Def1: Total sales value is defined as total item value plus sales tax.
- Def2: A graduate student with a grade-point average above 3.5 is classified as an honors student.

5.1.6 Policy

This kind of requirement is used to write policies that must be adhered to by the solution.

- The recommended syntax is a simple, active voice sentence in the form

Policy: <Entity Phrase | Agent Phrase> <“is” | “is not”> <Action Phrase>

Examples

- P1: Sales tax is computed on in-state shipments.
- P2: Sales tax is not computed on interstate shipments.

In the syntax for Conditional Requirement, any of the 6 requirements types mentioned above can be written in a conditional format by enclosing them in “if-then”. We will not show that for brevity.

6. Analyzing Requirements

RAT uses a two phased approach (shown in Figure 1) for analyzing requirements documents: 1) Lexical Analysis and 2) Syntactic Analysis. The lexical analyzer converts a requirement statement into a set of tokens and also classifies it as a particular type of a requirement. The syntactic analyzer uses state machines for analysis of syntaxes. The syntactical analyzer has a different machine for each syntax type and uses the classification from the lexical analyzer to decide which state machine to use. In this section, we will cover both lexical and syntactic analysis. We will also give an example of state machine based syntactic analysis for solution requirements.

6.1 Lexical Analysis

A lexical analyzer analyzes a requirement sentence and breaks it into tokens. The token types include the following:

1. Agent phrases: Terms defined as agents in entity glossary
2. Entity phrases: Non-agent terms in entity glossary
3. Action phrase: Terms in action glossary
4. Modal phrase: A set of fixed modal phrases such as “shall”, “will” and “shall be able to”.

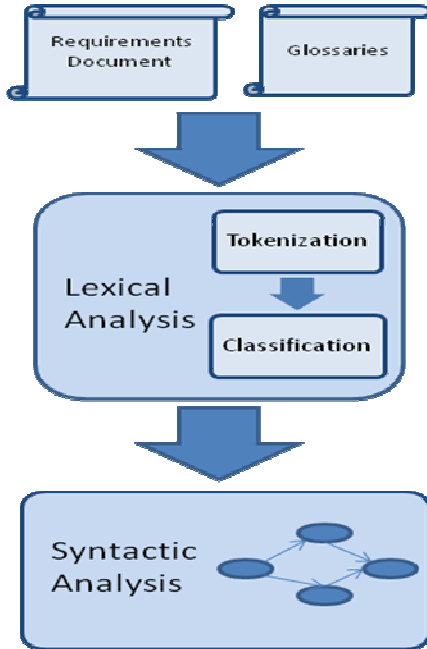


Figure 1: Architecture Requirements Analysis.

Tokenization is illustrated with an example. Consider the following requirement:

SA1: The SAP System shall send vendor data to the order processing system.

Table 3 shows the phrases of this requirement statement as well as the token types identified for them.

Table 3 Phrases of requirements statement

Phrase	Token Type
SA1:	Label
The SAP System	Agent phrase
Shall	Modal phrase
Send	Action phrase
Vendor data	Entity phrase
To	unknown
The order processing system	Agent phrase

After tokenization, a requirement is classified into one of the categories based on the modal phrase. Our grammar is designed such that each syntax type can be identified by its modal phrase. This classification is used by syntax analyzer to decide which state machine should be used to validate a requirement. In the example above, the keyword “shall” calls for validation using the solution state machine(Figure 2).

6.2 Syntactic Analysis

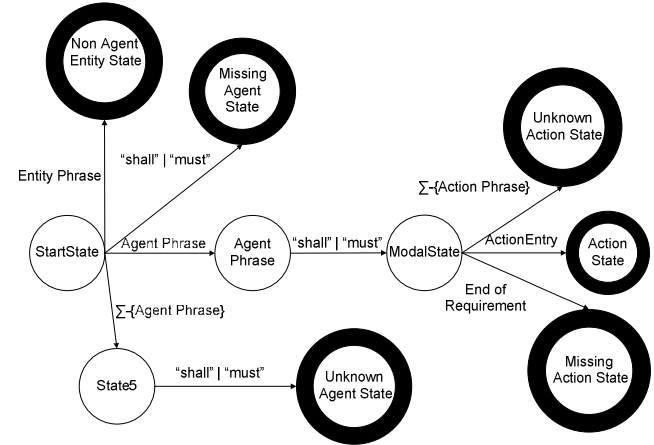


Figure 2 Solution State Machine

The syntactic analyzer uses a state machine based approach for analyzing requirements. For each syntax-type, the syntactic analyzer maintains a separate state machine. The tokenized requirements from the lexical analysis are run through the corresponding state machine for their syntax-type. A requirement is treated as syntactically correct when the state machine successfully transitions from start state to a valid final state. However if the state machine enters an error state, then a warning message is generated.

We will explain how the state machine approach works with the help of solution requirements. The syntax for solution requirements was defined in Section 5.1.1 as:

<Agent Phrase> <“shall” | “must”> <Action Phrase>

Corresponding to this syntax, the state machine for solution requirements (shown in Figure 2) is a 6-tuple $(\Sigma, S, s_0, \delta, F, E)$, which can be defined informally as follows

- Σ is the alphabet, which consists of set of modal phrases and phrases from the entity and action glossary
- S is the set of states based on the syntax = {Start State, Action State, Modal State, Agent State, Missing Agent State, Missing Action State, Unknown Action State, Unknown Agent State, Non Agent Entity State}
- s_0 is the start state. In this case, $s_0 = \text{“Start State”}$.
- δ is the transition function and is shown in Figure 2
- F is the set of final states denoting that the token stream was based on the syntax. In this case, $F = \{\text{Action State}\}$.
- E is the set of error states denoting that the token stream did not follow the syntax. In this case, $E = \{\text{Non Agent Entity}$

State, Missing Agent State, Unknown Action State, Missing Action State, Unknown Agent State}

For every error state, there is a pre-defined warning message that is displayed to the user. Table 4 illustrates the mapping of error states to warning messages. The statement of the warning message also explains why the requirement deviates from best practices.

Let us illustrate the use of the state machine with the help of an example. Consider the following requirement:

SA1: *The SAP System shall send vendor data to the order processing system.*

Using the state machine transitions in Figure 2, the token stream for the requirement will end up in “Action State” and it will be treated as a valid requirement. Consider another requirement: **SA2:** *shall display error messages in new window.*

For this requirement, the state machine will halt in Missing Agent State (as it lacks an agent phrase in the beginning). Using the mapping shown in the Table 4, the corresponding error message “*This requirement lacks an agent before ‘shall’. It can be confusing to leave the agent implicit.*” is generated.

Table 4 Error States and corresponding error messages

Error State	Warning Message
Missing Agent State	<i>This requirement lacks an agent before <variable at which error occurs>. It can be confusing to leave the agent implicit.</i>
Unknown Action State	<i>This requirement contains '<variable at which error occurs>' where an action is expected, but '<variable at which fault occurs>' is not in the action glossary.</i>
Unknown Agent State	<i>This requirement contains '<variable at which error occurs>' where an agent is expected, but '<variable at which fault occurs>' is not in the entity glossary.</i>
Non Agent Entity State	<i>This requirement contains '<variable at which error occurs>' where an agent is expected. '<variable at which error occurs>' is in the entity glossary but is not designated as an agent.</i>
Missing Action State	<i>This requirement lacks an action before '<variable at which error occurs>'. It can be confusing to leave the action implicit.</i>

State machines for other syntaxes are not discussed for brevity.

7. Generation of helpful warning messages for requirements

As we have mentioned before, RAT helps users deal with typical issues that occur in requirement documents. In principle, there are three main kinds of conceptual issues that RAT addresses:

1. Missing contextual information.
2. Use of non-standard terms.
3. Deviations from best practice syntaxes.

With the help of state machine approach outlined in Section 6.2, RAT generates a number of warning messages for each of these three categories of issues (shown in Figure 3). In this section, we will discuss the different types of warning messages that are generated by RAT.

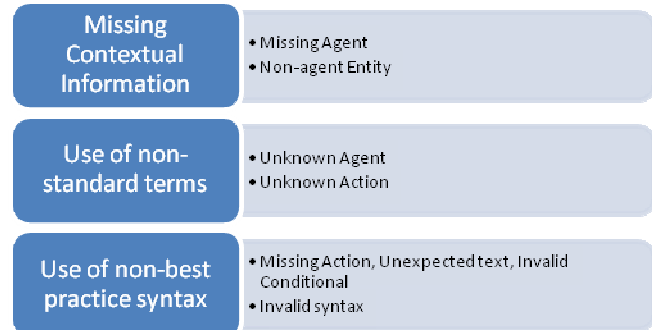


Figure 3: Issues and associated warning message types

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 issue typically occurs in two main forms:

1. **Requirements that are missing the agent/actor:** RAT deals with such requirements with the “Missing Agent” message. Consider the following requirement:

- “xReport1: must generate profit reports.”

RAT generates a warning message stating: “*This requirement lacks an agent before ‘shall’. 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:

- “xReport1: The order processing system must generate profit reports.”

2. **Requirements that are written in passive voice:** RAT deals with such requirements with “Non Agent Entity” warning message. Consider the following requirement:

- “xDiscount1: If the person’s age is greater than 60, the age factor discount shall be applied.”

RAT generates the following warning message for this requirement: “*This requirement contains ‘Age factor discount’ where an agent is expected. ‘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.”

Use of Non-standard terms: Another source of confusion is using non-standard terms that are not defined in the project glossaries. This issue also occurs in two main forms:

1. **Requirements that use a non-standard name to refer to an entity:** RAT deals with these requirements using the “Unknown agent” message. Consider the following two requirements:

- “Req3: The Order processing system shall provide an interface to enter user data.”
- “xReq4: The Sales Entry System shall send user data to the SAP system.”

If “*Order Processing System*” is in the Entity Glossary but “*Sales Entry System*” is not, RAT will generate the following warning message, “*This requirement contains 'Sales Entry System' where an agent is expected, but 'Sales Entry System' is not in the entity glossary.*”

The message should help the user consider whether a new agent (“*Sales Entry System*”) should be added to the entity glossary or whether the requirement should be written as:

- Req4: The Order Processing System shall send user data to the SAP system.”

2. **Requirements that use a non-standard name for an action:** RAT deals with requirements using the “Unknown Action” message. Consider the following two requirements:

- “login1: Order Processing System shall report failed login attempts.”
- “xlogin2: Order Processing System shall report unsuccessful authentication attempts.”

If “*report failed login attempts*” is in the Action Glossary while “*report unsuccessful authentication attempts*” is not, RAT will generate the following warning message “*This requirement contains 'report unsuccessful authentication attempts' where an action is expected, but 'report unsuccessful authentication attempts' is not in the action glossary.*”

The message should help the user consider whether a new action (“*report unsuccessful authentication attempts*”) should be added to the action glossary or whether the requirement should be changed to the following:

- “login2: Order Processing System shall report failed login attempts.”

Deviation from best practice syntaxes: This issue also occurs in two main forms:

1. *Minor deviation from a supported syntax:* The user tries to follow one of the best practice syntaxes but makes some minor mistake. RAT has a number of warning messages to deal with situation such as “Invalid conditional”, “Missing Action” and “Unexpected Text”. For brevity, we will not give an example of these.
2. *Use of an unsupported syntax:* This is the case where the user doesn’t use any of the supported syntaxes. In this case, RAT gives a, “Invalid Syntax” warning message that cautions users that they are not using a supported syntax.

A screen shot of RAT performing analysis of requirements is provided in Figure 4.

8. Early Deployment Results

We have not yet run formal studies or controlled experiments to assess the tool. However, we have piloted the tool on 11 real industrial software projects. All these teams used RAT to mark up their requirements and made changes to the requirements on the basis of the warning messages generated by RAT. The teams typically consisted of 2-3 business analysts and a supervisor. In most cases, each analyst was responsible for creating a set of requirements documents, and the supervisor was responsible reviewing the quality of requirements. While we will present detailed case studies in a later paper, early feedback shows the automated review of requirements has provided the following benefits:

- **10-30% reduction in time required to transform notes taken in interview sessions to well formed requirements.** Requirements are typically collected by business analysts in face-to-face meetings or JAR sessions, where they quickly jot down notes. At the end of the meetings, the analysts are responsible for converting the notes into well-formed requirements. Because business analysts were guided by warning messages generated by the tool, they were able to quickly convert requirements into one of supported syntaxes and reported 10-30% reduction in time in this task.
- **30-50% reduction in time needed to review requirements.** The tool marks up large requirements in a few seconds (1000 requirements take 30 seconds or less). Reviewers can use the tool to mark up the requirements text and then quickly go through the comments to see the syntactic issues. They can then spend more time on reviewing the content, as opposed to earlier, where they spent a lot of time dealing with syntactic issues.
- **5% estimated reduction in overall budget, due to expected reduction in requirements defects and associated reduction in rework.** This is a preliminary estimate, based on surveys of the business analysts (not the RAT project team), who applied their past experience to conclude that the higher quality requirements produced using RAT would reduce the number of requirements that would be missed or misinterpreted, and would ultimately lead to less rework. We will be able to verify these numbers in a few months by comparing actual project spending with original budget estimates and spending from previous releases of similar sizes that did not use RAT.

An earlier version of the tool only supported the solution requirement syntax. The other syntaxes were added based on feedback from the practitioners who were using the tool on real requirements documents. The warning messages were also made more descriptive based on the feedback from the users.

9. Conclusion and Future Work

In this paper, we have presented RAT, a tool that helps users write requirements that adhere to best practices by generating useful warning messages. In order to achieve this, the tool uses a set of controlled natural language syntaxes and a state machine based approach. This tool has been extremely well received by users and we have received great qualitative and quantitative feedback. We will be exploring two directions in the future. Firstly, we will be looking to add or modify the set of syntaxes based on user feedback. Secondly, we will be exploring use of semantic techniques for analyzing the content of the requirements [3].

10. References

- [1] Boehm, B. and In, H. 1996. Identifying Quality-Requirement Conflicts. *IEEE Softw.* 13, 2 (Mar. 1996), 25-35. DOI=<http://dx.doi.org/10.1109/52.506460>
- [2] Wiegers, K. E. 2003 *Software Requirements*, Microsoft Press.
- [3] Verma, K., Kass, A. 2008. Requirements Analysis Tool: A Tool for Automatically Analyzing Software Requirements Documents. In *Proceedings of the 7th International Semantic Web Conference* (Karlsruhe, Germany, October 26 - 30, 2008) .
- [4] Lami G. QuARS: A tool for analyzing requirements. Software Engineering Technical Report CMU/SEI-2005-TR-014, Software Engineering Institute, USA, September 2005.
- [5] van Lamsweerde, A., Letier, E., and Darimont, R. 1998. Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Trans. Softw. Eng.* 24, 11 (Nov. 1998), 908-926. DOI=<http://dx.doi.org/10.1109/32.730542>
- [6] Raven Software, www.ravensoft.com

[7] YACC, <http://dinosaur.compilertools.net/yacc/>

[8] ANTLR, <http://www.antlr.org/>

[9] Anderson, T., de Lemos, R., Fitzgerald, J.S. and Saeed, A. On Formal Support for Industrial-Scale Requirements Analysis. Workshop on Theory of Hybrid Systems Springer-Verlag Lecture Notes in Computer Science Vol. 736pp - 426-451, 1993. ISSN: 3-540-57318-6

[10] Yoo, J., Kim, T., Cha, S., Lee, J., and Son, H. S. 2005. A formal software requirements specification method for digital nuclear plant protection systems. *J. Syst. Softw.* 74, 1 (Jan. 2005), 73-83. DOI= <http://dx.doi.org/10.1016/j.jss.2003.10.018>

[11] Young, R. R. 2000 *Effective Requirements Practices*. Addison-Wesley Longman Publishing Co., Inc.

[12] IEEE Recommended Practice for Software Requirements Specifications. IEEE/ANSI Standard 830-1998, Institute of Electrical and Electronics Engineers, 1998.

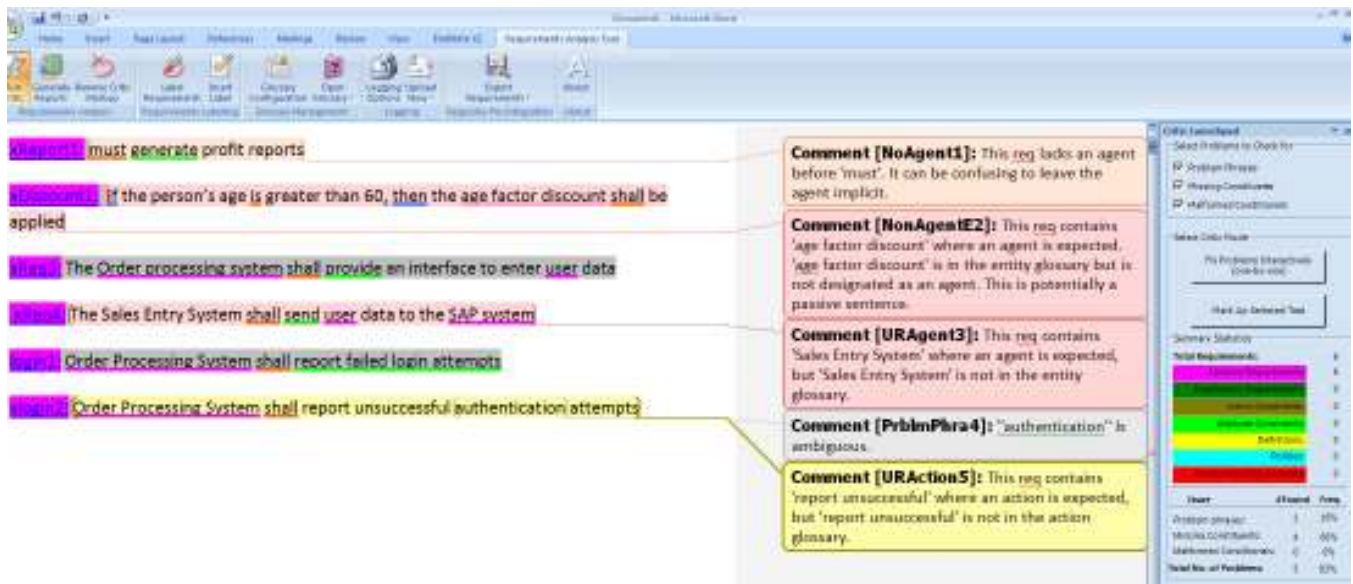


Figure 4 Screen shot of analysis of requirements by RAT