

A Framework for Identifying and Analyzing Non-functional Requirements from Text

Vibhu Saujanya Sharma, Roshni R. Ramnani, Shubhashis Sengupta
Accenture Technology Labs
Accenture, Bangalore, India
{ vibhu.sharma, roshni.r.ramnani, shubhashis.sengupta }@accenture.com

ABSTRACT

Early identification of Non-Functional Requirements (NFRs) is important as this has direct bearing on the design and architecture of the system. NFRs form the basis for architects to create the technical architecture of the system which acts as the scaffolding in which the functionality of the same is delivered. Failure to identify and analyze NFRs early-on can result in unclassified, incomplete or conflicting NFRs, and this typically results in costly rework in later stages of the software development. In practice, this activity is primarily done manually. In this paper, we present a framework to automatically detect and classify non-functional requirements from textual natural language requirements. Our approach to identify NFRs is based on extracting multiple features by parsing the natural language requirement whereby the presence of a certain combination of and relationship among the features uniquely identifies the requirement as an NFR of a particular category. These features are specified as pattern based rules which can be specified in a human readable language through the use of a domain specific language that we have defined. This enables great ease and flexibility in creating and extending rules. Our approach has been implemented as a prototype tool and here we also present the results of applying our approach on a publicly available requirement corpus.

Categories and Subject Descriptors

D.2.1 [Requirements/Specifications]: Methodologies, Tools

General Terms

Algorithms, Experimentation

Keywords

Requirement Analysis, Natural Language Processing, Non-Functional Requirements, NFRs

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

TwinPeaks'14, June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2848-7/14/06...\$15.00
<http://dx.doi.org/10.1145/2593861.2593862>

1. INTRODUCTION

Specifying Non Functional Requirements (NFRs) in adequate details and in an explicit manner is essential during the early phase of requirement engineering. NFRs are critical to describe the behavior and quality of the system and analysis of NFRs often play a crucial role in choice of architecture and design [1]. Unfortunately, NFR attributes are often overlooked and underspecified in early requirement documents and may cause serious impact in later stages of the Software Development Life-cycle (SDLC). Despite the inherent impreciseness, requirements are commonly written in natural language text for the purpose of easy communication and comprehension of the project stakeholders [2]. Past research suggests that any formal NFR analysis based on textual requirements will essentially have four steps. First, the NFRs (and attributes) need to be identified from the requirement text. Second, the NFRs have to be classified following a taxonomical hierarchy. Third, the NFRs have to be checked for completeness and coverage. Finally, we need to check if one defined NFR can potentially create conflict against another one.

There have been past works in each of these categories. In [3], authors discuss various frameworks for NFR representation and classification. In [4] authors suggest semi-supervised learning for NFR identification through a training set. There have been published works [5] [6] in tracing NFRs to functional requirements and thereby computing the extent of coverage (and absence) of “ilities” with respect to various functional requirements and flagging the un-addressed ones. The conflicting NFRs have been studied in [7], especially across security and usability requirements, and an analysis of the cross-cutting concerns in requirements is studied in [8]. However, most of these works either require a homogenized training set for learning or presence of an explicit ontology. Such methods are somewhat narrow in their approach, especially in the practical setting where domains vary widely and controlled experiments are not always feasible. It may be beneficial to utilize a more general semantic model of NFRs in IT domains and corresponding rule-based analysis.

In this paper, we put our emphasis on NFR identification from natural language text through textual pattern identification. Majority of the works so far [9, 3] have used simple keyword based identification techniques for NFRs. While keywords are good indicators for NFRs, they can also give rise to ambiguities and false classification. Consider the keyword “Password” which is an indicative word for the presence of Security NFR. Presence of the term “Password” in

the requirement sentence “The user password strength must be between 7 to 9 characters” clearly establishes that the requirement is about Security NFR. However, presence of term Password in the requirement sentence “Password management system should be implemented in module B” does not necessarily indicate any specific Security NFR. The hint of identification of a Security type NFR is strengthened by presence of other terms / patterns in the requirement sentence. For example, in the first sentence, the term “strength” (length) is an attribute of password and the phrase “7 to 9 characters” puts a constraint on that attribute. In the second sentence, no such specific property or attribute of Password is mentioned. Clearly, therefore, mere surface form identification of the NFR keywords is not enough. A detailed analysis of structure and sequence of lexical and semantic patterns is required to identify and classify NFRs.

In our work, we define and utilize a domain specific language (DSL) to specify and codify the syntactic (lexical) and semantic patterns - which then forms the vehicle of NFR detection. The approach for identification and classification is discussed in detail with a case study. This approach has also been tested against a publicly available NFR dataset [10]. Additionally, to aid the coverage and conflict analysis of NFRs, we propose creation of fine-grained computational models where conflicts could be analyzed mathematically. Completeness is addressed through evaluating whether the requirements cover atleast some aspects of the NFR categories and flagging out NFR categories that have been un-addressed. A finer grained analysis then addresses the under-specification in requirements with respect to individual categories.

The rest of the paper is as follows: In Section 2, we highlight the importance of NFR analysis on the architecture phase in the development lifecycle and we present a detailed description of our NFR analysis approach in Section 3. Section 4 presents a prototype implementation of this approach which has been evaluated against a publicly available requirement repository, and the results of the evaluation is presented in Section 5. Section 6 concludes the paper.

2. IMPORTANCE OF NFR ANALYSIS ON SYSTEM ARCHITECTURE PHASE

Early identification of NFRs is important as NFRs mandate specific evaluations and actions related to various aspects of the system being built. In particular, NFRs directly affect the technical design and architecture of the system. An architect must consider carefully the non-functional quality aspects (i.e., “ilities” like availability, scalability, usability etc.) to create suitable technical architecture of the system, so that system behavior and properties meet user expectations. Additionally, as proposed in [11], various architectural aspects like the presentation, data, integration and communication, development and operational aspects may also be signified by corresponding NFRs.

One important role NFRs play is in affecting the service level agreements (SLAs) to be signed-off with the clients/stakeholders of the system being built. Relatively small variations in non-functional requirements have significant bearing on the overall cost and architecture of the system. For example, it will typically be significantly more expensive to provide an average response time of 0.1 seconds as compared to 1 second for the same workload. Similarly, Fulfilling an NFR

of high availability would dictate the need of a design with failovers and redundancy upfront. NFRs may also affect the SDLC process. For example, certain security and regulatory compliance needs may mandate following certain certified processes and practices while developing the software. Lastly, as certain special skills may be involved in realizing elicitation and design for specific NFRs, the staffing of the project team and choice of the development environment may get influenced as well. Each of the above motivations are reasons enough for early and accurate identification and classification of NFRs in the requirements.

It is important to note that unlike the functional requirements, handling changes due to absent, imprecise, or conflicting non-functional requirements are significantly difficult. In many cases, varying functionality can be achieved by relatively localized changes to the data or control flow through the system. This, unfortunately, is not a case with many NFRs - a late discovery of stricter availability requirement may render the entire architecture unsuitable. More often than not, changing aspects like security, or performance of the system will at least need wide-spread changes throughout the system, and the complexity and cost of such rework later in the software development lifecycle may be significantly high compared to if these were identified earlier in the lifecycle. It has been shown in past that this increase in cost of rework grows exponentially, with the stages in software development lifecycle [12].

Technical architects therefore rely on NFRs to be properly identified early in the life cycle and need the NFRs to be well elicited, non-conflicting and relatively complete. Automated analysis of NFRs along these dimensions can therefore prove to be an important tool for requirement analysis as well as technical architecture phases of the system.

3. OUR NFR ANALYSIS APPROACH

Our NFR analysis is based on detecting a set of Linguistic “patterns” to identify occurrence of terms pertaining to non-functional attributes of a software system, in a natural language (English) text, and executing a set of “rules” on the identified pattern to denote presence of a category of NFR. We define and utilize a domain specific language (DSL) to specify and codify the patterns and rules into a rule engine which then forms the vehicle of NFR detection. In this section we present the key concepts and essential components of our work, such as the linguistic patterns, the rules, and their occurrences. We then present our algorithm for detecting these NFR patterns. We have implemented a prototype version of our NFR detection approach as well which we present later in the paper. This is followed by the approach we propose to pursue for completeness and conflict analysis of NFRs.

3.1 Patterns

Patterns are essential components of any detection approach - automated or manual. Typically, when a human evaluates a set of requirements for finding the non-functional ones, (s)he implicitly looks out for certain keywords, phrases, or linkages among various terms used in a statement. In this work, we define two broad types of patterns: Syntactic and Semantic.

3.1.1 Syntactic Patterns

Syntactic patterns are pre-specified set of singular language units or tokens (part-of-speech terms) such as nouns, verbs, adjectives, determinants, numeric values, literals and so forth, or co-located language units (called phrasal terms) such as a collection of consecutive nouns, adjectives and nouns, adverbial modifiers with adjectives and nouns, gerunds and so forth. Some examples of **Singular units** can be “Simultaneous”, “consecutive”, “concurrent” etc.; and some **Phrasal units** can be “round trip time”, “tolerance level”, “down-time”, etc. for example. The syntactic patterns can also be called as “keywords” or “indicator terms”.

3.1.2 Semantic Patterns

Semantic patterns are special set of language tokens that, in the context of the NFR analysis, would indicate or impose a special meaning. The members of semantic pattern sets can be present in the text. For example, the Semantic pattern <“timeunit”> will denote a set of language terms like “hour”, “minute”, “second”, “day”, “year”, “millisecond”, ... Each term in this set is called a member of the <“timeunit”> pattern. The member terms themselves or various abbreviation / variations of these terms (e.g. hr, min. sec ..etc) can be present in the text. Other examples can be <“calendar months”> or <“percentage”> etc.

3.2 Category and Category Hierarchy

We divide Non-Functional Requirements (NFRs) into categories under which the classification has to be done. For example, Figure 1 shows one slice of such hierarchy.

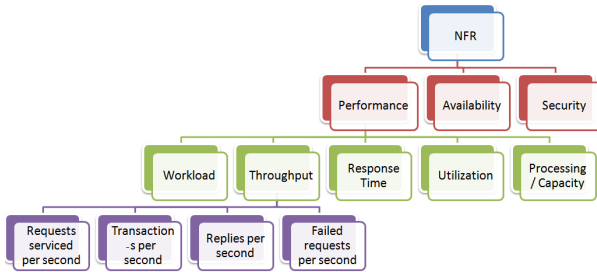


Figure 1: A sample NFR hierarchy

We may have keywords (Syntactic patterns) and semantic patterns defined at a commensurate level of category hierarchy. This is done to carry out more accurate and fine-grained analysis and classification of the text.

Note that the a repository stores such categories, sub-categories, and also the syntactic and semantic patterns and this repository can be manually / programmatically edited and appended. The contents may be generic and inclusive, and based on well known standards and state of practice, or may be specialized for only certain industry or non-functional domains, based on expert knowledge and industry knowledge-bases. An NLP parser like the *Link Grammar Parser* [13] can then be used to break down the input statements in its constituents and also provide the grammatical structure of the statement which is then used for the pattern matching. Note that for this analysis, we assume that the parser provides us information regarding any linguistic links that may exist among various parts of a natural language statement; i.e. we expect the parser to be a dependency

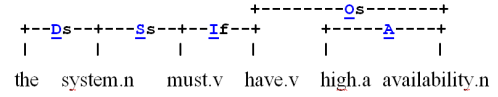


Figure 2: Parse of a sample statement showing linked tokens

parser.

3.3 Relationship among Patterns

We allow for utilizing deeper relationships (if exposed by the parse of the natural language) which might exist among the patterns based on English language parsing. Syntactic links provides a mechanism of knowing if two parts of speech or tokens arising of the text parse are grammatically connected (or dependent) on each other. For example, for the sentence - “The system must have high availability.”, a sample parse is shown in Figure 2. The subject <system> is linked to the verb <have> and that is, in turn, connected to object <availability>.

3.4 Detection Rules

A Rule defines how existence of a set of Syntactic and Semantic patterns in a sentence and relative positions of the patterns in the way they appear in the sentence, will validate the presence of an NFR and classify the sentence into a particular category / sub-category of the same. We divide the rules into following types -

1. R1 = “Existence of patterns” the Sentence s is of category c (from set C) if there exist at least one pattern p (from set P) in the sentence.
2. R2 = “Ordered patterns” the sentence s is of category c (from set C) where patterns $p_1, p_2, \dots, p_n$ (from set P) exist in manner where p_1 always precedes p_2 and so on.
3. R3 = “Sequential patterns” this is a special case of rule R2 where patterns $p_1, p_2, \dots, p_n$ have to be sequential.
4. R4 = “Ordered set of sequential patterns” this is special case of R2 and R3 where sets of sequential patterns $p_{i1} \dots p_{i4} \dots$ will have strict ordering.
5. R5 = “Linked patterns” this is a special case of R1, through R4 where there exist more than one pattern $p_1, \dots, p_n$ so that pattern p_i (i in $1, \dots, n$) is grammatically linked to p_j (j in 1 to n , $j \neq i$)

Note that there can possibly be more types of rules associated to various patterns including, but not limited to substring pattern (pattern p_1 is a sub-string of pattern p_2), indirect linkage (pattern p_1 is indirectly linked to pattern p_3 through pattern p_2) and so on.

3.5 The Domain Specific Language

One of the key component of our approach is the concept of expressing the rules in a format that allows easy editing and portability. For this purpose, we define a domain specific language (DSL) for NFR detection. In the DSL we define the following syntax -

1. Condition this is the “if” clause of the rule. Condition is defined through a Category definition i.e. *there is a category “{C}”* and a Sentence pointer i.e. *“there is a sentence in the file that contains sentence”*.

```

define "Workload a"
  When
    there is a category "Workload"
    there is a sentence in the file which contains patterns
    pattern-1 <"number">
    pattern-2 <"peak">
    pattern-3 "load"
    pattern-4 <"user">
    these pattern-2 pattern-3 are consecutive
  then
    Output category and return True
End

```

Figure 3: An example rule written in the proposed DSL

2. Syntactic Pattern types (key, value) (patternId, "pattern-String")
3. Semantic pattern types (key, value) (patternId, < "patternString" >)
4. Rules definition such as: E.g. (i) *"these pattern-X pattern-Y are consecutive"* (*these* and *consecutive* are DSL reserve words); E.g. (ii) *"these pattern-X pattern-Y are linked"* (*these* and *linked* are DSL reserve words).
5. Consequence such as "OUTPUT Z Category"

Figure 3 shows one example rule declaration using our DSL. This rule is for detecting NFRs which correspond to *Workload* (a sub category of Performance), and will, for example, be able to detect statements like "The system should have an average load of 5000 users". The creation of this DSL serves two purposes. First, it makes writing, editing and maintaining rules very easy. Second, it allows easy portability of the detection rules to any rules engine native syntax. For example, a way of porting the condition rule to Drools (Java based Rules engine) will be as follows

```
[condition] [] there is a category "{cat}" = category : Basic-Category( categoryName == "{cat}")
```

In the study we conducted for the proof of concept implementation of our approach, we have mapped these NFR detection rules to Drools Expert [14] (a sub project of JBoss Drools) an open-source rules engine.

3.6 Algorithm for NFR Detection

Let S be a set of requirement statements $s \in S$; C be a set of Categories / sub-categories $c \in C$; P be a set of patterns $p \in P$; R a set of rules $r \in R$. Here r is an instance of a rule defined in set R and r corresponds to a particular category c in C and is composed of a specific set of patterns p in P to be detected in the set S . We then prescribe an algorithm for NFR detection as shown in Algorithm A1.

The algorithm parses through all the requirement statements and executes all the rules to ascertain if any rule returns true for a statement. If so, the category corresponding to that rule is associated with the requirement statement and the rule(s) which fired true are also associated with it. The latter is particularly useful when a human reads the results as it helps validate why a statement was identified as an NFR and assigned a particular category.

3.7 Algorithm for NFR Completeness Analysis

We address completeness by evaluating whether the requirements cover atleast some aspects of all the NFR categories present. We flagging out NFR categories that could

Data: Sets S , C , R

Result: All classified non-functional requirements and detected rules.

initialization;

while an unprocessed $s \in S$ remains in set of requirement statements **do**

Parse s to extract tokens and links;

Execute all rules r in R ;

if any rule r corresponding to category $c \in C$ returns **True** **then**

Associate NFR category c to statement s ;

Associate NFR Rule r to statement s ;

Output s with associated c , r

else

Label and output the sentence as unclassified under

C ;

end

Goto next statement in S ;

end

Algorithm 1: A1: Algorithm for detecting and classifying NFRs

not be associated to any requirement statement. These signify that possibly additional requirements need to added to address such non-functional characteristics of the system. Note that many a times, certain NFR categories may not be pertinent to the system being designed, and in that case, we assume that the set C of the NFR categories is suitably updated to keep only those which are relevant to the analysis. The Algorithm A2 portrays the approach.

Data: Output of algorithm A1 (All $s \in S$, with associated c , r where $c \in C$, $r \in R$)

Result: All categories C' not detected in S

initialization;

Copy C to C' ;

while an unprocessed $s \in S$ remains in set of statements with associated category c and rule r **do**

if s is unclassified under C **then**

| / Do Nothing /

else

| $C' = C' - \{c\}$;

end

Goto next statement in S ;

end

Output set C' ;

Algorithm 2: A2: Algorithm for completeness analysis with regards to NFRs

Note that a finer grained analysis can further addresses the under-specification in requirements with respect to individual categories. For example, a response time requirement should also define or refer to a workload for which the response time is valid. However, here we do not perform this kind of analysis.

3.8 An Approach for NFR Conflict Analysis

The NFR category hierarchy described in the previous section may be used to populate a conflict matrix as presented in [15]. One of the possible approaches can be to use the output of Algorithm A1 and the conflict matrix to flag all pairs of requirements with conflicting categories. However the practicality of using this approach is limited as it may give rise to a huge number of (possibly) non-relevant results based on a broad generalization of possible conflict among the NFR categories.

We are currently developing a more elaborate approach to

detect conflicts at a deeper level. The approach is based on a multi-sentence modeling of the requirements, wherein the aim is to extract a model of the entities of the system based on their use in the requirement statements. This model, among other things, also documents the non-functional attributes associated with the entities and tries to populate known analytical (computational) models associated with the same. These models are relatively simple (but established) representations of non-functional attributes of a system and relate various non-functional aspects of the system with each other. For example, the inverse relationship between Downtime of a system and its percent Availability allows for a simple calculation or table (as at [16]). Now consider two requirement statements:

- s1: The system should have a downtime less than equal to 12 hours per year.
- s2: The system should have percent availability greater than 99.99%.

A cursory look at the table shown at [16], immediately tells us that the s1 and s2 are in conflict. This is because s1 allows for system downtimes upto 12 hours, thereby allowing a percent availability of less than 99.9% (which would correspond to a downtime of 8.76 hours), however s2 requires that the percent availability should be greater than 99.99% (or a downtime of less than 52.56 mins).

We are currently in the process of adding support for identifying such scenarios automatically. The approach identifies that an entity named system has an attribute called “downtime” and it is captured as “ ≤ 12 hours”. It also similarly associates an attribute “percent availability” with values “ > 99.99 ”. It then checks if there is any relationship (either analytical formula or a lookup table) associated with these and if so, does a validation check on the same. This work is still under progress and we are currently aiming to include only simple relationships like the one shown here.

4. AN NFR ANALYSIS PROTOTYPE

We implemented a prototype for NFR analysis based on the approach we have presented. This prototype has been built as a desktop application on Java JRE 1.6, with an aim to aid requirement analysis for NFRs. A schematic representation of this prototype is shown in Figure 4. We describe the key components of our implementation through this in the following subsections:

4.1 Parser

Each requirement sentence is processed by passing it through an open source dependency parser called link grammar (LG) [13]. This parser tokenizes each sentence into its constituent words, provides the POS tags for each word (noun, verb, determiner, number etc) and the linkage structures which connect the words.

4.2 Rule Engine

The analysis engine consists of a rule engine built using an open source Java rule engine called Drools Expert [14]. Drools is a Production Rule System (PRS) which matches Facts (or data) against a set of rules that must act on the data. If the facts meet the criteria defined by the rule, then a specific action (referred to as the consequence of the rule) must be taken. The Facts or data is represented as POJO

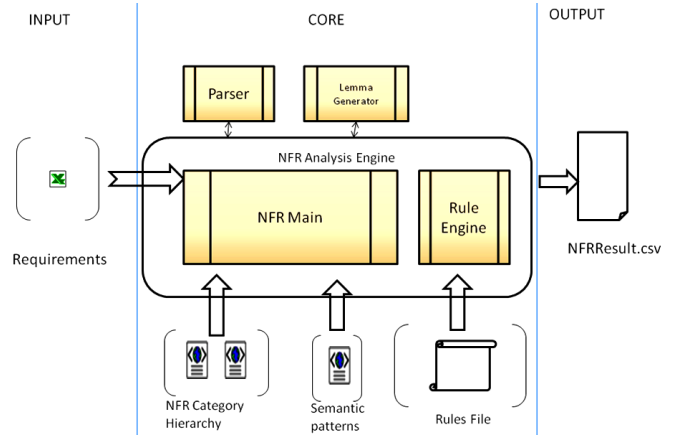


Figure 4: A schematic representation of the NFR detection prototype

(plain Java objects) and their attributes. The rules are written using a language native to the rule engine called DRL (Drools Rule Language) and defined in an external file. The rule engine executes by loading the facts into working memory and running the production rules on it. As detailed in the last section, the NFR classification rules are created using our own Domain (NFR) specific rule language or DSL. In order to allow the rule engine to execute these rules, a custom DSL mapping file is created which maps each valid statement in our DSL to one or more DRL language statements. The DSL mapping file is used by the rule engine to convert the rule file written in our DSL to the native language(DRL).

4.3 Lemma Generator

The lemmas of nouns and verbs are obtained by using Wordnet [17]. This is an open source lexical database for English. The identification of lemmas is necessary when matching semantic and syntactic patterns of the rule.

4.4 NFR Category hierarchy

This contains the list of categories of NFR and their sub-categories as described earlier. In our implementation, this is stored in XML format. The category definitions can be extended or updated by simply updating the corresponding XML files.

4.5 Semantic Patterns

The lists of semantic patterns that can be used as part of the rules are documented in XML format as well. In this prototype, a semantic pattern can be defined in at least 3 ways: a) By equating it to a list of words; b) By equating it to a regex expression syntax; c) By equating it to a part of speech (POS) tag. These are specified in corresponding xml files and the definitions can be extended easily.

4.6 Rules File

This contains the set of NFR rules. This is stored as a text file which can be edited and extended without rebuilding the entire source.

4.7 NFR Main

This is the component of the NFR Analysis engine which

Table 1: NFR Categories and Prototype Rules

S.No.	Category	Simple rules (only syntactic pattern type)	Complex rules (multiple pattern types & relationships)
1	Availability	32	14
2	Performance	65	20
3	Security	1450	4
4	Privacy	3	-
5	Usability	156	5
6	Compliance	32	3

orchestrates the processing steps. The broad steps executed by this engine are:

1. The requirement document is read using apache poi (open source Microsoft office library for Java applications).
2. Each sentence from the document is parsed using LG and the data obtained is loaded into the working memory of the rule engine.
3. The NFR category hierarchy and the semantic patterns are read from the external file and loaded into the working memory.
4. The rule engine reads the rules from the external file and fires (or executes) these rules on all the objects specified in the working memory. The lemma generator is used in the key comparison logic to ensure the matching between the facts and the syntactic/semantic patterns.
5. Finally the results of the analysis are saved in NFR-Results.csv file. This file contains all the requirements statements, along with their respective classification; i.e. to validate whether a requirement statement is an NFR, and if so, what category and subcategories does that map to, along with the rules which held true for the same.

5. AN EXPERIMENTAL CASE STUDY

We conducted an experimental study to gauge the capabilities of our NFR identification and classification approach. We took a set of requirement statements from the publicly available PROMISE [10] repository which has been used in past for such classification studies and allows for comparing automated classification with human categorization. We chose this repository for evaluating our results since it has been widely used for experimental evaluation of other NFR classification techniques [9], [3], [4].

Before delving deeper into the experimental study, let us qualify our NFR hierarchy further. Our NFR category hierarchy consists of the following high level categories: Availability, Performance, Security, Privacy, Usability and Compliance. Each of these categories contains a combination of simple (syntactic patterns) and complex rules (combination of syntactic patterns, semantic patterns and linked relationships). The simple rules are comparable with previous approaches [9]. The approx number of rules per category is listed in Table 1. As apparent from the Table 1 below, we have encoded a larger number of complex rules for categories like Performance and Availability and simple

rules for Usability and Security. Based on our interactions with requirements engineering practitioners and studying internal or publicly available non-functional requirements, we noticed that the Performance, Availability and Compliance requirements tend to follow a more repeatable kind of pattern structure with a limited vocabulary, whereas Usability and Security requirements tend to span a much larger vocabulary and follow no general structure.

We evaluated NFR Analysis prototype analyzing the requirement set from the PROMISE repository [10] for NFRs. We recorded the recall and precision measures across categories to ascertain the quality of detection and categorization. The PROMISE repository contains a set of 625 requirements manually classified into functional or non-functional requirements. The non-functional requirements are further classified into 11 categories:

- A=Availability
- L = Legal
- LF = Look and feel
- MN = Maintainability
- O = Operational
- PE = Performance
- SC = Scalability
- SE = Security
- US = Usability
- FT = Fault tolerance
- PO = Portability

We compared the results of our prototype implementation with the manual classification provided in the PROMISE repository [10]. The results indicate that the prototype performed fairly well. We show the evaluation results in Table 2. Note that, the NFR categorization defined in the repository defers to some extent from our category hierarchy. We attempt to map the PROMISE categorization to our categorization as follows:

- A (Availability) = Availability
- LF AND US (Look and Feel, Usability) = Usability
- PE (Performance) = Performance
- SE (Security) = Security, Privacy
- L (Legal) = Compliance

5.1 Experimental Considerations and Divergence

Given that there is no universal definition of NFR categories, we have observed that certain divergence exists between our interpretation of NFR categories and the manual categorization in the repository. This is mainly relevant to

Table 2: Evaluation Results

Category Name	Requirement Count in Repository	Requirement Count Identified by the Prototype	Requirement Count Correctly Identified	Precision	Recall
Availability (A)	21	22	18	0.8181	0.8571
Performance (PE)	54	48	33	0.6875	0.6111
Security and Privacy (SE)	66	68	43	0.6324	0.6515
Usability (US, LF)	105	87	77	0.8850	0.7333
Compliance (L)	13	21	9	0.4286	0.6923

the “Performance” and “Scalability” categories. Some sentences manually categorized under “Scalability” in the repository are considered to be “Performance” by us. This can be due to the overlapping nature of “Performance” and “Scalability”. For example, in the manually classified set, the statement: “The System shall allow for a minimum of 6 users to work at the same time.”, is classified as “Performance” (PE), however the following sentence is classified under “Scalability”: “The product shall be able support 1000 simultaneous users.”

As per our slightly altered classification, both these come under “Performance”. More specifically, the scalability requirements in the repository deal the following main topics:

1. Concurrency : Requirements that deal with the number of concurrent connections
2. Average Transactions or record processed in a given time period : Requirements that deal with the number of transactions or records processed in a day/year/month etc
3. Current user base : This handles the total user base and the growth expectations over the years
4. Database: The database scalability (or increase) requirements and integration with other database.

From these topics, 1) maps to “Performance— > Workload” and 2) maps to “Performance— > Transaction Mode and Batch” or “Performance— > Throughput”. Similarly, the definition of “Availability (A)” in the repository includes requirements dealing with the availability of 24-hour support. This consideration is not present in our current definition of these categories.

Another relevant difference is that while the manually categorized sentences in the repository are assigned only one classification per sentence, our approach can assign multiple categories to a sentence. This is because we have observed that in some sentences; more than one category may be relevant. Consider the sentence: “The system shall prevent malicious attacks including denial of service”. This has been classified as “Security” (SE) in PROMISE repository. However, this sentence can also be classified as “Availability” and hence we posit that the manual classification provided in PROMISE repository is not complete. Through our prototype tool, it will be categorized into both the categories and we believe that is the more pertinent way to go. This discrepancy in PROMISE repository however skews our results, leading to higher recall for “Security” and lower precision for “Availability”.

We also note that the “Legal” (L) category in the repository has a much broader definition than our “Compliance” categorization. In the context of industrial domain specification, “Compliance” is a sub-class of the Legal categorization.

Hence, our results will become better as and when our NFR hierarchy grows over time. To show the effect of subjectivity in categorization, if we go ahead and manually tag the requirements into “Performance” requirements based on our own definition and re-execute our analysis our precision changes as displayed in Table 3.

Table 3: Results for Performance (Altered Classification)

Category Name	# Requirements Identified by the Prototype	# Requirements Correctly Identified	Precision
Performance (PE)	48	44	0.9166

Apart from evaluation against the PROMISE repository, our prototype tool is also being used in some live customer projects to identify NFRs with good success.

5.2 Observations and Future Work

To our knowledge, this is the first integrated rule based approach for identifying and classifying natural language requirements for NFRs. Other classification techniques use keyword based [9] and machine learning approaches [9, 4, 18]. Our NFR analysis approach is able to achieve fairly high precision results without compromising on the recall values. This is in contrast to other approaches which exhibit similar recall values, while the precision remains relatively low. Also, the precision values for requirements like “Performance” (see Table 3), and “Availability” is achieved by a very small set of complex rules. We think that these complex rules are the key contribution of our approach. We would like to extend the prototype by creating rules for some other categories like “Scalability”, “Maintenance” etc., as well as discovering more rules for the existing categories. For “Security” and “Usability”, the same precision/recall value has been achieved using a larger set of simple rules. This implies that we can also extend our approach to use existing machine learning techniques to learn some of the syntactic patterns for these categories.

In addition, although other techniques have targeted a large variety of NFRs, only a few of them (like [6]) have a detailed ontology for NFR with subcategories for each category. We also aim to extend our category hierarchy to include other high-level NFR categories with their subcategories and create rules for each subcategory which could lead to a detailed ontology for NFR requirements.

6. CONCLUSION

Typically, software requirements are captured and documented in a natural language like English and at a broad level, these are supposed to capture the functional and non-functional aspects of the system to be designed. While functional requirements describe what a system is functionally supposed to do, Non-Functional Requirements (NFRs) describe the qualitative and quantitative measures of how a system should behave. Early identification of NFRs is important as these require specific actions, planning and evaluation with respect to various aspects of the system being built - all the way from what SLAs can be agreed to, how should the system architecture be, and what cost, effort and time building such a system will entail. Minor variations in NFRs can have a huge impact of all of these and thus analyzing software requirements for NFRs becomes critical.

In practice, the activity of identifying and analyzing of NFRs is mostly manual, effort intensive and subjective, and there is a dearth of tools to provide automation in this space. A few academic approaches exist, but these works largely focus on “bag of words” approaches with supervised and semi-supervised learning - which, in our view, may have limited practicality. In the real domain, it is better to model the NFRs in broad pattern based rules of syntactic and semantic types. Such patterns and rules must be generic enough to be applicable to different contexts, descriptive enough to be able to capture NFRs precisely (minimize wrong classification and ambiguities) and should be extendable and maintainable through external configuration files.

In this paper, we presented our work and discussed some of the initial results. The system is implemented in a modular and extendable way. We have found that our approach fares reasonably well against a common benchmark. We describe an algorithm for coverage analysis. We also briefly outline some of the on-going work in conflict analysis based on multi-sentence semantic analysis and related computational models. We mention some of the possible directions of our future work - especially around creating new rules and an ontology.

7. REFERENCES

- [1] B. Nuseibeh, “Weaving together requirements and architectures,” *Computer*, vol. 34, no. 3, pp. 115–119, Mar 2001.
- [2] C. Neill and P. Laplante, “Requirements engineering: the state of the practice,” *Software, IEEE*, vol. 20, no. 6, pp. 40–45, Nov 2003.
- [3] L. Chung and J. do Prado Leite, “On non-functional requirements in software engineering,” in *Conceptual Modeling: Foundations and Applications*, ser. Lecture Notes in Computer Science, A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, Eds. Springer Berlin / Heidelberg, 2009, vol. 5600, pp. 363–379. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02463-4_19
- [4] A. Casamayor, D. Godoy, and M. Campo, “Identification of non-functional requirements in textual specifications: A semi-supervised learning approach,” *Inf. Softw. Technol.*, vol. 52, no. 4, pp. 436–445, Apr. 2010. [Online]. Available: <http://dx.doi.org/10.1016/j.infsof.2009.10.010>
- [5] D. Mairiza, D. Zowghi, and N. Nurmuliani, “An investigation into the notion of non-functional requirements,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC ’10. New York, NY, USA: ACM, 2010, pp. 311–317. [Online]. Available: <http://doi.acm.org/10.1145/1774088.1774153>
- [6] T. H. Al Balushi, P. R. F. Sampaio, D. Dabhi, and P. Loucopoulos, “Elicito: A quality ontology-guided nfr elicitation tool,” in *Proceedings of the 13th International Working Conference on Requirements Engineering: Foundation for Software Quality*, ser. REFSQ’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 306–319. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1768904.1768927>
- [7] D. Mairiza and D. Zowghi, “An ontological framework to manage the relative conflicts between security and usability requirements,” in *Managing Requirements Knowledge (MARK), 2010 Third International Workshop on*, Sept 2010, pp. 1–6.
- [8] L. Rosenhainer, “Identifying Crosscutting Concerns in Requirements Specifications,” *First Monday*, 2004.
- [9] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc, “Automated classification of non-functional requirements,” *Requir. Eng.*, vol. 12, no. 2, pp. 103–120, May 2007. [Online]. Available: <http://dx.doi.org/10.1007/s00766-007-0045-1>
- [10] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012, June) The promise repository of empirical software engineering data. [Online]. Available: <http://code.google.com/p/promise-data/source/browse/trunk/text/nfr/nfr.arff>
- [11] G. Gokyer, S. Cetin, C. Sener, and M. Yondem, “Non-functional requirements to architectural concerns: Ml and nlp at crossroads,” in *Software Engineering Advances, 2008. ICSEA ’08. The Third International Conference on*, Oct 2008, pp. 400–406.
- [12] B. Boehm and V. R. Basili, “Software defect reduction top 10 list,” *Computer*, vol. 34, no. 1, pp. 135–137, Jan. 2001. [Online]. Available: <http://dx.doi.org/10.1109/2.962984>
- [13] D. Temperley, J. Lafferty, and D. Sleator. (2014) Link grammar parser. <http://www.abisource.com/projects/link-grammar/>.
- [14] J. community. (2014) Drools expert. <http://www.jboss.org/drools/drools-expert.html>.
- [15] V. Sadana and X. Liu, “Analysis of conflicts among non-functional requirements using integrated analysis of functional and non-functional requirements,” in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 1, July 2007, pp. 215–218.
- [16] Wikipedia. (2014) High availability. http://en.wikipedia.org/wiki/High_availability#Percentage_calculation.
- [17] P. University. (2014) Wordnet: A lexical database for english. <http://wordnet.princeton.edu/>.
- [18] W. Zhang, Y. Yang, Q. Wang, and F. Shu, “An empirical study on classification of non-functional requirements,” in *The Twenty-Third International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)*, 2011, pp. 190–195.