USING LINGUISTIC PATTERNS FOR IMPROVING REQUIREMENTS SPECIFICATION

Carlos Videira, David Ferreira, Alberto Rodrigues da Silva

INESC-ID, Rua Alves Redol, 9, 1000-029 Lisboa, Portugal

cvideira@inesc-id.pt, david.ferreira@inesc-id.pt, alberto.silva@inesc-id.pt

Abstract: The lack of quality results in the development of information systems is certainly a good reason to justify the presentation of new research proposals, especially those that address the most critical areas of that process, such as the requirements specification task. In this paper, we describe how linguistic patterns can be used to improve the quality of requirements specifications, using them as the basis for a new requirements specification language, called ProjectIT-RSL, and how a series of validation mechanisms can be applied to guarantee the consistency and correctness of the written requirements with the syntactic and semantic rules of the language.

1 INTRODUCTION

The requirements specification task is one of the most critical steps in the development of information systems. Not only because it encompasses the initial activities, whose results are critical for the success of the succeeding activities, and of the global project, but because it deals with the identification of the scope of the system to be developed, and the problem to be solved. Several surveys and studies (such as The Chaos Report, available at http://www.standishgroup.com) have emphasized the costs and quality problems that can result from mistakes in the early phases of system development, such as inadequate, inconsistent, incomplete, or ambiguous requirements (Bell, Thayer, 1976).

The requirements concept is one of those IS/IT concepts where there is no standard and widely accepted definition. A classical definition from Kotonya says that a "requirement is a statement about a system service or constraint" (Kotonya, Sommerville, 1998). The number of proposals, both research and practical, has grown in the last decade, but there is still not a universal or most accepted practice. The consequence is the use of different approaches for requirements specification, with different levels of formality; the most adopted solution is still the use of natural language to elaborate requirements specification documents.

This paper describes how the identification of the patterns most frequently used in requirements documents can be used to implement a series of techniques to improve the requirements validation process, using a number of parsing components. Section 2 presents an overview of the ProjectIT research program and describes the main features of a new requirements specification language, called ProjectIT-RSL. Section 3 describes the architecture and the parsing algorithms adopted, section 4 presents related work and section 5 overviews the paper and presents the future work.

2 PROJECTIT-RSL OVERVIEW

As a result of the experience gathered from previous research and practical projects, the Information Systems Group of INESC-ID (http://gsi.inesc-id.pt/), started an initiative, called ProjectIT (Silva, 2004), whose goal is to provide a complete software development workbench, with support for project management, requirements engineering, and analysis, design and code generation activities (the work presented in this paper is partially funded by the Portuguese Research and Technology Foundation, under project POSI/EIA/57642/2004 - Requirements engineering and model-based approaches in the ProjectIT research program). ProjectIT-Requirements (Videira, Silva, 2004b and

2004c) is the component of the ProjectIT architecture that deals with requirements issues. The main goal of this project is to develop a model for requirements specification, which, by raising their specification rigor, facilitates the reuse and integration with development environments driven by models. One of the results of this project is a new requirements specification language, called ProjectIT-RSL (Videira, Silva, 2004a and 2005).
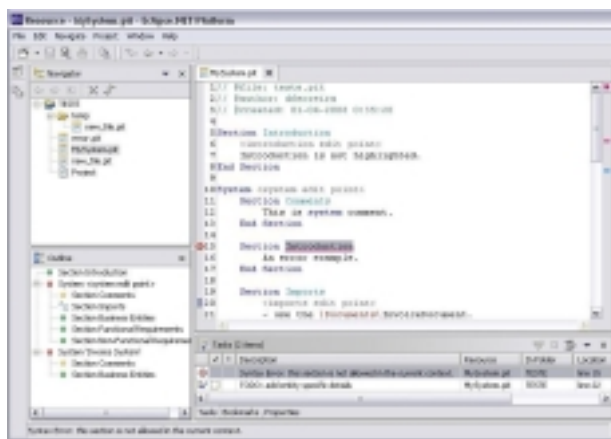

Figure 1: ProjectIT-RSL Editor

The definition of ProjectIT-RSL took in consideration the format and structure of the requirements documents of the projects we have developed, which led to the identification of a set of linguistic patterns associated with requirements. From these patterns we determined the main concepts used in requirements specification, how they are structured, organized, and combined into wider scope blocks. We derived a metamodel of the concepts identified, which is also the base of a profile (called XIS - Silva, Lemos, Matias, Costa, 2003), common to all our tools. Based on the patterns identified, we defined the syntax of ProjectIT-RSL, which was tested in a prototype developed using the features provided by Visual Studio .NET and the .NET Framework (Carmo, Videira, Silva, 2005), and is now being supported by an integrated set of tools, called ProjectIT-Studio. An example of the editor of ProjectIT-RSL is shown in Figure 1.

The complete specification of all ProjectIT-RSL rules is beyond the scope of this paper, and can is presented in more detail in (Videira, Ferreira, Silva 2006). ProjectIT-RSL allows the definition of different "application units": (1) reusable components, which can be specified independently, and integrated in broader systems; (2) complete executable systems, that can "include" some of the previous ones (reusing their functionality); (3) architectural templates and application templates, which allow pattern reuse and instance reuse, respectively. The rules expressed below in EBNF notation abstract the structure of our requirements document.

```
<Requirements Document>: [<Introduction Section>]
<Application Unit>*
<Application Unit>: <Section>*
<Section>: <Sentence>*
```

The sentences of our requirements documents are divided in two groups, declaration and definition sentences; the first ones just give names to concepts, associating them with a specific type (which is what happens in an Operation Declaration) whereas definition sentences detail the features of a concept (such as an Entity Definition).

```
<Sentence>: <Declaration> | <Definition>
<Declaration>: <Application Unit Declaration> |
<Operation Declaration>
```

Our profile identified three base concepts, Entities, Actors and Operations, defined by the following rules, which basically state that the complete specification of a concept can be done by a number of sentences.

```
<Operation Specification>: <Operation Declaration>
<Operation Definition>*
<Actor Specification>: <Actor Definition>*
<Entity Specification>: < Entity Definition>*
```

For example, the number of rules currently used for validating an entity specification is already very large, as the following EBNF rules, although not complete, show.

```
<Entity    Definition>:    <Entity    Inheritance
Definition> | <Entity Property Definition> |
<Entity   Equivalence   Definition>   |   <Entity
Association Definition>
<Entity Inheritance Definition>: <Entity> is a
<Entity>
<Entity Equivalence Definition>: <Entity> is the
same as <Entity>
<Entity   Property   Definition>:   <Entity>   has
<Property Definition>*
<Property   Definition>:   [a|an|the]   [<Primitive
Type>] [<Quantifier>] <Property>
<Quantifier>: <number> | at least <number> | at
most <number> | a list of | each | …
<Property>: Name | <Entity>
<Entity Association Definition>: <Entity Active
Association   Definition>   |   <Entity   Passive
Association Definition>
<Entity   Active   Association   Definition>:
[<Quantifier>]   <Entity>   <Active   Verb>
[<Quantifier>] <Entity>
<Entity   Passive   Association   Definition>:
[<Quantifier>]   <Entity>   <Passive   Verb>
[<Quantifier>] <Entity>
```

Although we want to allow the users of our tools to use natural language, the parsing mechanisms, as well as the integration with code generation tools, imply that we must restrict the terms allowed to a recognizable subset, such as the fixed terms we have seen in the above rules. This set of rules, called the TS rules (Template Substitution rules), which can be defined and changed for a specific project, enables the incremental evolution of these terms, just by adding more rules, or by defining synonyms between words. This approach not only supports different writing styles and natural languages, but also is the base for the definition of domain specific languages. The rules are stored in groups related to the sections they apply, and as such we have specialized business entities, functional requirements and non-functional requirements rules.

3 PROJECT COMPONENTS

As figure 2 shows, the architecture that supports ProjectIT-RSL is composed by a number of different components, from which we must emphasize the roles of three of them: a text editor, two specialized parsers and an inference engine.
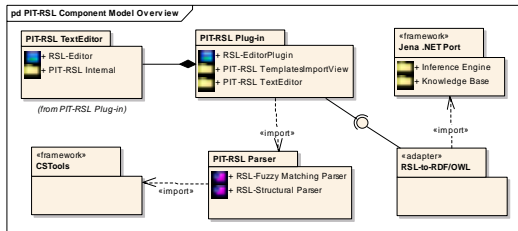

Figure 2: ProjectIT-RSL components

3.1 The Text Editor

The text editor, represented by the package PIT-RSL Plug-in, is a plug-in built upon the capabilities of Eclipse.NET (a port we have performed of Eclipse to the .NET platform), with features such as auto-complete, auto-format, warnings and errors annotations (text underline and vertical bars marks), syntax-highlighting and suggestions.

When the user opens a requirements specification document written in ProjectIT-RSL (a .pit file) with the text editor, it performs an initial full parsing of the document's contents and starts a read-evaluate-print cycle typical of event oriented interfaces such as the one we are using. Upon detection of a document manipulation, the plug-in sets a timeout mechanism that triggers an event after a configurable short time interval and, which revaluates the whole document again, applying only the parsing algorithms to new or modified requirements. Therefore, this lazy document's evaluation mechanism avoids repetitive calls to the parsing mechanism while the user is temporarily typing. Consequently, having in mind the Model-View-Controller (MVC) architecture's analogy, all model dependent views (subscribers) of the parsing result (the RSL model) stay immutable until new relevant changes occur in the document's contents presented in the text editor (a view-controller component), instead of being constantly refreshed.

3.2 The Parsing Components

The analysis of the requirements sentences is performed by two parsers, the RSL-Structural Parser and the RSL Fuzzy Matching Parser. The first is generated by a set of tools called CSTools (available at http://cis.paisley.ac.uk/crow-ci0/) and performs the initial parsing steps, validating the document's structure. The second is called PIT-RSL Fuzzy Matching Parser and is responsible for processing Natural Language (NL) text to find the optimal parsing tree, by successive testing NL patterns contained in the TS set of rules.

3.3 The Structural Parser

The generation of the Structural Parser is based upon two script files, one that contains all the regular expressions that recognize the tokens specified in the PIT-RSL language definition, and the other that contains all non-terminal and abstract syntax tree nodes specifications.

One of the first steps of the Structural Parser is to break in tokens its input (the requirements document): it parses the file accordingly with the semantic contexts introduced by the SYSTEM and SECTION tokens. This enables the detection and validation of the requirements hierarchical numeration and enforces a predefined sentences' structure for each of the above scopes. This step was essential for early detection of potential problems and inconsistencies.

For dealing with the nested structural scopes of the requirements specification document, we introduced a context stack mechanism. Additionally, it was necessary to establish an error and warning

mechanism to allow the parsing process to continue to run until the end of the requirements document file, even in the presence of non-critical errors, introducing this way a certain level of robustness.

The output of the early stage transformations is an abstract syntax tree (AST) which contains an overall improved representation of the original free-form document, where each requirement is contained in a specific section of the nested structural hierarchy and consists of a sequence of words. Subsequent iterations over the produced AST are used to supply information to other internal data structures belonging to other components which are responsible for the next parsing stages. Before starting to examine the requirements' semantics, the parsing mechanism must first import all the referenced documents/systems present in the section type "Section Imports". The import mechanism follows a depth-first approach while parsing and loading the documents/systems specified for import but, for safety, it maintains a path trace which avoids endless importing cycles and redundancies.

3.4 The Fuzzy Matching Parser

The second parsing component is called the Fuzzy Matching Parser (FMP), which heuristically analyses the semantics of each requirement through the adherence of the statement's semantics to its syntactic structure, which is typical of requirements sentences. Initially, this component performs a morphological and syntactic analysis of each word of every requirement statement. This categorization analysis, based on each word definition and on the context in the statement (relationship with adjacent and related words in a sentence or paragraph), is performed by an external component that implements the Brill Tagger's algorithm for part-of-speech tagging (more information available at http://www.cs.jhu.edu/~brill/code.html), which marks each word by appending a set of grammatical information tags, providing the required words' classification for further appliance of the FMP algorithm.

To avoid redundant "labelling" and parsing information, we store a list of all distinct words used in the requirements document, and implement a hashing mechanism that assures words' uniqueness when adding new words to the list. The use of the word list constitutes an efficient way of managing the terms available and, simultaneously, improves memory usage since it is based on the GIF format compression algorithm (Sayood, 1996), where each

sentence is converted in a sequence of numbers, each corresponding to a previously tagged word.

For each new requirement addition, the FMP algorithm is called with the requirement statement and a set of specific TS rules, depending on the section where the requirement belongs, providing a more refined parsing. The algorithm attempts to find the optimal parsing tree by recursively trying, until exhaustion, to match the requirements statement information with the TS rule's templates part and, upon a successive match, substitutes the matched information with the TS rule's substitution part. This recursive search is guided by a heuristic function, which gives a score value for each TS rule applied. In each step the algorithm iterates over all TS rules and, for each, tries to find if there is a complete match between it and the statement being parsed.

A complete match occurs when for all elements of TS rule template part there is at least one match between that template element and a word of the requirement's sentence. If a complete match occurs, then for all valid matches, the algorithm applies a substitution operation which replaces the matched text fragment of the current requirement's statement by the template part of the TS rule under analysis within the algorithm step. During the replacement process, the template's variables are bound to the corresponding word values of the statement and this new requirement's statement is further used in the next recursive step. Then, the algorithm determines an overall score for the step with a parameterized heuristic function which is based in the following three aspects: (1) the "match quality", which includes the relative word positions, eventual words inversions, and number of words discarded; (2) the TS rule template length, number of variables, and number of constants; finally (3) the length of the requirement statement's fragment that couldn't be parsed. At the end of the step, the algorithm calls itself in a recursive manner executing the same behaviour until it reaches a terminal case where no more rules can be applied.

At the end, the algorithm verifies if the achieved results demonstrate a minimal level of parsing quality by comparing its score with a minimum threshold value: if the attained score exceeds the threshold, the algorithm returns the best results found – the optimal parsing tree; otherwise it returns the original requirement sentence. Since this algorithm is intrinsically recursive, we have to guarantee that it neither enters in an endless loop, nor repetitively reapplies the same TS rule, which in both cases mean a possible TS rules specification error, and consequently a PIT-RSL linguistic

patterns' constructs problem. To solve this issue we have introduced a Background Thread Variable Timeout Mechanism (BTVTM). Its goal is to run this heavy-weight FMP algorithm FMP in the background. It has an associated timer for only letting the algorithm perform its tasks during a previously specified period, thus assuring that the algorithm eventually ends. This strategy also guarantees that the tool always provides feedback to the end user instead of blocking each time the algorithm runs during the parsing process.

3.5 Inference Engine

Finally, and to allow further knowledge inference capabilities, important for requirements validation, PIT-Studio/RSL uses two other components: RSL-to-RDF/OWL and Jena .NET Port. The former contains the adapter pattern code that provides a clean C# API for using the .NET ported Jena framework without the necessary traces of java syntax code. The Jena .NET Port package represents a .NET port of Jena framework (available at http://jena.sourceforge.net/), which supplies the PIT-RSL plug-in with knowledge-base and inference-engine capabilities

4 RELATED WORK

The use of natural language in the initial phases of the software development process has received attention for more than 20 years. Abbot (Abbot, 1983) proposed that nouns could be used to identify classes, adjectives to identify attributes, and verbs to identify methods. OICSI is a tool developed by Rolland and Proix (Rolland, Proix, 1992), to help the identification of requirements from natural language text and available domain knowledge. Attempto Controlled English (ACE), first described in (Fuchs, Schwitter, 1996), is one of those approaches that use a controlled natural language to write precise specifications that, for example, enable their translation into a first-order logic similar representation (called DRS).

The use of parsing techniques to elaborate a conceptual model from natural language requirements is a common approach; in (Macias, Pulman, 1993) we can find descriptions of proposals to use a controlled natural language with a limited syntax in order to specify requirements with more quality. Some of the previous initiatives were concerned with detecting problems in previously

written requirements documents (Fantechi, Gnesi, Lami, Maccari, 2002), while others are concerned with the elaboration of requirements documents without such problems (Ben Achour, 1998, Denger, 2002). NL-OOPS (Mich, Garigliano, 1999) and LIDA (Overmyer, Lavoie, Rambow, 2001) are systems that process natural language requirements to construct the corresponding object-oriented model. A similar system is described in (Nanduri, Rugaber, 1996). Although the number of initiatives seems to justify the potential of natural language requirements, there are studies reporting problems in using natural language requirements specifications (Berry, Kamsties, 2003).

A number of different approaches have researched on the elaboration of requirements specification using patterns of natural language. Approaches such as (Ben Achour, 1998) and (Rolland, Proix, 1992) reduce the level of imprecision in requirements by using a limited number of sentence patterns to specify a requirement for a particular domain. Denger (Denger, 2002) has also identified natural language patterns used to specify functional requirements of embedded systems, from which they developed a requirements statements metamodel. Juristo and Moreno try to formalize the analysis of natural language sentences in order to create precise conceptual model (Juristo, Morant, Moreno, 1999).

Ambriola and Gervasi proposed the CIRCE project (sometimes defined as a "lightweight formal method") (Ambriola, Gervasi, 2003), which uses natural language as the specification language, and is also supported by fuzzy matching parsing techniques to extract knowledge from requirements documents and produce a formal validation of requirements. Although Circe and ProjectIT-RSL have some similarities, there are between them many differences, namely in the architecture, concepts and algorithms used, and above all, in the strategy: the goal of CIRCE has initially been requirements validation, and only recently integrated with model driven approaches, whereas our goal with requirements specification is to obtain a consistent requirements document that enables the use of model driven techniques and code generation.

5 CONCLUSIONS AND FUTURE WORK

The importance of requirements specification led us to propose a new specification language, closely

supported by a number of tools that cover most of the requirements specification process, mainly the specification and validation steps. This paper focused on the description of the parsing steps algorithms, following others where we have described the language ProjectIT-RSL in more detail (Videira, Ferreira, Silva, 2006). The language and the tools have already reached an important maturity level, and the application in small examples has led us to conclude that, although sharing points with other initiatives, we think that our approach has a unique combination of ideas that has not been tried.

In the near future we will concentrate in the development of the requirements reuse mechanisms and in advancing tool support. For example, we will automate the generation of the TS Rules from the ProjectIT-RSL abstract rules, and we will develop plug-ins to show, in different formats, the information stored in the knowledge base. When our ProjectIT-RSL and its supporting tools reach a sufficient maturity level, it is our intention to use them in real projects, to better test and proof the ideas we are proposing.

REFERENCES

Abbot, R., Program design by informal english description, Communications of the ACM, 16(11), pp. 882-894, 1983

Ambriola, V., Gervasi, V., The Circe approach to the systematic analysis of NL requirements, Technical Report TR-03-05, University of Pisa, Informatics' Department, 2003.

Bell, T., Thayer, T., Software requirements: Are they really a problem?, Proceedings of the 2nd Int. Conference on Software Engineering, pp. 61-68, 1976

Ben Achour, C., Guiding Scenario Authoring, Proceedings of the 8th European-Japanese Conference on Information Modeling and Knowledge Bases, pp. 152–171, IOS Press, Vamala, Finland, May 1998

Berry, D., Kamsties, E., Ambiguity in Requirements Specification, Perspectives on Software Requirements, eds. J. C. Sampaio do Prado Leite and J. H. Doorn, Kluwer Academic, pp. 191-194, 2003

Carmo, J., Videira, C, Silva, A., Using Visual Studio Extensibility Mechanisms for Requirements Specification, 1st Conference on Innovative Views on .NET Technologies, Porto, June 2005

Denger, C., High Quality Requirements Specifications for Embedded Systems through Authoring Rules and Language Patterns, M.Sc. Thesis, Fachbereich Informatik, Universität Kaiserslautern, 2002

Fantechi, A., Gnesi, G., Lami, G., and Maccari, A., Application of Linguistic Techniques for Use Case Analysis, Proceedings of the IEEE Joint International Requirements Engineering Conference (RE'02), IEEE Computer Society Press, Essen, Germany., 2002

Fuchs, N., Schwitter, R., Attempto Controlled English (ACE), CLAW 96, First International Workshop on Controlled Language Applications, University of Leuven, Belgium, March 1996

Juristo, N., Morant, J., Moreno, A., A formal approach for generating oo specifications from natural language, The Journal of Systems and Software, Vol. 48, pp. 139-153, 1999

Kotonya, G., Sommerville, I., Requirements Engineering Processes and Techniques, New York. Jonh Wiley & Sons, 1998

Macias B, Pulman S., Natural language processing for requirements specification, Safety-critical Systems, pp 57–89, Chapman and Hall: London, 1993

Mich, L., Garigliano, R., The NL-OOPS Project: OO Modeling using the NLPS LOLITA, Proc. of the 4th Int. Conf. Applications of Natural Language to Information Systems, pp. 215-218, 1999

Nanduri, S., Rugaber, S., Requirements Validation via Automated Natural Language Parsing, Journal of Management Information Systems, 1996

Overmyer, S., Lavoie, B., Rambow, O., Conceptual Modeling through Linguistic Analysis using LIDA, Proc. of the 23rd Int. Conf. Software Engineering, pp. 401-410, 2001

Rolland, C., Proix, C., A Natural Language Approach for Requirements Engineering, Proceedings of the 4th Int. Conf. Advanced Information Systems, CAiSE 1992

Sayood, K, Introduction to Data Compression. Morgan Kaufmann, 1996

Silva, A., O Programa de Investigação "ProjectIT", Technical report, V 1.0, October 2004, INESC-ID

Videira, C., Silva, A., The ProjectIT-RSL Language Overview, UML Modeling Languages and Applications: UML 2004 Satellite Activities, Lisbon, Portugal, October 2004a

Videira, C., Silva, A., ProjectIT-Requirements, a Formal and User-oriented Approach to Requirements Specification, Actas de las IV Jornadas Iberoamericanas en Ingeniería del Software e Ingeniería del Conocimiento - Volumen I - pp 175-190, Madrid, Spain, November 2004b

Videira, C., Silva, A., A broad vision of ProjectIT-Requirements, a new approach for Requirements Engineering, in Actas da 5ª Conferência da Associação Portuguesa de Sistemas de Informação, Lisbon, Portugal, November 2004c

Videira, C., Silva, A., Patterns and metamodel for a natural-language-based requirements specification language, CaiSE 2005 Forum, Porto, June 2005

Videira, C., Silva, A., A linguistic patterns approach for requirements specification language, Euromicro SEAA 2006 Conference, Dubrovnik, August 2006