An Approach for Automating Use Case Refactoring

Alejandro Rago[1,2], Paula Frade[2], Miguel Ruival[2], Claudia Marcos[1,2]

[1] Instituto Superior de Ingeniería de Software Tandil (ISISTAN), CONICET
Campus Universitario, Paraje Arroyo Seco, B7001BBO, Tandil, Bs. As., Argentina

[2] Facultad de Ciencias Exactas (ISISTAN), UNICEN University
Campus Universitario, Paraje Arroyo Seco, Tandil, Bs. As., Argentina

{arago,cmarcos}@exa.unicen.edu.ar - {maripau07,miguebt}@gmail.com

Abstract. Carrying out requirements capture and modeling activities successfully is not easy, often requiring a thoughtful analysis of clients' needs and demanding an adequate expertise from analysts. To ensure a fluid communication among stakeholders, analysts must take advantage of modeling techniques while describing requirements and exploit reuse and abstraction practices so as to avoid redundancy (for instance, using relations between use cases). Unfortunately, these practices are seldom applied because inspecting requirements such as textual use cases by hand, looking out for faulty or duplicate functionalities, is a challenging and error-prone activity. In this context, we introduce an assistive approach called *ReUse* that searches redundancy deficiencies in use case specifications and allows to fix them with relation-based refactorings. Our approach makes use of text processing and sequence alignment techniques to discover deficiencies (e.g., duplicate functionality). We have evaluated *ReUse* in five case studies, achieving promising results.

Keywords: use case refactoring, sequence alignment, requirement defect, domain classification, requirements engineering

1 Introduction

Successful software projects commonly attain a clear understanding of clients' needs. Elaborating a "good" requirements specification is a key aspect to achieve this goal, which permits to describe and thus communicate functionalities of a system effectively through the software development process [9]. Despite the experience the industry has gained in the last decades regarding Requirements Engineering (RE), software companies still have trouble for eliciting, documenting and managing requirements. Many projects fail to deliver working software within time and budget due to poor quality requirements [11]. In general, requirements are specified with textual descriptions written in natural language. Use cases, for instance, are a widespread and powerful technique to document requirements. Use cases are simple to communicate [10] and are supported by

guidelines of good practices [5]. Yet, in practice these use cases exhibit a number of deficiencies, such as: duplicate functionality [4], lack of abstraction [13], tangled and scattered descriptions [3].

Deficient specifications, apart from being complex to analyze and communicate, entail many problems throughout the software development process. Recently, several researchers have focused on the identification of deficiencies in textual specifications and the development of mechanisms to solve them [14,19,13]. These mending mechanisms are usually arranged in refactoring catalogues for requirements. However, applying a catalogue to an existing specification is a responsibility of the analyst, who must inspect requirements by hand in order to find potential deficiencies that can (and should) be improved. In this context, we propose an approach called *ReUse* that helps analysts to improve use case specifications. *ReUse* performs an iterative analysis of textual specifications to identify potential deficiencies and explores refactoring catalogues to find them a proper solution for those deficiencies. The approach is implemented in a prototype tool that simplifies the interaction of analysts during the improvement of the requirements. The contributions of this work are two-fold. First, we defined an iterative approach that progressively improves requirements specifications in an automated fashion and guides the analysts for searching deficiencies and selecting/executing refactorings. Second, we developed a component to discover duplicate functionality by combining natural language processing and machine learning with sequence alignment techniques (generally used in bioinformatics for genetic research). In order to evaluate the performance of our prototype, we have performed a series of experiments in five publicly-available case studies. The results obtained so far are encouraging, correctly detecting most of the deficiencies and suggesting the right refactorings for fixing them.

The rest of this work is organized as follows. Section 2 reviews related works that have addressed requirements deficiencies and refactorings. Section 3 introduces our tool approach and describes the organization of its components. Section 4 reports the results obtained in the experiments. Finally, Section 5 gives the conclusions and analyzes future lines of work.

2 Related Work

In the last decade, there has been a growing interest in strategies for improving the "quality" of requirement specifications. By quality we mean that requirements must be clear, unambiguous and essential (i.e., non-redundant). Specifically, several authors have developed approaches and tools to achieve this goal. The general schema of requirements improvement can be organized into three parts: (i) the evaluation (or analysis) of textual specifications, (ii) the identification of deficiencies and (iii) the application of refactorings.

On one hand, only a few works have explored the search for deficiencies to guide and focus the general improvement process of requirement documents. The most comprehensive study is that of Ciemniewska et al., who developed techniques to identify defects in use case specifications [4]. Defects are classified in

three levels: *specifications*, *use cases* and *steps*. The *specifications level* comprises behavioral duplication. The *use cases level* comprises use cases that are too short or too long, complex extensions, among other defects. The *steps level* comprises intricate syntactical expressions, actor omissions, among other problems. Despite this work does not propose refactorings for the defects, it is relevant because it defines simple heuristics to automatically detect defects in the text.

On the other hand, other works have addressed the definition of refactoring catalogues. Some of those works formalize the requirements analysis process by means of the Goal Question Metrics (GQM) method [17]. GQM prescribes the creation of an evaluation and improvement plan, defining a measurement models to assess the quality of software products. For instance, Ramos et al. instantiated the GQM framework for use cases [13]. Their approach, called AIRDoc, allows to iteratively improve the quality of use cases via evaluations and refactorings. The approach describes a series of steps that can solve each of the problems. However, the identification of defects has to be manually made by an analyst.

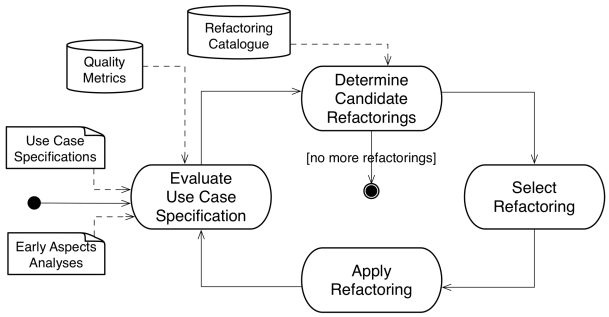
With similar goals, some works have focused on fixing deficiencies of the requirements in textual documents. Rui et al. studied the application of refactorings to use cases [14]. Their refactorings are described using a meta-model of use cases. The model comprises concepts at different levels, such as use cases, textual scenarios and actors, among others. However, the task of determining "where" and "how" to apply such refactorings in the text is uncertain, and this responsibility is left to the analysts' criteria. In the same line, Yu el al. suggested that to ensure the construction of use cases with a correct granularity and proper abstraction level, we have to look for "shared" functionality, avoiding duplicate behavior and promoting reuse relations [19]. To this end, their work introduces a catalogue of refactorings to solve duplicate functionality in software specifications. The catalogue makes use of the concept of episodes to operationalize refactorings. An episode is defined recursively as: a complex behavior that is composed of more simpler scenarios, or an atomic unit of behavior (i.e., an interaction between an actor and the system). To execute a refactoring, an analyst has to align the episode model to a particular specification and then make the transformation required. Similarly to [14], identifying the location of the defects is out of the scope of Yu's research.

3 *ReUse*: an Approach for Refactoring Use Cases

To ease the analysts' efforts to inspect requirements documentation and ensure the specifications comply with quality criteria, we propose an iterative approach that allows to automate the refactoring of use cases. This approach, called *ReUse* (Refactoring Assistant for Use Cases), can support the search for deficiencies in use case specifications in an automated fashion and then assist analysts to mend the problems found. The *ReUse* approach receives as input a set of textual use case specifications and the analyses of *early aspects* (optionally made with a third-party tool). In addition to those inputs, the approach also makes use of a series of artifacts specially developed for *ReUse*. These artifacts are the *quality*

measures for use cases and a *refactoring catalog*. A measure defines the properties that expose a particular kind of deficiency. The refactoring catalogue contains solutions to pre-existing deficiencies in use cases.

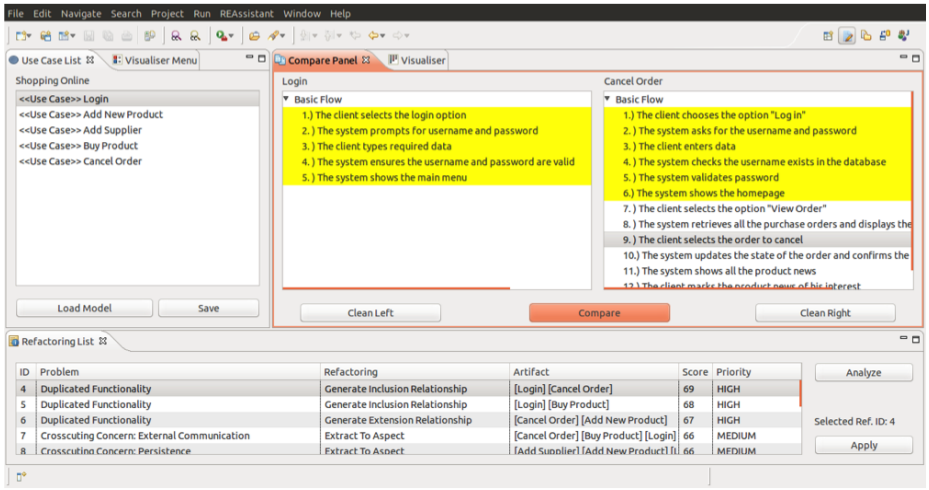
Fig. 1. Approach for Refactoring Use Cases



The *ReUse* approach is organized into four main activities that altogether contribute to the improvement of use cases (see Figure 1). The activity Evaluate Use Case Specification has the objective of finding deficiencies in textual requirements using quality measures. As output, this activity generates a set of potential problems, viz. behavioral duplication, lack of abstraction or improper relations between use cases. The activity Determine Candidate Refactorings has the objective of deducing which refactorings are fit for solving the problems found earlier. The activity Select Refactoring has the objective of establishing which of the suggested refactorings is more beneficial in terms of requirements quality. With this purpose, refactorings are prioritized using several factors, producing a "ranking" of refactorings. The ranking is presented to the analysts, who will then choose either the refactoring with highest priority or another one according to their personal criteria and expertise. Finally, the activity Apply Refactoring takes the selected refactoring and executes it on the textual specification. This means to perform a series of transformations on the text to systematically mend the deficiency and solve the problem. When a particular refactoring requires additional information (e.g., the name of a newly created use case), the analyst is prompted on demand. The output of this last activity is a partially-improved use case specification, which constitutes the input of the next iteration of the approach.

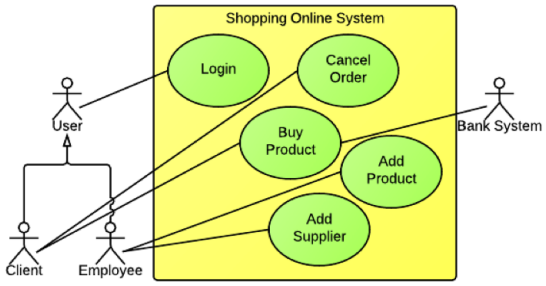
ReUse is implemented as a number of Eclipse plugins, which allow analysts to intervene in each activity of the iterative process of the approach. Figure 2 shows how the prototype tool displays several deficiencies found in a system. To better understand the approach, the activities of the approach will be further explained with an example. Let us consider the following subset of use cases from an Online Shopping System (Figure 3). The specification of this system consists

Fig. 2. User Interface of the Prototype Tool



of five use cases. The Login use case allows each user, either employee or client, to identify himself/herself in the system The *Add Product* and *Add Supplier* use cases allow company employees to register new products and suppliers to the system. Finally, the *Buy Product* and *Cancel Order* use cases allow customers to purchase items and cancel a previous purchase that has not been sent yet. In the sequel, we describe each activity of the *ReUse* approach.

Fig. 3. Use Cases of the Online Shopping System



3.1 Evaluate Use Case Specification

This activity collects information from textual specifications that allows to recognize existing deficiencies. The results of analyzing the use cases are a set of metrics regarding particular quality goals. For instance, after inspecting the *Login* and *Cancel Order* use cases, *ReUse* will find evidences of duplicate functionality deficiencies, since both use cases describe the steps required to enter into the system.

Table 1. Questions and Metrics associated with each Quality Objective

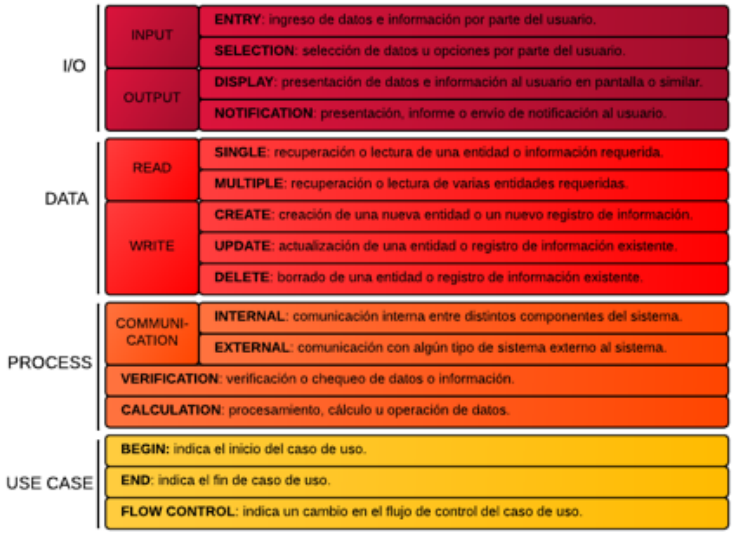
Quality Goal	Question	Metric
Reuse / Modifiability	Q1: Are there duplicate portions of functionality? Are there sections in the specifications that, although written in different terms, describe the very same behavior?	M.1.1: Duplicated functionality blocks
	Q2: Are non-functional requirements encapsulated and well modularized?	M.2.1: Non functional requirements not modularized
	Q3: Are functional requirements encapsulated and well modularized?	M.3.1: Functional requirements not modularized
Understandability / Maintainability	Q4: Are there elements in the model that lack of value and meaning? Are there use cases that do not comply with UML semantics?	M.4.1: Wrongly defined or meaningless actors M.4.2: Wrongly defined or meaningless use cases M.4.3: Inadequate relations between use cases
	Q5: Are requirements specifications simple but yet representative of user's needs?	M.5.1: Overly large use cases M.5.2: Overly short use cases M.5.3: "Happy" use cases (without alternative flows)

We chose to instantiate the schema of objectives, questions and metrics proposed by the GQM (Goal-Question-Metric) method [17] for searching deficiencies in requirements specifications. GQM defines an ordered number of activities for inspecting software artifacts such as textual use cases, or architectural models, among others. First, we determined a set of quality goals and accompanying questions to respond if those goals are fulfilled. We considered two quality goals: (i) *Reuse / Modifiability*, that ascertains the level of reuse and support for change of requirements documents by observing the encapsulation and duplication of information, and (ii) *Understandability / Maintainability*, that ascertains the level of clarity and readability of requirements by observing simplicity and intelligibility markers, as well as the correct use of UML relations in use case models.

For actually discovering whether a quality goal is satisfied, each of its accompanying questions came loaded with a set of metrics to answer them (see Table 1). The metrics also have an associated gathering procedure that indicates how they should be measured. Those procedures can be rather simple, e.g. eval-

uating the length of a use case, or they can be significantly more complex, e.g., determining sections in the document with duplicate functionality descriptions. The most interesting procedure in *ReUse* is the one that detects duplicate functionality. Automating this procedure is not easy because software requirements are written in natural language, bringing along difficulties such as synonyms or ambiguities, among others. To overcome such obstacles, we combined *Machine Learning* (ML) techniques [1,12] with *Sequence Alignment* (SA) algorithms [16].

Fig. 4. Hierarchy of Domain Actions for Use Cases



The strategy proposed to compare use case steps (that describe interactions with the system) and thus locate duplication consists of identifying abstractions in use cases so as to clarify their intention and semantics. An abstraction removes the particularities of each expression and makes it possible to deal with use case steps by their intention and not by the terms that define them. Figure 4 depicts the type of categories (i.e., abstractions) that we defined for the use case domain. Such categories, called *Domain Actions* (DAs), are arranged in a hierarchical structure that goes from classes of actions (at roots) to specific actions (at leaves). DAs are a refinement of the work presented in [15]. Use case specifications are processed with a previously trained DA classifier, generating a discretization of every use case step and revealing their "true" semantic meaning.

After classifying the steps, the search for duplicate functionality is performed by comparing the DAs found in the use cases. For this purpose, DAs of particular use case sections like a basic flow are assembled in what we called sequences. Under this new representation, we use a *sequence alignment* algorithm (provided

Table 2. Refactorings available in *ReUse*

Refactoring	Description	Priority
Generate Inclusion Relation	Add inclusion relation between two use cases to avoid duplicate functionality	High
Generate Extension Relation	Add extension relation between two use cases to avoid duplicate functionality	High
Generate Generalization Relation	Add generalization relation among use cases promoting abstraction of shared functionality	High
Extract Early Aspect	Encapsulates crosscutting concerns related to an early aspect	Medium
Extract Use Case	Divide use case that describes more than one functional requirements, reducing its complexity	Medium
Unify Use Case	Unify several use cases that describe functional requirements that are very simple, improving the maintainability of the use case model	Medium
Remove de Inclusion / Extension / Generalization Relation	Delete a relation that, due to the evolution of requirements, has lost meaning in the model and does not comply with use cases semantics (e.g., including a use case that no other use case includes)	Low
Remove Use Case / Actor	Remove an element of the use case model lacking of meaning, either because it was duplicated or it is obsolete after requirements evolution	Low

by JAligner[3]) that, treating use case specifications as DNA chains, computes the similarity between portions of functionality described in the use cases. The comparison made with the algorithm finds the relative position among two sequences that maximizes their similarity (behaviorally speaking) using a customizable scoring system. For more information, the reader is referred to [16].

3.2 Determine Candidate Refactorings

The second activity is for determining which instruments (given by use cases) can mend the deficiencies identified before. To achieve this goal, *ReUse* analyzes different kind of refactorings for each problem. A refactoring is a solution for a particular deficiency that makes sense when certain conditions are met. Our approach is equipped with a refactoring catalogue derived from related works [14,18,19]. The same derivation took special care for the (semi)automation of the improvement process. Table 2 enumerates the list of refactorings available. Each refactoring has five parts: (i) an application context (i.e., the conditions),

[3] http://jaligner.sourceforge.net

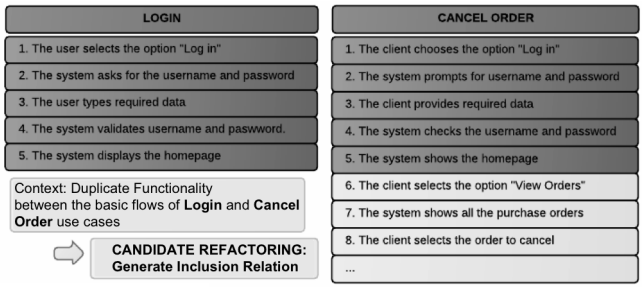
(ii) a general description of the solution, (iii) a systematic mechanism to execute the refactoring (in the text), (iv) a priority, and (v) samples that explain the refactoring in action. As example, Table 3 gives details on the different parts of the Generate Inclusion Relation refactoring.

Table 3. Detailed Refactoring: Generate Inclusion Relation

Generate Inclusion Relation	
Context	There is a portion of functionality duplicated in the basic flows of two or more use cases, which is not affected by any condition. If in one of the use cases the duplication covers the complete basic flow, then some special considerations are made because it is an special case of the refactoring.
Solution	Create a new use case encapsulating the duplicate functionality and generate an inclusion relation from the base use cases to the newly created one. Finally, remove the duplicated functionality of the base use cases.
Mechanism	1. Create a new use case in which the duplicated functionality is not encapsulated in some other use case of the model. Reference it as the use case to be included. 2. Name the use case accordingly. 3. Identify the sequence of use case steps to be removed of each base use case. 4. Move the use case steps from the base use cases to the included one. 5. Move the actors no longer participating in the base use cases to the included one, and copy the ones that participate on all of them. 6. Move the alternative flows of the base use cases that are exceptions of the sequence moved to the included use case. 7. Remove the duplicated use case steps from the base use cases. 8. Generate an inclusion relation between the base and the included use cases. 9. Modify the specification of the base use cases making an explicit reference to the included use case.

In order to establish if a refactoring for a particular problem is appropriate, the approach analyzes the use case model along with the collected metrics and evaluates if the necessary conditions are satisfied. For instance, observing the metric that assessed duplicate functionality between use cases it is possible to suggest a refactoring whose goal is the reuse of functional descriptions, such as Generate Inclusion Relation. Furthermore, for that refactoring to be applicable, the approach must check that the duplication occurs on basic flows of the two use cases (a requisite for this kind of relations). As example, Figure 5 shows an excerpt of two use cases of the Online Shopping System that are considered for improvement by means of an inclusion refactoring. After the analyses are done, the he refactorings suggested with *ReUse* and their impact over the use cases can be visualized graphically, as depicted in Figure 2.

Fig. 5. Determining Candidate Refactorings in two Use Cases



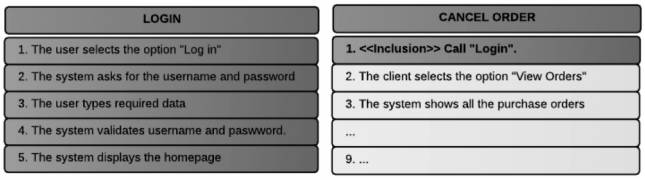
3.3 Select Refactoring

The third activity in our approach assists analysts to chose some of the suggested refactoring to mend an individual deficiency in the current iteration. Since certain deficiencies are more relevant and bring greater benefits in the quality of requirements specifications, *ReUse* helps the users of the tool by generating a ranking of the suggested refactorings. The underlying idea here is that the analysts should probably solve the most important deficiency first. The score of each refactoring is computed considering two factors: the confidence of detection of the deficiency and the priority of the problem (obtained from the refactoring). The confidence value indicates the "trust" we have regarding the detection of repeated chains. In the case of duplicate functionality, the sequence alignment algorithm produces information regarding the certainty of the matched use cases. The priority value is associated to each refactoring, and it indicates the relevance of the problem address (see Table 2). Moreover, there is specific information about each refactoring that allows to weight the priority. For example, before the execution of a Generate Inclusion Relation refactoring, we can take into account the percentage of the use case affected by duplication. Using these parameters, the approach is able to compile a list of refactorings ranked by importance.

3.4 Apply Refactoring

The fourth activity, which is also the last of the iterative improvement schema defined in *ReUse*, has the objective of applying the refactoring selected by an analyst to textual use cases. In order to "execute" the refactoring, the tool searches the selected refactoring in the catalogue and runs the sequence of steps described in its mechanism part. This mechanism transforms a faulty specification into one of better quality. Considering the previous example, Figure 6 shows the result of applying the Generate Inclusion Relation refactoring to a duplicate functionality deficiency. Since the duplication is already encapsulated within the *Login* use case, creating a new use case is not necessary (see Figure

Fig. 6. Executing a Use Case Refactoring



5). Regarding the *Cancel Order* use case, the approach only has to remove the duplicate use case steps (numbered from 1 to 5) because there are not actors or alternative flows that need to be deleted. Once these steps are subtracted, an inclusion relation is added from *Cancel Order* to *Login*. This means adding an explicit reference from the former to the latter use case.

4 Preliminary Evaluation

With the purpose of validating our approach, we conducted a series of experiments with publicly-available cases-studies. The main goal of this preliminary evaluation was to corroborate the performance of the approach to identify deficiencies (mainly, duplicate functionality) and suggest refactorings to solve them. The experiments were carried out on five systems, namely: DLibraCRM [4], MobileNews [4], WebJSARA [4], HWS [8], y CRS [2]. The first three systems comprise use case specifications that are part of the Software Development Studio (SDS) project. The use case models of DLibraCRM, MobileNews y WebJSARA contain 15, 15 and 29 use cases, respectively. The length of the requirements in these systems is of approximately 13, 12 and 22 textual pages, respectively. The fourth system is called Course Registration System (CRS) [2], a distributed system used to manage courses and registrations of a university. The documentation of CRS contains 8 use cases (approximately 20 pages). The fifth and last case study is the Health Watcher System (HWS) [8], a web system that mediates between citizens and municipal government to attend health-related topics. The documentation of HWS contains 9 use cases (approximately 19 pages).

For quantifying the performance of the approach, we chose to adapt measures derived from the *Information Retrieval* field [12]. Similar experimental assessments have used these measures before [4], observing the values of *Precision*, *Recall* and *F-Measure* to draw empirical conclusions. In the context of our work, there are to factors at play: (i) the interpretation of the measures in this experiment, and (ii) the reference solution (baseline) used to compare the outputs of the tool and so compute the measures. On one hand, *Precision* gauges the rate of correct suggestions (TP: true positives) in contrast to incorrect suggestions (FP: false positives). False positives can be caused by the identification of a deficiency

that actually is not, as well as the recommendation of a wrong refactoring for a true deficiency. This measure is computed as: $Precision = \frac{tp}{tp+fp}$. On the other hand, *Recall* gauges the rate of correct suggestions (TP) in contrast to the real number of deficiencies in the case-studies. This means that to compute *Recall*, we need to know the deficiencies that the tool omitted (FN: false negatives). This measure is computed as: $Recall = \frac{tp}{tp+fn}$.

To prevent biasing the experimentation, we entrusted the analysis of the use case specifications of the case-studies to four senior functional analysts. They were in charge of manually inspecting textual documents, identifying faulty or duplicate requirements in the specifications and deciding which alternatives would help to improve the use cases. The analysts worked individually and had approximately eight hours to complete this task. Next, we arranged a meeting with the analysts, in which they could present their discoveries. After a constructive discussion, they reached to a mutual agreement about the deficiencies in the case-studies. The result of this exercise allowed us to elaborate a reference solution, that embodies the reasoning of the analysts and permits to make comparisons and thus compute *Precision* and *Recall*.

The evaluation aimed at corroborating if the approach can discover the deficiencies in the use cases, and if it works well on real-world requirements specifications. The experiments comprised running *ReUse* in the five case studies, without taking into account the intervention of human analysts. The deficiencies and refactorings outputted with the tool were executed progressively, selecting in each iteration the recommendation at the top of the ranking generated by the approach. The results of this experiment were compare to the reference solutions of the case-studies. This way, we were able to compute *Precision* and *Recall*. Since the case-studies did not contain simple mistakes, like meaningless use cases or actors, the evaluation is mainly focused on duplicate functionality. It is worth noting that during the experiments, we were forced to make some exceptions regarding the suggestions of the prototype. Particularly, the approach could identify remarkably similar functional descriptions among two or more use cases, that did not necessarily indicate the presence of a deficiency. For example, use cases that describe interactions about create-retrieve-update-delete operations (CRUD) are usually written using the same terms, and consequently our tool marked them as candidate duplicate functionality. For the sake of the evaluation, we preferred to omit those suggestions in such instances.

Table 4. Experimental Results of *ReUse*

	DLibraCRM	MobileNews	WebJSARA	CRS	HWS	Total
TP	9	5	12	3	13	43
FP	6	1	10	1	7	25
FN	0	1	6	0	0	7
Precision	0.60	**0.83**	0.54	**0.75**	0.65	0.63
Recall	**1.00**	**0.83**	0.66	**1.00**	**1.00**	**0.86**

Table 4 summarizes the results of the experiment. The prototype recovered most of the deficiencies and also recommended the correct refactoring (~85% *recall*), without making many mistakes (~65% *precision*). In some of the case-studies, like DLibraCRM, CRS and HWS, our tool identified all the deficiencies of the specifications, achieving a 100% *recall*. The deficiencies were complemented with candidate refactorings. The majority of the refactorings recommended were Generate Inclusion Relation, Generate Extension Relation and Unify Use Cases. In MobileNews and WebJSARA, our tool obtained a ~85% and ~65% *recall*, respectively. Particularly, we noticed that the prototype got many FN in WebJSARA. This situation is attributed to a poor detection of duplicate functionality, which ultimately caused the omission of some refactorings (mainly, Generate Generalization Relation and Generate Inclusion Relation).

Regarding *precision*, the prototype attained acceptable results. In MobileNews and WebJSARA, our tool achieved *precision* values above 75%. In the rest of the case-studies, the tool got a lightly lower *precision* than before, averaging a 60%. This drop in precision was caused by a faulty detection of duplicate functionality, resulting in an incorrect recommendation of refactorings, such as Generate Inclusion Relation and Generate Extension Relation. However, the portion of the use cases mistakenly marked as duplicated share substantial lexical and semantic similarities (i.e., their terms and domain actions).

The experiments demonstrated the potential of our prototype. For an assistive RE tool like ours, we think that it is preferable having an acceptable *precision* to achieve a high *recall*. This line of thinking has also been followed in other evaluations of automated RE tools [6,7]. Nevertheless, it would be interesting to quantify the quality of duplicate functionality identification (i.e., boundaries) and the advantages of producing a ranking of refactorings before recommending the analysts.

5 Conclusions

In this article, we presented an approach called *ReUse* that facilitates the analysts' tasks for assessing the quality of requirements specifications, finding deficiencies and providing means to mend them in a iterative fashion. The approach is implemented as a prototype tool that supports the analysis of textual use cases by combining advanced text processing techniques. Specially, *ReUse* leverages on a domain-specific classifier (of use cases) to obtain an abstract representation of the requirements and, using this knowledge, adapts a sequence alignment technique to search for duplicate functionality. Thereafter, redundant functionality is compared with a catalogue of use case refactorings, recommending the best-suited solutions that allow to (semi-)automatically solve the deficiencies.

We performed a preliminary evaluation with five publicly-available case studies covering different system domains. The results of the experiments were encouraging, achieving a very good *Recall* and an acceptable *Precision*. Attaining

a high *Recall* means that the prototype was able to identify the majority of the deficiencies in the documents, which is really important from the analysts' viewpoint. The *Precision* was significantly lower than *Recall*, what means that the tool is prone to make some mistakes identifying deficiencies and recommending refactorings. Still, we are confident that a human analyst will discard these mistakes by just taking a quick look at the outputs of *ReUse*.

Despite the satisfactory results, we could notice some limitations of the approach during the experiments as well. For instance, the techniques employed failed to differentiate those behaviors that were similarly written from those that are actually similar (e.g., CRUD use cases). Furthermore, *ReUse* sometimes failed to identify well the boundaries of duplicate functionality. As future work, we are analyzing alternatives to refine the hierarchy of domain actions so as to better capture the semantics of use cases. Also, we are planning to adjust the parameters of the sequence alignment technique to improve the detection of duplicate functionality. Furthermore, we will evaluate the tool with more case studies to assess the performance of *ReUse*.

References

1. Baeza-Yates, R., Ribeiro-Neto, B., et al.: Modern information retrieval, vol. 463. ACM press New York. (1999)
2. Bell, R.: Course registration system. `http://sce.uhcl.edu/helm/RUP_course_example/courseregistrationproject/indexcourse.htm` (2011)
3. Chernak, Y.: Building a foundation for structured requirements. aspect-oriented re explained - part 1. Better Software (January 2009)
4. Ciemniewska, A., Jurkiewicz, J.: Automatic Detection of Defects in Use Cases. Master's thesis, Poznan University of Technology (2007)
5. Cockburn, A.: Writing effective use cases, vol. 1. Addison-Wesley Reading (2001)
6. Cuddeback, D., et al.: Automated requirements traceability: The study of human analysts. In: 18th IEEE Int. Req. Eng. Conf. pp. 231 –240 (October 2010)
7. Dekhtyar, A., et al.: On human analyst performance in assisted req. tracing: Statistical analysis. In: 19th IEEE Int. Req. Eng. Conf. pp. 111 –120 (2011)
8. Greenwood, P.: Tao: A testbed for aspect oriented software development. `http://www.comp.lancs.ac.uk/~greenwop/tao/` (2011)
9. Hull, E., Jackson, K., Dick, J.: Requirements Engineering. Springer (2010)
10. Jacobson, I., et al.: The unified software development process. A-W (1999)
11. Kamata, M.I., Tamai, T.: How does req. quality relate to project success or failure? In: Procs. of the 15th IEEE Int. Req. Eng. Conf. pp. 69–78 (2007)
12. Manning, C., et al.: Introduction to Information Retrieval. CUP (2008)
13. Ramos, R., et al.: Quality improvement for use case model. In: SBES'09. pp. 187–195. IEEE (2009)
14. Rui, K., Butler, G.: Refactoring use case models: the metamodel. In: Procs. of the 26th Australasian Computer Science Conf. pp. 301–308 (2003)
15. Sinha, A., et al.: An analysis engine for dependable elicitation of natural language use case description and its application to industrial use cases. IBM Report (2008)
16. Smith, T., Waterman, M.: Identification of common molecular subsequences. Journal of Molecular Biology 147(1), 195 – 197 (1981)

17. Van Solingen, R., Berghout, E.: The Goal/Question/Metric Method: a practical guide for quality improvement of software development. McGraw-Hill (1999)
18. Xu, J., Yu, W., Rui, K., Butler, G.: Use case refactoring: a tool and a case study. In: 11th APSE Conf. pp. 484–491. IEEE (2004)
19. Yu, W., Li, J., Butler, G.: Refactoring use case models on episodes. In: Procs. of the 19th Int. Conf. on Aut. Soft. Eng. pp. 328–335. IEEE (2004)