Softw Syst Model (2016) 15:579–603
DOI 10.1007/s10270-014-0431-3

REGULAR PAPER

Identifying duplicate functionality in textual use cases by aligning semantic actions

Alejandro Rago · Claudia Marcos · J. Andres Diaz-Pace

Received: 19 December 2013 / Revised: 20 June 2014 / Accepted: 8 August 2014 / Published online: 27 August 2014

Abstract Developing high-quality requirements specifications often demands a thoughtful analysis and an adequate level of expertise from analysts. Although requirements modeling techniques provide mechanisms for abstraction and clarity, fostering the reuse of shared functionality (e.g., via UML relationships for use cases), they are seldom employed in practice. A particular quality problem of textual requirements, such as use cases, is that of having duplicate pieces of functionality scattered across the specifications. Duplicate functionality can sometimes improve readability for end users, but hinders development-related tasks such as effort estimation, feature prioritization, and maintenance, among others. Unfortunately, inspecting textual requirements by hand in order to deal with redundant functionality can be an arduous, time-consuming, and error-prone activity for analysts. In this context, we introduce a novel approach called *ReqAligner* that aids analysts to spot signs of duplication in use cases in an automated fashion. To do so, *ReqAligner* combines several text processing techniques, such as a *use case-aware classifier* and a customized algorithm for *sequence alignment*. Essentially, the classifier converts the use cases into an abstract representation that consists of sequences of semantic actions, and then these sequences are compared pairwise in order to identify action matches, which become possible duplications. We have applied our technique to five real-world specifications, achieving promising results and identifying many sources of duplication in the use cases.

Keywords Use case modeling · Use case refactoring · Natural language processing · Sequence alignment · Requirements engineering · Machine learning

1 Introduction

In mostly any software development, getting a clear understanding of system requirements and describing them in an accurate and unambiguous manner is a necessity [22]. A deficient analysis and documentation of requirements usually has negative effects downstream, compromising the success of a project [26]. In fact, several studies have shown the correlation between overtime/overbudget projects and poor-quality requirements [32]. Requirements are generally specified using textual specifications [30]. In this work, we focus on use case specifications, as they are widely used in industry [22]. A use case captures a behavioral contract between the system and its external actors [5], documenting their interactions via textual descriptions with little or no structure.

In spite of existing guidelines for writing use cases [1,10], the harsh reality is that the actual use cases generated in software projects do not often meet the standards of what it is considered a "good" use case model [15]. We argue that the

Communicated by Prof. Daniel Amyot.

This work was partially supported by ANPCyT, CONICET and CIC (Argentina) through PICT Project 2010 No. 2247, PIP Project 2012-2014 No. 11220110100078, and Project No. 813/13, respectively.

A. Rago (✉) · C. Marcos (✉) · J. A. Diaz-Pace (✉)
ISISTAN Research Institute, UNICEN University,
Paraje Arroyo Seco, Tandil, Argentina
e-mail: arago@exa.unicen.edu.ar

C. Marcos
e-mail: cmarcos@exa.unicen.edu.ar

A. Rago · J. A. Diaz-Pace
CONICET, Buenos Aires, Argentina

J. A. Diaz-Pace
e-mail: adiaz@exa.unicen.edu.ar

C. Marcos
CIC, Buenos Aires, Argentina

duplication of functionality in use case specifications is a major problem in software development. Duplicating functionality is the action of repeating the description of some interactions between the system and actors [8]. We are interested in detecting situations of duplicate functionality in use cases because they impact on later development stages such as architectural design, implementation, testing, and maintenance [24].

Several factors contribute to this duplication phenomenon, namely: applying use cases in large projects with many requirements, having inexperienced analysts documenting the requirements, or having requirements that change very often. There are also some authors that ascribe the problem to the abuse of copy/paste features in text processors [24]. In particular, we argue that duplicate functionality is a problem that analysts should deal with since early development stages. Although duplication is not always a quality defect, and it might be there for the sake of readability of non-technical stakeholders, the issues entailed to lack of modularity and abstraction can have a profound (negative) effect on the developers conducting activities such as effort estimation [27], project planning , architectural design [6], change impact analyses [23], and evolution management [1].

During the last decade, several researchers studied ways of improving textual requirements [8,15,48]. They developed a number of catalogs of bad smells (or defects) for requirements documents, which enable analysts to identify a poor-quality specification by looking at some writing patterns or metrics. Unfortunately, these catalogs require analysts to manually inspect the requirements for addressing each type of defect, as it is the case of duplicate functionalities. Some advances have been made in the automated discovery of defects [18,24], but unfortunately these works are presented as a proof-of-concept for duplicate functionality and do not provide support to improve the specifications afterward. An interesting alternative to address this challenge is to have an expert system (or assistant) able to spot potentially duplicate portions of functionality in textual requirements, and then ask the analyst to decide which of these duplications correspond to real defects.

In this context, we present a novel approach called *ReqAligner* that aids analysts in the search of duplicate functionality in textual use cases by means of advanced text-processing techniques. The contributions of this work are twofold. First, we have implemented an algorithm that permits to quickly discover duplicate functionality by combining state-of-the-art *Natural Language Processing* (NLP) and *Machine Learning* (ML) techniques with *Sequence Alignment* (SA) algorithms (typically used in the bioinformatic domain [34]). Second, we have built a prototype that supports the process of improving use cases. To do so, our *ReqAligner* tool basically guides the analyst in the visualization of behavior that appears to have duplication across a set of use cases. Furthermore, the tool recommends alternatives to resolve such duplication in the form of use case refactorings that use UML relationships (e.g., inclusion, extension, inheritance). In order to evaluate our approach, we have performed a series of experiments in five publicly available case studies. These experiments were developed to address questions related to: (i) whether *ReqAligner* works well in real-world specifications and (ii) whether the recommendations provided to analysts are helpful for improving the use cases. The results obtained so far are encouraging, as our approach has correctly detected most duplication problems and provided helpful refactoring solutions for the affected use cases.

The rest of this article is organized in 8 sections. Section 2 discusses the adverse effects of duplicate functionality on software development activities. Section 3 introduces our approach and shows how an end user interacts with the *ReqAligner* tool. Sections 4, 5, and 6 discuss the combination of techniques applied for discovering duplicate functionality in use case steps. Section 7 reports on the experiments performed with the tool and their results. Section 8 reviews related techniques and tools and how they stand with respect to our approach. Finally, Sect. 9 gives the conclusions and analyzes future lines of work.

2 The effects of duplicate functionality on software development

Requirements of poor quality, being ambiguous, incomplete and/or redundant, are often the cause of time and cost overruns in software projects [26]. For this reason, several researchers have studied the so-called *requirements (bad) smells* [9,18]. A requirement smell is a deficiency in the requirements and a concrete symptom for a quality defect. If it happens that an analyst finds one of such smells, then it brings the opportunity to improve the quality of the requirements. Requirement smells can be categorized according to their extent. In use cases, some smells are confined to individual use case steps (e.g., ambiguous adverbs and adjectives, vague pronouns, negative statements or subjective language) [18]. Other smells, however, are confined to a single use case (overly complex or simple scenarios or "happy" use cases, among others) [10]. Lastly, some smells cannot be confined to a single document and impact on several use case specifications, as it is the case of duplicate functionality descriptions [9]. This kind of smell is the one addressed in our work.

We argue that the duplication of functionality in the use case specifications is a risky smell. Duplication can occur at different granularities. If two individual steps of different use cases are literally written with the same words, they can be considered a duplication. Moreover, if two individual steps are written with different terms but they mean the same, either because they refer to the same semantic behavior

and/or work on the same entity (e.g., changing some information or operating user's profile), they can also be considered a duplication. It could also happen if high-level "transactions" (comprising at least 3 use case steps) perform the same functional operation in different use cases, we may be in presence of duplication across use cases.

Analysts should pay special attention to duplicate functionality among use cases because this smell can be the source of critical defects downstream [24]. From a development viewpoint, getting rid of duplicate functionality in specifications brings several advantages [27], namely: it clarifies the purpose of use cases, simplifies functional documentation, streamlines use case prioritization, improves effort/work estimations (e.g., using use case points) and project planning (enhancing the allocation of work assignments), and refrains the implementation of repeated functionality [17]. Also, a well-modularized (duplication-free) model should help to maintain requirements in evolving systems, confining changes to a single location [44].

Fortunately, use cases do have support for managing duplicate functionality. Since it is natural for use cases to share some common behavior, it is efficient to reuse existing text rather than repeating the same sequence of events every time they are needed. Jacobson defined the concepts of relations among use cases (inclusion, extension and generalization) to handle these situations [1]. However, if analysts make an excessive usage of relations, stakeholders without a strong Software Engineering background may have a difficult time to read and understand the specifications. The inclusion relation, for instance, often contributes to readability because it simply factors out repeated functionality [28,44]. Extension and specially generalization relations, in turn, occasionally interfere with the comprehension of functional requirements by end users, distracting them during the validation of key features [44]. This trade-off between readability and modularity should be carefully handled by analysts, striking a balance between the creation of intelligible requirements and the convenience of the development staff.

3 The *ReqAligner* approach

In order to help analysts to produce good-quality requirements descriptions, we have developed an approach that discovers textual portions of use cases containing semantically similar functionality and presents those scenarios as candidate duplication deficiencies. The analyst can then intervene and decide whether the specification should be rephrased or amended. Our approach relies on recent technological advancements on NLP [43] that make the problem of identifying duplication symptoms tractable in computational terms.

Our approach is called *ReqAligner* (Requirement analyzer with sequence Aligner) and works as follows (see Fig. 1). Initially, the analyst inputs a set of textual use cases. These use cases are analyzed by several NLP components (step 1: BASIC NATURAL LANGUAGE PROCESSING) with the end goal of gathering elementary knowledge about the textual scenarios (that is, basic and alternative courses of the use cases).

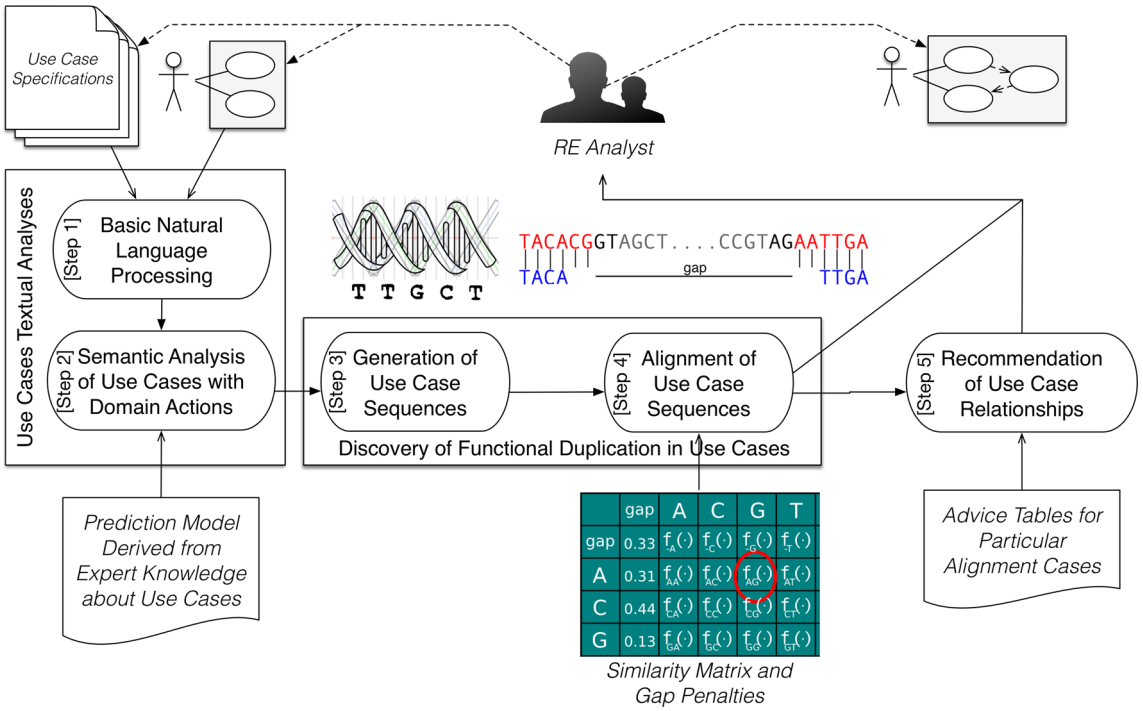


Fig. 1 The *ReqAligner* technique for use case specifications

The NLP components output information such as sentences and token boundaries (*sentence splitter* and *tokenizer*), token properties (*part-of-speech tagger* and *stemmers*), and semantic constituents (predicates and arguments with *semantic role labeling*).

Once these initial analyses over the use cases are completed, the intention of each sentence of the specifications is examined (in the context of Requirements Engineering) by leveraging on particularities of the use cases domain (step 2: SEMANTIC ANALYSIS OF USE CASES WITH DOMAIN ACTIONS). This means that *ReqAligner* tries to determine the meaning of a use case step as it was conceived by the analyst who wrote it. For instance, it might predict if a use case step informs about an interaction with an external system or if it is actually an inquiry for the users to input data, among other kind of interactions. With this information, the tool derives an abstract representation of the use case scenarios in the form of what we refer to as *semantic actions*. These semantic actions are derived from both a large number of specifications and recent studies about the writing style of use cases [25,45].

After the semantic analyses of use case steps is completed, the approach assembles sequences (or chains) of the semantic actions generated earlier (step 3: GENERATION OF USE CASE SEQUENCES). The resulting sequences represent a summarized view of a scenario, which enables the comparison of scenarios based on their overall semantics. At this point, the different sequences are processed and matched (pairwise) by means of a customized *sequence alignment* (SA) technique [34] (step 4: ALIGNMENT OF USE CASE SEQUENCES). The sequence alignment basically compares the constituent elements of two different sequences (also referred to as symbols or characters in the bioinformatics jargon) and aims at finding a subchain that maximizes a given similarity function. When two sequences go under analysis, the comparison depends on two pre-defined parameters [34]: the substitution matrix and the penalization table, which we adapted to the use cases domain. On one hand, the substitution matrix defines the similarities between the use case steps (according to their intention). On the other hand, the penalization table defines adjustments (i.e., penalties) the aligner should apply to the similarity score when there are gaps between the matched sequences.

If some textual portions of a pair of use cases were successfully aligned, it means that we have one or more candidate duplications in the functional specification. Based on these results, the approach applies simple heuristics to determine the kind of problem detected and the most likely UML relationship that can help to refactor (and thus, improve) the use cases (step 5: RECOMMENDATION OF USE CASE RELATIONSHIPS). The idea here is not to propose new refactorings but rather to take advantage of existing knowledge about requirements quality assessments [41,48]. We believe that a set of simple rules are sufficient to aid analysts in the improvement of the textual use cases. For instance, these rules can analyze features such as the origin of the sequences, or their extension (among other properties), in order to determine which relationship could help solving a particular duplication between two use cases. At last, the information gathered in the two previous activities is presented to the analyst. The output of the approach is a set of candidate defects related to duplicate functionality and recommendations for fixing them. It is up to the analyst to corroborate the findings of *ReqAligner* and make an informed decision, potentially accepting the recommendations given by the tool.

ReqAligner has been implemented as a set of Eclipse plugins, which allow analysts to interact with the tool during the process of analyzing the use cases. From an analyst's standpoint (as a user of the tool), she is only responsible for feeding the textual use cases into the tool, and then executing the whole analysis for obtaining an output. The GUI is divided into three main panels, as depicted in Fig. 2. On the left panel (atop), the user has the list of use cases of the system. One the right panel (atop), there is a view for comparing side-by-side the textual contents of two use cases. The user can also inspect the duplications found, highlighted with colors at the level of use case steps. In the bottom panel, there is an execution button that starts the analysis. In that same panel, once *ReqAligner* has computed the alignments, the user gets the list of duplicate functionalities, each one associated with use case refactorings. The user can simply click on any of them to update the detailed use case viewer.

In order to better understand the approach, let us introduce a motivating example of an automatic teller machine (Fig. 3). For the sake of brevity, the specifications only cover a subset of the system documentation, including five use cases, namely: *Transfer Money*, *Withdraw Money*, *Deposit Money*, *Pay Service* and *Add Service*. The first three use cases allow clients to perform money operations (withdraw, deposit or transfer) with their accounts. The *Pay Service* and *Add Service* use cases allow clients to pay services such as electricity and internet service bills from the ATM terminal. Figure 4 shows a digest of the specifications of the last two use cases.

At first glance, an analyst inspecting the ATM use cases will not find any defect whatsoever in the use cases described in Fig. 4. A deeper look into the use cases *Add Service* (UC#1) and *Pay Service* (UC#2) further confirms that they are indeed expressing distinct functionalities. However, the resemblance between the basic flow of UC#1 and the alternative flow of UC#2 constitutes a duplicate behavior. This kind of problems is not easily discernible for the untrained eye, and we expect *ReqAligner* to help in this regard. In this particular situation, the use cases can be improved by using an extension relationship (UC#1 extends UC#2 when there are no registered services). Figure 5 shows the resulting use

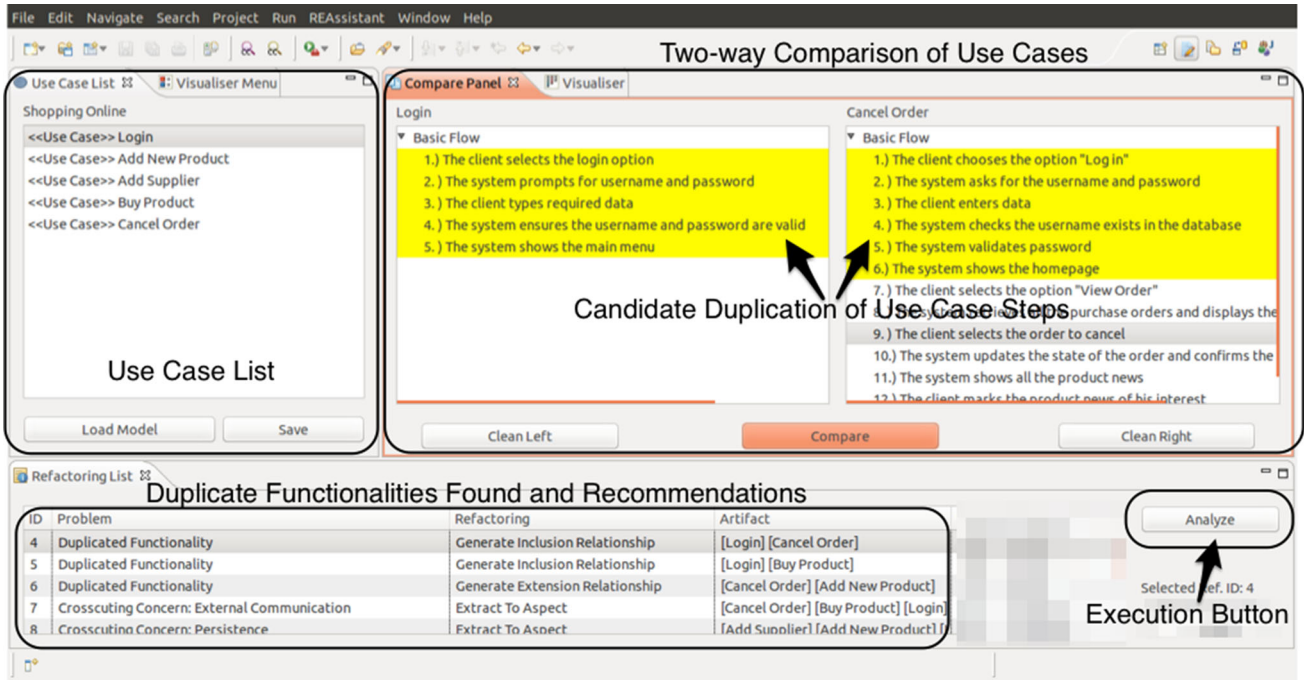


Fig. 2 Snapshot of the *ReqAligner* tool at work

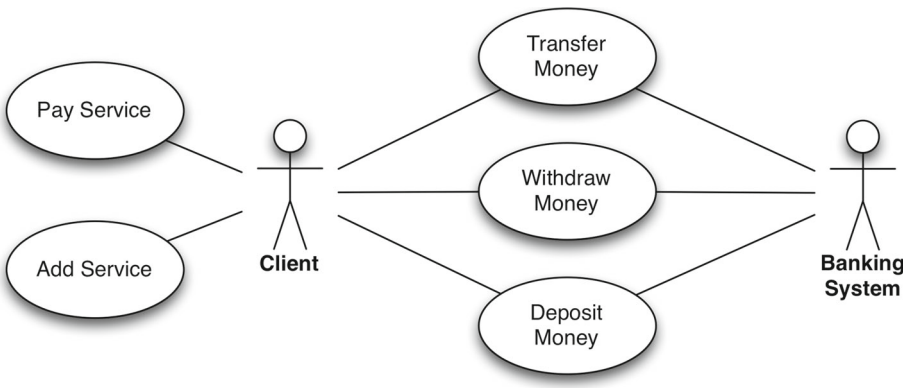


Fig. 3 Use cases from an ATM system

case model after applying the refactorings. In the following sections, we will explain the details of the activities executed by *ReqAligner* based on our motivating ATM example.

4 Analysis of textual use cases

For making useful inferences out of textual requirements, we need to understand what they mean in the context of the system under development. The first block (USE CASES TEXTUAL ANALYSES) shown in Fig. 1 (on the left) holds two activities defined in *ReqAligner* to deal with this endeavor: BASIC NATURAL LANGUAGE PROCESSING and SEMANTIC ANALYSIS OF USE CASES WITH DOMAIN ACTIONS. The subsections below explain the conceptual and technological underpinnings of these activities.

4.1 Basic natural language processing

The first activity processes the raw textual contents that come along with the use case specifications. Specifically, this activity runs a pipeline of NLP modules provided by third parties and generates a number of annotations for later analyses (e.g., detection of duplicate information). The textual use cases are initially divided into sentences (*Sentence Splitting*). The sentences are then divided into tokens holding the information of individual words (*Tokenizer*). Both the *Sentence Splitting* and *Tokenizer* are implemented by the OpenNLP[1] library. The tokens are further analyzed by other modules, which retrieve their parts of speech (*POS Tagging*), their syntactical and morphological roots (*Stemming and Lemmatizing*), and

[1] OpenNLP: available at http://opennlp.apache.org.

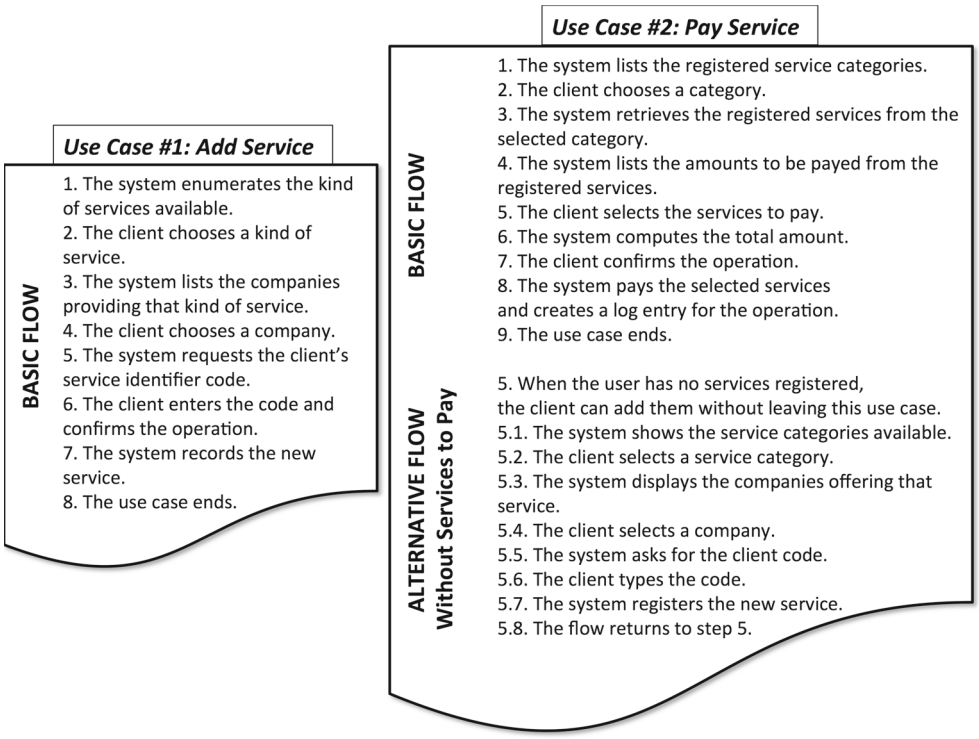
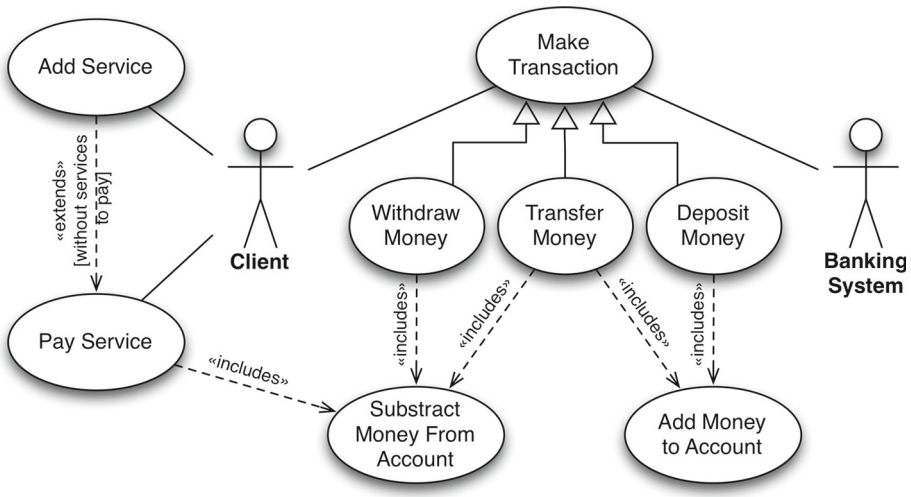


Fig. 4 Use case specifications from the ATM system

Fig. 5 Improved ATM use case model



their relevance to statistical analyses (*Stop-Words Detection*). To do so, we relied on several libraries available in the NLP community, namely: the OpenNLP library and the Stanford CoreNLP[2] package for *POS Tagging*, the Snowball[3] package for *Stemming* and the Mate-Tools[4] package for *Lemmatizing*, and a publicly available word list for *Stop-Words Detection*.[5]

Finally, the pipeline ends with an advanced NLP module that attempts to determine the meaning of a sentence and its constituents. This module, called *Semantic Role Labeling* (*SRL*), recognizes the main predicate of a sentence and its arguments [33], and also labels these constituents with descriptions about their semantic role within the sentence. For *Semantic Role Labeling*, we used an implementation that is part of the Mate-Tools package.

Figure 6 shows a graphical representation of the information produced after executing the NLP pipeline discussed above on a use case step (an excerpt of the example of previous sections). The text of the use case step is first divided into

[2] Stanford CoreNLP: available at http://nlp.stanford.edu/software/corenlp.shtml.

[3] Snowball: Available at http://snowball.tartarus.org.

[4] Mate-Tools: available at http://code.google.com/p/mate-tools/.

[5] Available at http://www.ranks.nl/resources/stopwords.html.

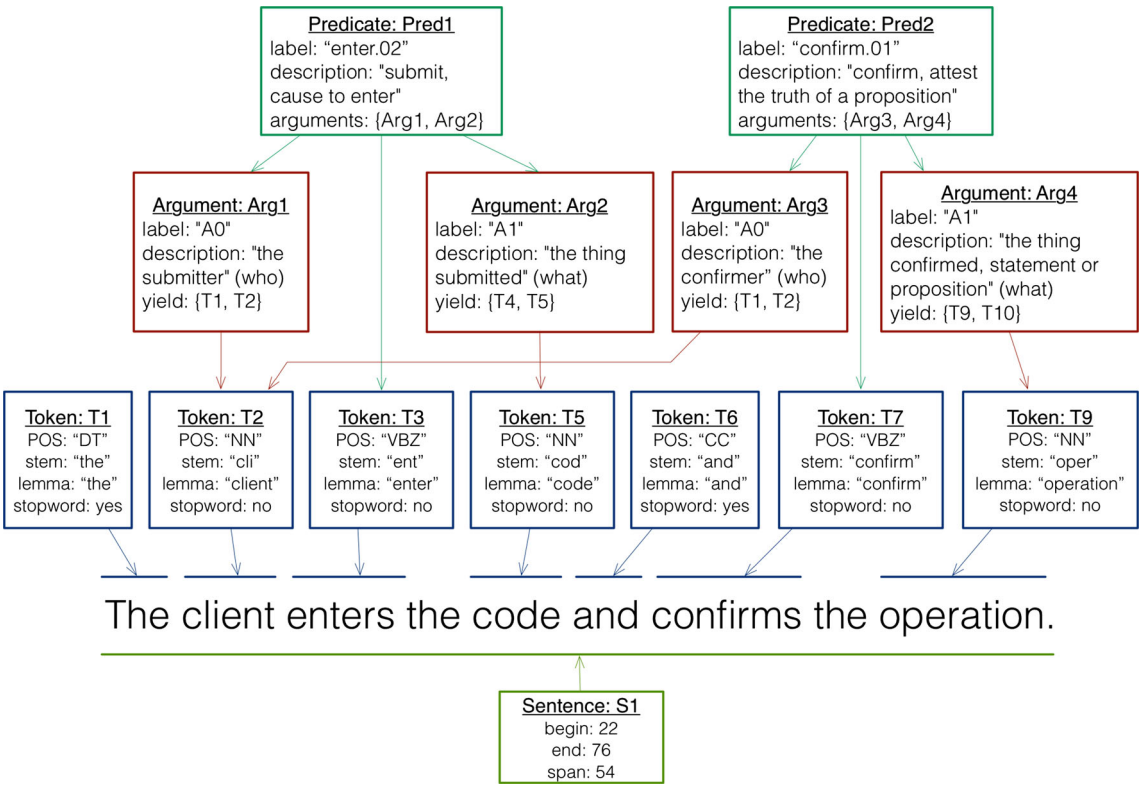


Fig. 6 Execution of NLP modules in a use case step

tokens (boxes T1 to T10). Then, these tokens are enriched with properties like parts-of-speech, stems, and lemmas. In token T5, for example, the POS property indicates that the syntactical function of the word is noun. Predicates and arguments can be found right above tokens, since they are built upon their properties and represent a semantic interpretation of the text. In the example, there are two predicates and four arguments (Arg1 and Arg2 for Pred1 and Arg3 and Arg4 for Pred2). The reason this sentence has two predicates (instead of just one) is because there is a conjunction operator ("and") within the use case step. Predicate Pred1 describes an action for "submitting or cause to enter" something, whereas Pred2 is an action for "confirming or attesting the truth of a proposition". Finally arguments delimit the "who" (submitter and confirmer, respectively) and the "what" (the thing submitted and confirmed, respectively) of each predicate.

4.2 Semantic analysis via domain actions

Tracking down and identifying patterns of functionality repeated across several textual sentences is a complex activity, mainly due to the expressiveness and ambiguity of the English language. For instance, a common obstacle for automated analysis is when two (or more) analysts describe the same behavior using a dissimilar set of terms and a very different organization (syntactically speaking). Therefore, for the sake of comparing two textual requirements, a detection procedure should not only focus on finding repeated terms, but also on understanding (at least, approximately) the intent of these requirements. By intent, we refer to the meaning of a particular requirement in the context of the system under development. Although inferring intent is not an easy undertaking, there are certain peculiarities about requirements specifications that can be exploited to deal with the problem.

In order to simplify the analysis of duplicate functionality, we took advantage of two properties of use case specifications: (i) the lexicon (i.e., terminology) used to describe behavior is generally limited; and (ii) the "story telling" of scenarios that, if analysts follow the recommended guidelines, often retains a common structure and employs semantically similar concepts (e.g., inputting data or processing information, among others). Based on these two properties, every use case step can be viewed as (or reduced to) a *Domain Action* (DA). A DA represents an abstraction of a recurrent interaction between the system and actors. For example, a DA could group use case steps related to data management, user interactions, or information displaying, among others. Similar abstractions have already been discussed by other researchers [25,45], applying classification techniques for reducing use case steps to simpler and more conceptual classes. For defining our own categorization of DAs, we specifically relied on Sinha's work [45], because the abstraction level of their classes fitted well with the kind of analy-

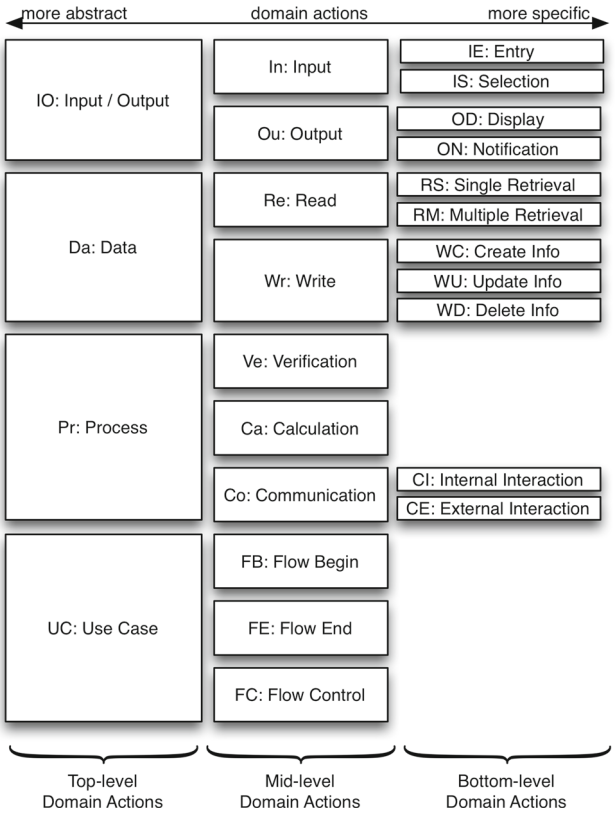


Fig. 7 Hierarchy of domain actions (use case abstractions)

ses made by *ReqAligner*. We made a refinement of Sinha's classes, removing some system-specific classes that were not suitable for our purposes (for example DELEGATE and UNCLASSIFIED) and renaming some others. We also incorporated classes for recognizing control flow sentences, which contain terminology that is specific to use cases. A difference with previous works is that we organized our DA classes into a hierarchy. This feature not only enables our technique to identify the intention of use case steps at different levels of abstraction, but also helps to improve the accuracy of classification in predictive applications [47]. Overall, we defined 25 specific DA classes and arranged them in a three-level hierarchy. Figure 7 sketches the complete hierarchy. The upper level of the hierarchy groups DAs associated to semantically similar interactions, such as INPUT/OUTPUT, DATA, PROCESS and USE CASE. The middle level introduces more specific information. For example, the DATA top DA is divided into two mid-level DAs: READ and WRITE related interactions. Finally, the bottom level provides information at the lowest abstraction level possible, and it includes the following DAs: ENTRY, SELECTION, DISPLAY, NOTIFICATION, CREATE INFO, or UPDATE INFO, among others.

Upon the information gathered with the NLP analyses, we developed a module that computes DAs out of SRL predicates and arguments [37]. The problem here is how to determine the possible DAs for a given predicate. We have approached this problem as a classification task. Basically, a classifier is a Machine- Learning (ML) technique that takes an instance and associates it with one or more disjoint labels representing conceptual classes. In our case, the classifier annotates each use case step with a set of likely DAs. Classifiers are able to forecast the label of a particular instance by learning from a subset of previously labeled examples. For learning our classifier, we used the Mulan toolkit[6] with a dataset containing sample use cases from several projects[7] in which the DAs were manually tagged. The textual sentences of each use case step were pre-processed (e.g., stop-word removal, stemming) and then translated as strings to a vector space model. Furthermore, the information about predicates and arguments produced by the SRL technique was also included as input of the training dataset. Both predicates and arguments enrich the knowledge of use case steps and thus improve the classifier performance. In the training phase, different techniques for text classification were explored, and we empirically choose a combination of hierarchical and multi-label classifiers with well-known vector machine algorithms [47]. As for the testing phase, the dataset was randomly partitioned, using k% of the labeled data for training the classifier and the remaining data for evaluating its performance.[8] The k parameter was varied between 20 and 50 %, and the partitions were built several times to average the results. We achieved acceptable results for $k = 40\,\%$ of the training data, with an average precision and recall of 80 %. Figure 8 sketches how *ReqAligner* labels DAs out of the constituents of a use case step, such as predicates and arguments. To this end, the DAs hierarchy and prediction model previously obtained are fed into the classifier so as to categorize new and unforeseen use case steps. In the example, predicates Pred1 and Pred2 are classified by our classifier as ENTRY → INPUT → I/O DAs, indicating that the two behaviors refer to typing in some information.

5 Alignment of use case sequences

Once the use case steps are labeled with domain actions, our technique is one step closer to identifying duplicate functionality. In this section, we focus on the discovery of duplicate behavior inside use cases, which corresponds to the second block of activities DISCOVERY OF FUNCTIONAL DUPLICATION IN USE CASES in Fig. 1. The application of text-

[6] Mulan: Available at http://mulan.sourceforge.net/.

[7] The projects and the use cases used for the training phase were developed by System Engineering students, in the context of a Software Development Methodologies course taught at UNICEN University in 2011 and 2012.

[8] Please note that this schema is more rigorous than a traditional k-fold cross-validation evaluation, in which the dataset is partitioned in k-1 subsets for training and just 1 subset for testing.

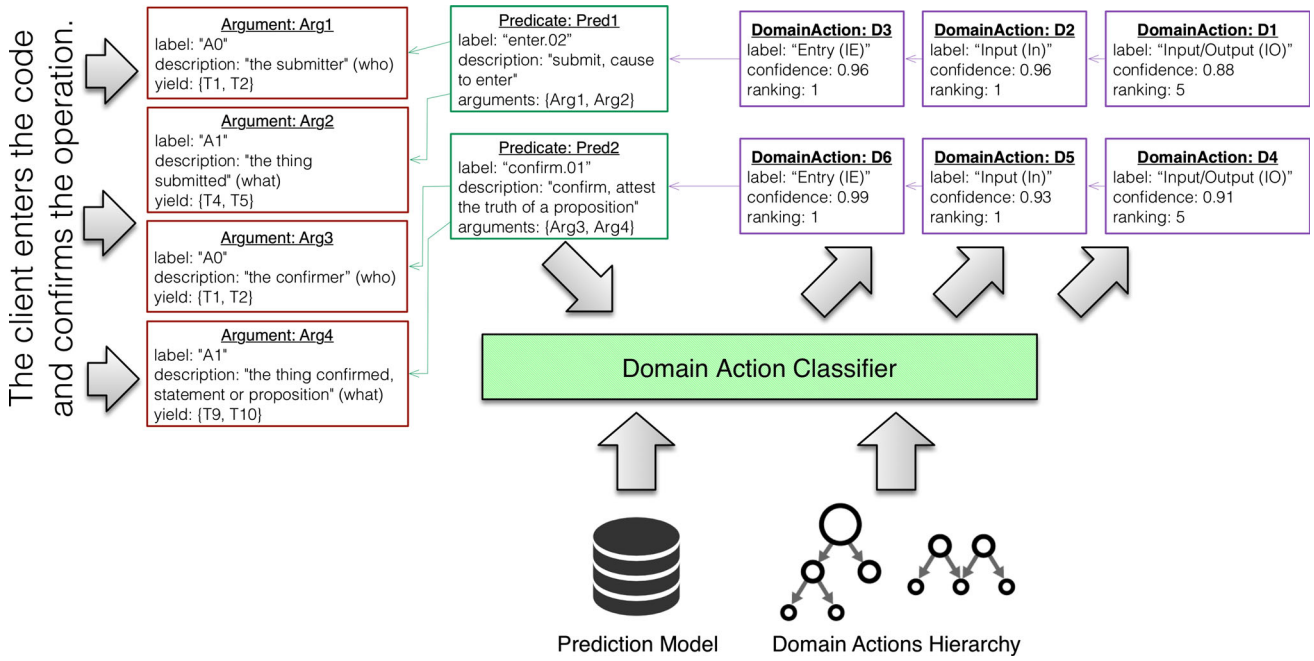


Fig. 8 Labeling domain actions out of a use case step

matching techniques can be handy in this regard. Unfortunately, text-matching techniques require to known the "pattern" to be searched in advance. This means to know which behavior to look for, or alternatively, to try out every possible combination. As the reader might have noticed, neither the performance nor the accuracy of this kind of techniques is good enough to properly address the problem [34], particularly in the case of a large number of textual descriptions to be processed.

We believe that there are two conditions that a technique should have to discover duplicate functionality in requirements. First, it should be capable of processing large amounts of textual requirements documents within reasonable time frames. Second, it should be flexible and configurable for working in particular domains and making adjustments in experimental environments. An appealing alternative to text-matching techniques that fulfills these conditions is *Sequence Alignment* (SA) techniques. The motivation behind SA techniques emerged in bioinformatic research [34], where researchers needed to represent and compare two or more DNA sequences for highlighting similarity zones or markers. DNA sequences are often very large, and in consequence, simple pattern-matching techniques are not capable of resolving this task. General SA techniques are designed to solve the optimal alignment of pairs of strings, i.e., to align one string with another in such a way the best similarity between the two strings is obtained. In order to align two strings, gaps might be inserted into the strings, and/or mismatched entries might be accepted. The application of SA techniques in fields other than bioinformatic have captured the attention of researchers worldwide during the last years, especially in the NLP area.

In this work, we apply a similar reasoning as the one used in biological research [34] and plagiarism detection [14,21] for finding sequences of semantically equivalent behaviors repeated across use case specifications. SA techniques fulfill the two conditions listed previously. SA techniques are flexible in the sense that two sequences may be marked as equals even if subtle differences exist between them, via the insertion or deletion of gaps. For instance, in use case specifications, two scenarios often share several interactions (i.e., use case steps), but one of the scenarios contains some interactions in between that make it harder to detect such duplication. SA techniques are also configurable in the sense that the similarity analysis between the sequences can be adjusted to perform better in a given environment. For example, it is not the same to compare humans' DNA samples than to compare monkeys' DNA [34]. The same reasoning can be extrapolated to use cases. Across dissimilar system domains there are certain functionalities that may relate differently one to another. For instance, while persisting information in a distributed application may very well be related to communicating through an external interface (because of the distributed nature of the system), in a embedded application this assertion is likely to be untrue.

5.1 Generation of use case sequences

Activity GENERATION OF USE CASE SEQUENCES is responsible for building a suitable representation of textual use cases

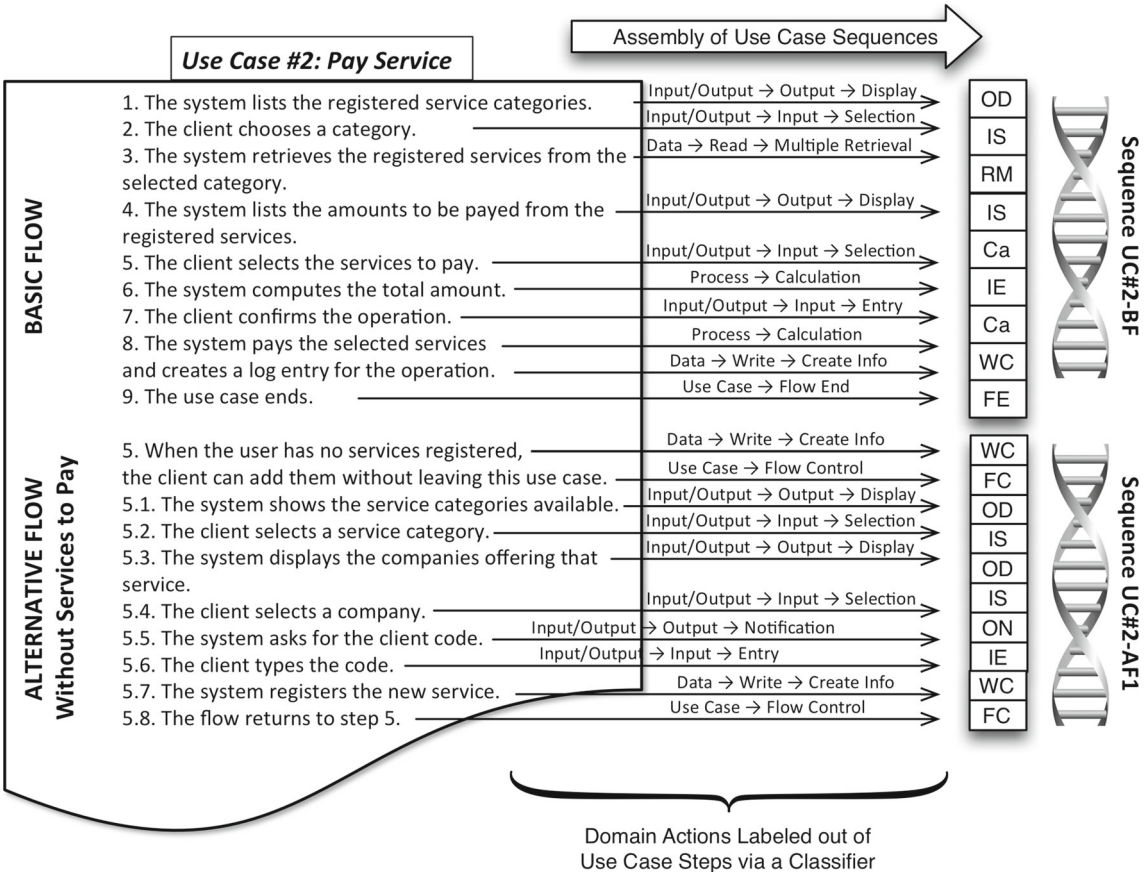


Fig. 9 Transforming a textual use case into sequences of domain actions

that can be fed into a sequence aligner. Furthermore, since we want the sequences to embody the intention of the specifications, we take advantage of the domain actions (DAs) computed earlier, assembling sequences of semantically enriched information from the use case steps. Specifically, we designated a particular character (or symbol) for each DA (see Fig. 9). In this way, sequences are represented as an ordered set of characters in which every character stands for a single DA. For the sake of simplicity, the representation generated by *ReqAligner* treats each flow of a use case as a flat structure, rather than using a more complex structure (e.g., a directed graph containing information of the basic and alternative flows altogether). This design choice makes the analysis of the sequences straightforward. The textual use cases are sequentially processed, one use case step at a time. Since use case steps generally have a single intention or behavior, semantic analyses will most of the times produce a single DA per sentence. However, there are other cases in which use case steps express more than one intention using *coordinating conjunction* connectors like "and" and "or" (such as in the example of Fig. 6). If so, the classifier is prepared to produce more than one DA per sentence, an the resulting sequence will end up having more characters than the total number of use case steps. Figure 9 exemplifies the transformation of the basic and alternative flows of a use case to a sequence of DAs. On the left, we have the use case steps of the two flows. The arrows capture the prediction made by our DA classifier, and we show it as a pathway in the hierarchy. For instance, step 3 is classified as a DATA → READ → MULTIPLE RETRIEVAL DA, which represents the top, middle and bottom classes of our hierarchy, respectively. At last, the blocks on the right represent the use case sequences. The text hold in each cells is an acronym of a particular DA. Only the leaves in the hierarchy of DAs are used in the assembly of sequences.

5.2 Alignment algorithm

The SA technique compares the characters of the generated sequences (representing domain actions) as well as the textual contents of the specifications. To do so, we have integrated the JAligner[9] library for Sequence Alignment into *ReqAligner*. This library implements Smith–Waterman's algorithm [46] and Gotoh's improvement [19] for local pairwise sequence alignment using a penalty model [34]. Local alignments are useful for comparing dissimilar sequences that are suspected to contain regions of similarity within a

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

larger sequence context. The Smith–Waterman algorithm is a general local alignment method based on dynamic programming. Pairwise sequence alignment methods are used to find the best-matching piecewise of two query sequences. Pairwise alignments can only be used between two sequences at a time, but they are efficient to compute and are often used by methods that do not require extreme precision. The drawback of algorithms based on dynamic programming is that they are computationally expensive when applied to large sequence alignments. Fortunately, use case sequences are generally limited in length, particularly after being converted to DAs. Furthermore, since we are only considering pairwise alignments, the use of dynamic-programming algorithms is feasible in the context of our work.

Basically, for computing the similarities between two sequences A and B, the Smith–Waterman's algorithm uses a scoring system that includes a substitution matrix S (also referred to as similarity matrix) and a gap-scoring scheme G. Scores for aligned characters are specified by the substitution matrix, where $S(a_i, b_j)$ is the similarity of characters $a_i \in A$ and $b_j \in B$. The gaps are computed with an affine gap penalty schema, where the handicap is a function of the gap length. To search the optimal local alignment, a matrix H (for pairwise alignments) is built. The entry in row i and column j is denoted here by $H(i, j)$. There is one column for each character in sequence A, and one row for each character in sequence B.

$$H(i,0) = 0, 0 \le i \le m$$
$$H(0,j) = 0, 0 \le j \le n$$
$$H(i,j) = \max \begin{cases} 0 & \\ H(i-1, j-1) + S(a_i, b_i) & \leftarrow \begin{matrix} Match/ \\ Mismatch \end{matrix} \\ H(i-1, j) - G(k) & \leftarrow Deletion \\ H(i, j-1) - G(k) & \leftarrow Insertion \end{cases}$$

where:

- $A = \{a_1, \ldots, a_m\}, B = \{b_1, \ldots, b_n\}$ are sequences of characters of length m and n, respectively
- $G(k) = \begin{cases} G_o & \leftarrow k = 1 \\ (k-1) * G_e & \leftarrow k > 1 \end{cases}$ is the gap-scoring schema for a gap of length k
- G_o is the score for opening a gap and G_e is the score for extending a the gap.

Once H is computed, a dynamic-programming algorithm is used for backtracking the best local alignment. To compute the pairwise alignment that actually gives the optimal score, the entry with maximum score in H is used as the starting point. From this place, the algorithm compares its value with the three possible sources (*match/mismatch*, *insertion*, and *deletion*) to see where it came from. If it is *match*, then a_i and b_j are aligned, if it is *deletion*, then a_i is aligned with a gap, and if it is *insert*, then b_j is aligned with a gap. This comparison is made incrementally, navigating the matrix from right to left and from bottom to top, until a cell with a score of 0 is found. In general, if there are more than one starting point, the algorithm applies an heuristic that leads to one of the many alternative optimal alignments.

The substitution matrix defines the similarity between two given characters. In our technique, the values of the matrix are expected to tell whether two use case steps (specifically, their DAs) express the same behavior. To determine the matrix values, we studied requirements specifications modeled with use cases from a variety of software domains, such as business (e-commerce), financial, device-sensing and content management systems. We assessed the analysts' intentions at the time of writing a specific functionality and how the predefined DAs related to those intentions. Based on this knowledge, we estimated the values for the substitution matrix depicted in Table 1. Matching DAs (on the diagonal) received a maximum value of 5.0, whereas mismatching DAs received similarity values ranging from 4.0 to 1.0. Completely dissimilar DAs were assigned to a score of 0.0. For example, the

Table 1 Substitution matrix in the use cases domain

	IE	IS	OD	ON	RS	RM	WC	WU	WD	CI	CE	Ve	Ca	FB	FE	FC
IE	5.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
IS	3.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
OD	0.0	0.0	5.0	3.0	2.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ON	0.0	0.0	3.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RS	0.0	0.0	2.0	0.0	5.0	4.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0
RM	0.0	0.0	2.0	0.0	4.0	5.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0
WC	0.0	0.0	0.0	0.0	0.0	0.0	5.0	3.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0
WU	0.0	0.0	0.0	0.0	0.0	0.0	3.0	5.0	2.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0
WD	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	5.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0
CI	0.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0	1.0	5.0	2.0	0.0	0.0	0.0	0.0	0.0
CE	0.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0	1.0	2.0	5.0	0.0	0.0	0.0	0.0	0.0
Ve	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	2.0	0.0	0.0	0.0
Ca	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	5.0	0.0	0.0	0.0
FB	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	0.0	0.0
FE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	0.0
FC	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0

Key
IE = entry, IS = selection, OD = display, ON = notification
RS = single Retrieval, RM = multiple Retrieval, WC = create info, WU = update info
WD = Delete Info, CI = internal interaction, CE = external Interaction, Ve = Verification
Ca = calculation, FB = flow begin, FE = flow end, FC = flow control

use case steps labeled with RS and RM DAs (which model single data retrieval and multiple data retrieval behaviors, respectively) have a similarity of 4.0 (as marked in Table 1) because they refer to the same behavior (retrieving some information), but are not exactly the same (one expresses a single information retrieval while the other expresses a bulk information retrieval). Other DAs being not similar at all, like RS and CA (that refers to some particular computation), have a similarity score of 0.0.

The gap-scoring schema considers the possibility of inserting null characters or deleting characters from both of the sequences under matching for maximizing the similarities between the use case steps (i.e., steps which could not be aligned at all). This is where the penalization schema comes into play. If there is a gap between the use case sequences, *ReqAligner* can continue aligning their use case steps at the expense of penalizing the presence of such gap. JAligner defines two parameters in the penalizing schema: a gap opening (G_o) and a gap extent (G_e). Since we wanted to penalize the small gaps but also permit the presence of larger gaps, we defined a large G_o penalty followed by a G_e penalty that is smaller than G_o. The actual values of our gap-scoring schema are $G_o = 5$ and $G_e = 2$. In this way, the algorithm will penalize several small gaps by the same extent as one large gap. There are no distinctions among DAs enclosed by a gap. That is, every type of DA is treated equally when computing the gaps. In the domain of use cases, we believe that an alignment similarity over 15 is an indicator of duplicate functionality. We chose 15 as threshold because there has to be at least 4 use case steps being semantically equivalent in the two sequences.

In order to understand how the SA algorithm works on use cases, let us consider a practical example. Figure 10 shows two scenarios from use cases UC#1 and UC#2, which were already processed by the classifier and transformed to DA sequences. *ReqAligner* explores the many possible alignments between both sequences and analyzes if there is a relevant subchain in any of the alignments that maximizes their similarity. For instance, Fig. 10 also sketches a possible alignment between the use cases. The sequence aligner then computes the pairwise similarity for each of the characters (that is to say, the DAs) of the sequences, as depicted in Table 2a. After obtaining the similarity from the substitution matrix, the algorithm proceeds to compute the H matrix. For the sake of simplicity in the construction of H, we used gap extension values of $G_o = G_e = 5$ in our example. Table 2b shows the resulting matrix. The algorithm searches the highest scoring cell in H, which in this example is 30. Since this value is over 15, it is considered a candidate duplication of functionality. However, there are more than one cell in the matrix with this value. The selection of one of these cells is performed by an heuristic implemented in JAligner. Let us assume that the algorithm chooses the cell marked with a circle in Table 2b at $H(10, 9)$ as the starting point. If so, the algorithm then backtracks the optimal local alignment moving incrementally from the starting point to a cell with a score of 0 (see the path of arrows drawn on the Table). The algorithm assures that the chain obtained is the best local alignment between the use cases (or at least, one of many optimal alignments). Finally, observing the arrow directions of Table 2b, the algorithm can determine whether the alignment inserted an empty space or deleted some character from the sequences. As a result of this observation, a visual representation of the local alignment found is returned (marked with bold characters and underlined):

Use Case #1 =
—·—·**OD**·**IS**·**OD**·**IS**·**ON**·**IE**·**IE**·**WC**·—

Use Case #2 =
WC·FC·**OD**·**IS**·**OD**·**IS**·**ON**·**IE**·—·**WC**· FC

Although the strategy of reducing use case steps to DAs works very well and makes the application of sequence alignment straightforward, our technique cannot ignore the terms

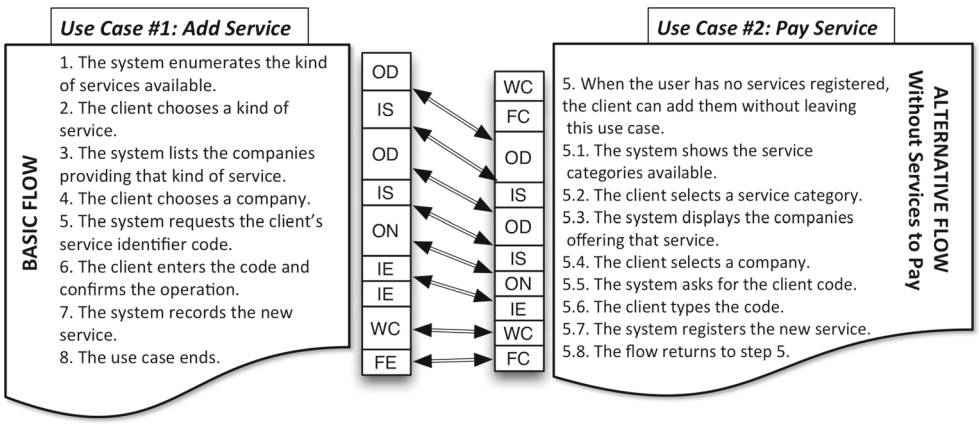


Fig. 10 Possible alignment of use case sequences

Table 2 Local alignment of use case sequences with Waterman–Smith's algorithm

(a) Similarity Computation for Pairwise Alignment

	Δ	WC	FC	OD	IS	OD	IS	ON	IE	WC	FC
Δ	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
OD	0.0	0.0	0.0	**5.0**	0.0	**5.0**	0.0	**3.0**	0.0	0.0	0.0
IS	0.0	0.0	0.0	0.0	**5.0**	0.0	**5.0**	0.0	**3.0**	0.0	0.0
OD	0.0	0.0	0.0	**5.0**	0.0	**5.0**	0.0	**3.0**	0.0	0.0	0.0
IS	0.0	0.0	0.0	0.0	**5.0**	0.0	5.0	0.0	**3.0**	0.0	0.0
ON	0.0	0.0	0.0	**3.0**	0.0	**3.0**	0.0	**5.0**	0.0	0.0	0.0
IE	0.0	0.0	0.0	0.0	**3.0**	0.0	**3.0**	0.0	**5.0**	0.0	0.0
IE	0.0	0.0	0.0	0.0	**3.0**	0.0	**3.0**	0.0	**5.0**	0.0	0.0
WC	0.0	**5.0**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**5.0**	0.0
FE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

(b) Computing the Optimal Alignment of the Sequences from H

	Δ	WC	FC	OD	IS	OD	IS	ON	IE	WC	FC
Δ	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
OD	0.0	0.0	0.0	**5.0**	0.0	5.0	0.0	3.0	0.0	0.0	0.0
IS	0.0	0.0	0.0	0.0	**10.0**	5.0	10.0	5.0	6.0	1.0	0.0
OD	0.0	0.0	0.0	5.0	5.0	**15.0**	10.0	13.0	8.0	6.0	1.0
IS	0.0	0.0	0.0	0.0	10.0	10.0	**20.0**	15.0	16.0	11.0	6.0
ON	0.0	0.0	0.0	3.0	5.0	13.0	15.0	**25.0**	20.0	16.0	11.0
IE	0.0	0.0	0.0	0.0	6.0	8.0	16.0	20.0	**30.0**	25.0	20.0
IE	0.0	0.0	0.0	0.0	3.0	6.0	11.0	16.0	**25.0**	30.0	25.0
WC	0.0	5.0	0.0	0.0	0.0	3.0	6.0	11.0	20.0	**30.0**	30.0
FE	0.0	0.0	5.0	0.0	0.0	0.0	3.0	6.0	15.0	25.0	30.0

that constitute the sentences of use case steps. We need to be sure that the DAs actually represent the same behavior and operate over the same "things". To complement the abstraction introduced by DAs, we apply simple *Information Retrieval* techniques. We defined a *text similarity function* that counts the number of repeated terms between two use case steps and allow the technique to compare two use cases at a lexical level. To mitigate problems of synonymy and textual noise, we previously applied stop words and stemming techniques [29]. The text similarity function produces a number between 0.0 and 1.0, where 0.0 indicates a low lexical coincidence, and 1.0 indicates a high lexical coincidence between the use case steps. This function is integrated into the *ReqAligner* by changing the original similarity computation of the SA technique for a multiplication of the substitution matrix and the text function. The end result of this change is an attenuated version of the alignment values. Table 3 shows how the similarity values are re-computed in the optimal alignment previously introduced. This time, the best local alignment achieves a similarity of 22.0, a little below than the 30.0 obtained without the text function, but still enough to consider it a candidate duplication of functionality.

Table 3 Matching and computing scores with sequence alignment

Sequence UC#1-BF	Sequence UC#2-AF1	Matching result	Substitution matrix	Text function	S (a_i, b_j)
–	WC	Gap opening (deletion)	0.0	0.0	0.0
–	FC	Gap opening (deletion)	0.0	0.0	0.0
OD	OD	Matched	5.0	0.8	4.0
IS	IS	Matched	5.0	0.8	4.0
OD	OD	Matched	5.0	0.7	3.5
IS	IS	Matched	5.0	0.8	4.0
ON	ON	Matched	5.0	0.6	3.0
IE	IE	Matched	5.0	0.9	4.5
IE	–	Gap opening (insertion)	−5.0	–	−5.0
WC	WC	Matched	5.0	0.8	4.0
FE	FC	Mismatched	0.0	0.0	0.0
			30		22.0

6 Recommending use case relationships

The final activity is that of giving advice to analysts about the duplicate functionality just detected with the SA algorithm (see activity RECOMMENDATION OF USE CASE RELATIONSHIPS in Fig. 1). Along this line, UML is equipped with reuse mechanisms, such as *inclusion*, *extension*, and *generalization relationships* between use cases, which help to deal with duplication-related deficiencies via refactoring (of the use cases). Given the detailed information provided by the SA algorithm (e.g., it indicates the boundaries of the duplication), we have developed a rule-based heuristic as part of *ReqAligner* in order to recommend UML relationships to analysts, so that they can mend duplication-related deficiencies. In this regard, *ReqAligner* analyzes two properties of the matched sequences: *origin* and *coverage*. Origin refers

to the location from where a sequence was assembled, for example: basic flow, alternative flow, special requirements, among others. Coverage refers to the place occupied by a duplicate functionality "within their containing sequence". We mapped the different cases for this property to five alternatives, namely:

- At The Beginning: when duplicate functionality starts in the same place the containing sequence starts.
- At The End: when duplicate functionality ends in the same place the containing sequence ends.
- In The Middle: when duplicate functionality does neither starts nor ends in the same place the containing sequence starts and ends, respectively.
- Whole Sequence: when duplicate functionality starts and ends at the same places that the containing sequence.
- Both Beginning And End: when the subset of duplicate functionality starts and ends at the same places that the containing sequence, but there is a considerable gap in between introduced by the sequence aligner.

Upon the discovery of a (best) local alignment among two sequences, there are many possible combinations of origin and coverage. Thus, we defined several tables for generating useful recommendations. For example, Table 4a depicts the advice to be provided by the tool if some duplicate functionality is found among two basic flows. The rows and columns of the table indicate the variation of the coverage property, and the cells contain the advice given to the analysts under a given circumstance. We have created tables for other combination of origins, such as sequences coming from a basic flow and an alternative flow (see Table 4b), or from two alternative flows (see Table 4c). Other combinations among preconditions, posconditions, triggers and special requirements have not been explored yet. It is up to the analysts to decide whether a defect is correctly detected with *ReqAligner* and if they should follow the advice of the tool to mend that defect.

Let us explain the workings of the recommendation process, continuing with our motivating example of UC#1 and UC#2 introduced in previous sections. *ReqAligner* just discovered a duplicate functionality between two sequences coming from the basic flow of UC#1 and from an alternative flow of UC#2. For recommending an action, our technique looks at the properties of the sequences and the matching, that is, their origin and coverage. We already know where they came from, meaning that we should start looking at Table 4b, so let us analyze the kind of coverage obtained. In UC#1, the aligned sequence covers the whole basic flow. However, in UC#2 the technique almost covers the whole alternative flow. In order to disambiguate this difference, we apply the following rule: if more than 85 % of the use case steps are matched, then the coverage is considered to be "whole". This is indeed the case of our the example. Therefore, the advice of *ReqAligner* (highlighted in Table 4b) is to: (i) remove the alternative flow from UC#2, (ii) create an extension point in the basic flow of UC#2, and (iii) modify UC#1 by adding an extension of UC#2 at the extension point just created and then including a condition for the extension to take place. Let us assume the analyst performs this improvement, and the use case diagram of the ATM system is modified as seen in Fig. 5.

7 Evaluation

To investigate whether *ReqAligner* will satisfy the expectations of requirements analysts when applied to real-world projects, we conducted a series of experiments with several case studies. We hypothesize that *ReqAligner* should help to identify and improve a great deal of duplicate functionality deficiencies from a set of use case specifications, and furthermore take less time than what it would take to analysts. In the case studies, we identified three response variables that will help to corroborate the hypothesis. The first variable measures the time needed to identify duplicate functionality. The second variable measures the amount of correct duplicate functionalities recovered with our technique (without losing track of the omissions). The third variable measures the ability to produce good recommendations for improving the use cases.

Several pilot projects and their corresponding requirements specifications were analyzed for choosing case studies that were representative enough for generalizing the findings of this evaluation. As selection criteria, we took into consideration parameters such as application domain (i.e., type of project), programming paradigm and requirements capture method, or availability of the project artifacts in the community, among others. We aim at evaluating our hypothesis with projects that cover a wide range of software domains, which apply object-oriented modeling techniques and use cases for capturing requirements, and which are publicly available for other researchers to compare their studies with ours.

As a result, we ended up selecting 5 case studies, namely: DLibraCRM, MobileNews, WebJSARA [8], HWS [20], and CRS [4]. The first 3 case studies consisted of a set of use cases created as part of the Software Development Studio[10] (SDS) projects, and their sources are included in the master thesis of Ciemniewska and Jurkiewicz [8]. These 3 case studies comprise dissimilar software domains, including: (i) a web-based system for cataloging and selling book-related goods online (DLibraCRM), (ii) a news feed system for delivering the latest bulletins to mobile devices

[10] Software Development Studio (SDS) Development Server—http://sds.cs.put.poznan.pl.

Table 4 Recommendation tables for analyzing the sequences found by *ReqAligner*

(a) Advice when both Sequences are Originated from Basic Flows

Origin: Sequence#1 and Sequence#2 come both from Basic Flows (of UC#1 and UC#2)		Coverage of Sequence#2 (from UC#2)				
		Beginning of Sequence#2	Ending of Sequence#2	Middle of Sequence#2	Whole Sequence#2	Beginning and Ending of Sequence#2 (with a gap in the middle)
Coverage of Sequence#1 (from UC#1)	Beginning of Sequence#1	1. Create a new use case UC#3 2. Copy the duplicated contents to UC#3 3. Remove the duplicated behavior from UC#1 and UC#2 3. Use an inclusion relation referencing UC#3 from both UC#1 and UC#2			1. Use an inclusion relation referencing UC#2 from UC#1	N/A
	Ending of Sequence#1					
	Middle of Sequence#1					
	Whole Sequence#1	1. Use an inclusion relation referencing UC#1 from UC#2			1. Remove either UC#1 or UC#2, since they are alike	
	Beginning and Ending of Sequence#1 (with a gap in the middle)	N/A				1. Create a new use case UC#3 2. Move the duplicated contents at the beginning and the ending of UC#1 an UC#2 to UC#3 3. Add two generalization relations between UC#3 and UC#1 and between UC#3 and UC#2

(b) Advice when One Sequence is Originated from a Basic Flow and the Second from an Alternative Flow

Origin: Sequence#1 comes from the Basic Flow (of UC#1) and Sequence#2 comes from an Alternative Flow (of UC#2)		Coverage of Sequence#2 (from UC#2)				
		Beginning of Sequence#2	Ending of Sequence#2	Middle of Sequence#2	Whole Sequence#2	Beginning and Ending of Sequence#2 (with a gap in the middle)
Coverage of Sequence#1 (from UC#1)	Beginning of Sequence#1	N/A (We can't tell whether the duplicated functionality in Sequence#2 is strong enough by itself to be refactored into a new use case)			1. Create a new use case UC#3 2. Copy the duplicated contents to UC#3 3. Remove the alternative flow from UC#2 5. Create an extension point in the basic flow of UC#2 6. Add an extension relation from UC#3 to UC#2	N/A
	Ending of Sequence#1					
	Middle of Sequence#1					
	Whole Sequence#1	1. Create a new use case UC#3 2. Copy the contents of Sequence#2 to UC#3 3. Remove the alternative flow from UC#2 4. Remove the duplicated contents from UC#3 5. Add an inclusion relation from UC#3 to UC#1 6. Create an extension point in the basic flow of UC#2 7. Add an extension relation from UC#3 to UC#2			1. Remove the alternative flow from UC#2 2. Create an extension point in the basic flow of UC#2 3. Add an extension relation from UC#1 to UC#2	
	Beginning and Ending of Sequence#1 (with a gap in the middle)	N/A				N/A

(c) Advice when both Sequences are Originated from Alternative Flows

Origin: Sequence#1 and Sequence#2 come both from Alternative Flows (of UC#1 and UC#2)		Coverage of Sequence#2 (from UC#2)				
		Beginning of Sequence#2	Ending of Sequence#2	Middle of Sequence#2	Whole Sequence#2	Beginning and Ending of Sequence#2 (with a gap in the middle)
Coverage of Sequence#1 (from UC#1)	Beginning of Sequence#1	N/A (We can't tell whether the duplicate functionality in both Sequence #1 and Sequence#2 is strong enough by itself to be refactored into a new use case)			1. Create a new use case UC#3 2. Copy all the steps of Sequence#2 to the basic flow of UC#3 3. Remove alternative flows from UC#1 and UC#2 4. Create an extension point in the basic flow of UC#1 and UC#2 5. Add an extension relation from UC#3 to UC#1 and from UC#3 to UC#2	N/A
	Ending of Sequence#1					
	Middle of Sequence#1					
	Whole Sequence#1	1. Create a new use case UC#3 2. Copy all the steps of Sequence#1 to the basic flow of UC#3 3. Remove alternative flows from UC#1 and UC#2 4. Create an extension point in the basic flow of UC#1 and UC#2 5. Add an extension relation from UC#3 to UC#1 and from UC#3 to UC#2			1. Create a new use case UC#3 2. Copy all the steps of either Sequence#1 or Sequence#2 to the basic flow of UC#3 3. Remove alternative flows from UC#1 and UC#2. 4. Create an extension point in the basic flow of UC#1 and UC#2 5. Add an extension relation from UC#3 to UC#1 and from UC#3 to UC#2	
	Beginning and Ending of Sequence#1 (with a gap in the middle)	N/A				N/A

Table 5 Characteristics of the case studies

	DLIBRACRM	MOBILENEWS	WEBJSARA	CRS	HWS	Total
Kind of development	Academia	Academia	Academia	Industry	Government	
Domain	Online bookstore	Mobile news delivery	Content management system	University course management	Health care system	
Number of use cases	16	15	29	8	9	77
Textual pages	14	10	22	20	19	85
Number of words	1,360	1,180	1,990	3,670	3,280	11,480

Table 6 Summary of the baseline solution for the case studies

	DLIBRACRM	MOBILENEWS	WEBJSARA	CRS	HWS	Total
Duplicate functionality	9	6	18	3	13	49
↓	↓	↓	↓	↓	↓	↓
Generate inclusion	7	4	11	0	0	22
Generate extension	2	2	2	3	13	22
Generate generalization	0	0	5	0	0	5

(MOBILENEWS), and (iii) an information management system with publishing, searching, uploading and downloading capabilities (WEBJSARA). The fourth case study is the Course Registration System (CRS) [4], a distributed system to be used for managing an university courses and subscriptions. The fifth case study is the Health Watcher System (HWS) [20], a web-based system that serves as a mediator between citizens and the municipal government. Table 5 summarizes the characteristics of each case study.

The analysis with case studies is comparative in nature, and hence, we need to contrast the results obtained with our technique with another one. To keep experimental biases at a minimal level, a valid basis for assessing the results of the case studies must be identified in advance. Unfortunately, the identification of duplicate functionality for the case studies selected for this evaluation has not been (to the best of our knowledge) discussed neither at academic nor industrial forums. Therefore, we had to carefully define an exercise that would allow us obtain an objective baseline solution. For the sake of comparison, we entrusted the analysis of the 5 case studies to four senior functional analysts.[11] These analysts graduated from UNICEN University and were 35–40 years old. They have been working in industry for the past 10 years. Particularly, they are knowledgeable of RE practices (e.g., the unified process or agile principles) and have been working with use cases for at least 5 years. They were in charge of manually inspecting the use cases, identifying duplicate behaviors across the use case specifications and deciding on the mending procedures for removing such duplication. The analysts were allotted 8 h to perform this task, each one working separately from the rest. During this time, they had to examine all five case studies in a random order (mitigating learning biases). At last, we arranged a meeting with the analysts, in which they exposed their findings and we acted as facilitators of the discussion. After some discussions, the analysts reached a consensus about their findings, and we consolidated the knowledge they acquired from the projects and established a single baseline solution from each case study. The number of duplicate functionality reported for the case studies was larger than we expected. Table 6 summarizes these numbers, and also shows the analysts' strategies to improve the use case specifications (e.g., generate inclusion, generate extension, generate generalization).

Regarding the interpretation of the response variables, especially the second and third ones, we chose to apply measures from the Information Retrieval (IR) field like *Precision* and *Recall* [29]. *Precision* gauges the rate of correct suggestions made by the technique (**tp**: *true positives*) in contrast to the amount of incorrect suggestions (**fp**: *false positives*). False positives can be originated either from a wrongly detected duplication of functionality or from a wrong recommendation given to deal with a duplication correctly detected by the technique. *Recall* gauges the rate of correct suggestions made by the technique (**tp**) in contrast to the amount of missed suggestions (**fn**: *false negatives*) from all the duplicate functionalities present in the case study. *Precision* and *Recall* are computed as follows:

[11] To reduce confounding factors, we ensured these analysts had a college degree in Software Engineering with a solid academic background. Moreover, the final meeting between the four of them allowed us to manage the different learning curves which might have affected their findings.

$$Precision = \frac{tp}{tp + fp}$$

$$Recall = \frac{tp}{tp + fn}$$

The analysis of the experiments is organized along four research questions, which are addressed in the subsections below.

7.1 Can ReqAligner identify duplicate behaviors in due time and form?

The two response variables under consideration here are: (i) the time for executing the technique, and (ii) the identification of duplicate functionality. We ran *ReqAligner* with the 5 case studies and analyzed the functionality presumably repeated among the use cases, as outputted by our technique. For the first variable, we recorded the time lapses needed for completing the individual activities of the technique, namely: *Basic NLP*, *Semantic Analysis of Use Cases with Domain Actions*, *Generation and Alignment of Use Case Sequences*, and *Recommendation of Use Case Relationships*. When it comes to the second variable, we compared the items of duplicate functionality identified with *ReqAligner* against those defined in the baseline solution. The comparison allowed us to obtain basic figures for *tp*, *fp* and *fn*, which then led to more indicative measures like *Precision* and *Recall*. Since there is no intervention whatsoever of the analysts in the outputs, we did not have to take any consideration for confounding factors that may affect the results.

7.1.1 Question #1: How much time does ReqAligner takes to process each case study?

The execution of *ReqAligner* in the case studies demanded a very short time. Table 7 shows the performance (measured in seconds) taken for each individual activity executed. We observed that the whole-automated analysis of the use cases only required at most a few minutes to complete. The biggest portion of the total execution time was attributed to the basic NLP analysis, and particularly to the *Semantic Role Labeling* task. Because this activity is tied up to the number of sentences, its complexity is approximately linear in the length of the documents. Regarding the SA activities, note that their execution just took some seconds. This result is promising, mainly because SA consumes only a fraction of the time needed for conducting the basic NLP analysis. Furthermore, since the SA technique works on a pairwise basis, its complexity is defined by the number of use cases under analysis, making it feasible to apply *ReqAligner* in larger projects without significant time overheads.

Let us make an educated guess of *ReqAligner* performance in a more stressful scenario. Cockburn argued that a "good" use case should have around 3–10 steps in main/basic course and short alternative flows. A recent study of 16 industry projects revealed that 66 % of industry requirements usually have use cases with main courses of 3–9 steps. Moreover, this investigation showed that 75 % of those use cases had more than one alternative flow of 1–4 steps each [2]. So, let us assume a typical use case scenario (i.e., a basic course) with an average of 10 use case steps, and also including at least two alternative flows with fewer steps (5 steps). Therefore, we can say that a typical use case can be decomposed into ∼3 use case sequences. Let us assume then that such length remains constant as we move from small- to large-sized systems. In addition, large systems often demand around 80 or more use cases for capturing the whole set of functionality. Upon these observations, we would have ∼240 sequences for a large system: 80 of ∼10 steps and 160 of ∼5 steps, in contrast to the largest case study of our evaluation (WEBSJARA) that had ∼100 sequences derived from 29 use cases. Comparing sequences in a pairwise fashion implies that our technique needs to match the combinations of all possible pairs of sequences: $\binom{240}{2} = \frac{240!}{2!\cdot 238!} = 28.680$. This means that the number of sequences to align grows exponentially as the number of use cases increases.

The time complexity of Smith–Waterman's local alignment algorithm is $\mathcal{O}(n \cdot m)$, where n and m are the lengths of the first and second sequences, respectively. The space complexity is also $\mathcal{O}(n \cdot m)$. JAligner includes two optimizations to improve the complexity of the original algorithm. First, it reduces the space complexity from $\mathcal{O}(n \cdot m)$ to $\mathcal{O}(\max(n, m))$ by using single-dimensional arrays of size $\max(n, m)$ instead of the original matrix of $n \cdot m$. Second, it speeds up the traceback of the algorithm by mapping the matrix to a single-dimensional array. Because the complexity of the algorithm

Table 7 Execution time for executing *ReqAligner* (measured in seconds)

	DLIBRACRM	MOBILENEWS	WEBJSARA	CRS	HWS	Total
Basic NLP	32	43	58	55	74	262
Semantic analysis	6	5	5	4	6	26
Sequence generation	1	1	1	1	1	5
Sequence alignment	7	5	15	9	10	46
Advice generation	1	1	1	1	1	5
Total	47	55	80	70	92	344

is affected mainly by the length of the sequences under comparison, and not by the number of combinations we try out, we can ensure a good performance of our technique that will increase linearly with the number of use cases. That is, the worst time/space scenario with use cases is aligning two main courses of ∼10 steps length each, which is fairly quick to compute with JAligner.

Having in mind that the generation of the baseline solution demanded around 10 work hours of four senior analysts, we are satisfied with the performance of our technique. We believe that *ReqAligner* can provide analysts a quickly and rather accurate view of duplicate functionality in use cases.

7.1.2 *Question #2: How many of the duplicate functionalities were correctly identified and how many were missed?*

Table 8 summarizes the data collected from the execution of *ReqAligner* on each of the case studies by comparing the output of our techniques against the corresponding baselines. The *Precision* and *Recall* measures are shown in the rows at the bottom of Table 8. In the rightmost column, we combined the data from the 5 case studies, enabling an analysis of the overall performance of the technique. By observing the latter column alone, we can conclude that the quality of detection achieved was encouraging, because *ReqAligner* identified many duplications existing in the use cases (86 % recall), without making significant mistakes in the process (63 % precision).

When it comes to the individual *Recall* results, the technique identified the entire set of deficiencies in 3 of the case studies (DLIBRACRM, CRS and HWS). This result is reflected by a perfect recall score in the table. In the remaining 2 case studies, MOBILENEWS and WEBJSARA, the tool obtained ∼80 and 65 % recall, respectively. WEBJSARA results had a considerable number of false negatives, attributed to many duplicate functionalities that were not uncovered. A possible explanation for this effect is that some functionalities identified as duplicate in the baseline were very small (in textual length). Because our technique is best suited for spotting sizable chunks of use case steps repeated across the use cases, the identification of these small functionalities did not work well.

Regarding the *Precision* of the case studies, our technique obtained acceptable results. In MOBILENEWS and CRS, *ReqAligner* achieved a precision above 75 %. This value means that from the complete set of duplicate functionality identified, only a quarter corresponded to false positives. In this context, we argue that analysts can start working on this initial set (provided by the tool) and remove spurious instances with little effort. The results of the remaining 3 case studies stood a little behind. The precision in DLIBRACRM, WEBJSARA and HWS dropped to a ∼60 % precision. A more detailed analysis revealed that this precision loss was caused by a flawed detection of duplicate functionality by our technique. Even so, when we inspected these mistakes, we noticed that some use cases involved shared lexical (i.e., the terms) and semantic (i.e., the domain actions) aspects. Consequently, the detection problem can be (at least, partially) attributed to the way analysts specify use cases, which generally replicate the writing style of use case interactions. Unfortunately, *ReqAligner* can have a hard time telling the difference between real and subtle variations of use cases. For instance, an anomalous situation we encountered in the experiment was that of CRUD (*Create-Retrieve-Update-Delete*) use cases [10]. In DLIBRACRM, WEBJSARA and HWS, several of the duplications identified involved use cases including CRUD interactions. The specifications of such use cases were written using the same terms and organized using the same syntactical structure, and consequently ended up being identified as potential duplication deficiencies. We believe that this phenomenon might be ascribed to an abuse of copy/paste in text processors. Nonetheless, we were compelled to treat these findings as mistakes (i.e., false positives) in our analysis, what ultimately impacted on the *Precision* of the case studies.

Overall, the automated identification of duplicate functionality produced reasonable results. For an assistive RE tool such as *ReqAligner*, we think that an acceptable precision and a high recall are desirable. That is, we would like *ReqAligner* to find as many deficiencies in the use cases as possible, even at the cost of detecting a few incorrect ones, because the analyst will ultimately assess the outputs produced by the tool. Thus, in the context of this evaluation, we favor recall over precision, as it is often easier for the analysts to remove incorrect suggestions than to find all the

Table 8 Analysis of the duplicate functionality recovered with *ReqAligner*

	DLIBRACRM	MOBILENEWS	WEBJSARA	CRS	HWS	Overall
TP	9	5	12	3	13	42
FP	6	1	10	1	7	25
FN	0	1	6	0	0	7
Precision	0.60	0.83	0.55	0.75	0.65	0.63
Recall	1.00	0.83	0.67	1.00	1.00	0.86

Table 9 Analysis of the recommendations generated with *ReqAligner*

	DLIBRACRM	MOBILENEWS	WEBJSARA	CRS	HWS	Overall
TP	9	5	9	3	13	39
FP	0	0	3	0	0	3
Precision	1.00	1.00	0.75	1.00	1.00	0.93

duplications via a manual inspection. This line of thinking has also been followed in other evaluations of automated RE tools [11].

7.1.3 *Question #3: Were the duplicate functionalities correctly delimited?*

Finally, we also assessed the quality of the alignments identified with our technique. This is to say, how well-formed the alignments identified as duplicate were when contrasted with the analysts' opinions. To do so, we looked at the boundaries of the alignments and checked whether they matched the duplicate functionalities defined by the baseline solution. We found that none of the alignments outputted by the technique included fewer use case steps than those of the baseline. However, we did found that many alignments included more use case steps than those determined by experienced analysts. In general, the boundaries were exceeded by one or two additional use case steps. There were two factors that contributed to this fallback. First, some mistakes made when labeling the use case steps with domain actions, which caused matching mistakes. While in general our classifier works well, there were cases in which the use case steps were wrongly classified. Second, as we mentioned earlier, there were also cases in which the similarities between the terms of some use case steps drove our technique to produce faulty alignments.

7.2 Question #4: Is ReqAligner advice helpful to the analysts?

This question tries to answer whether our technique can provide analysts with good recommendations for fixing instances of duplicate functionality discovered in the specifications. Here, the focus is on the third response variable, which takes the *ReqAligner* advice for duplications correctly detected and compares them with the opinion provided by the expert analysts. This comparison serves us to determine: (i) the number of *tp*, for those cases in which the advice effectively guided the users to perform the same resolution chosen by expert analysts; and (ii) the number of *fp*, for those cases in which the advice given by the tool was wrong. Table 9 summarizes the collected values, including the computation of *Precision* and *Recall*. Note that the columns for each case study must add up to the number of true positives from Table 8.

The results obtained with the rule-based heuristics implemented in *ReqAligner* were very precise, making only a few mistakes in the process. The overall precision was quite high (93 % precision), exceeding our expectations. Four out of 5 case studies obtained the maximum precision, which means that, for those duplicate behaviors correctly identified, our technique was able to provide the right advice every time. In WEBJSARA, we observed that there were three instances of inaccurate recommendations. A more detailed analysis showed that these recommendations were due to poorly identified boundaries in 3 pairs of use cases. These use cases contained a duplication of functionality that could be easily fixed using a generalization relationship. However, since our technique failed to notice the large gap among the sequences where the behavior was specialized, the tool (incorrectly) recommended an inclusion relationship to be applied.

7.3 Lessons learned

After experimenting with five real case studies, we had valuable insights on the working of *ReqAligner*. During the evaluation, we noticed that comparing use case steps by only looking at their domain actions was not always enough. For example, in the analysis of CRUD-related use cases, the tool made some mistakes and wrongly suggested duplication of functionality. We believe that if the approach would have considered the objects associated to a particular action (i.e., which kind of entity is the system creating/modifying/removing), some false positives could have been avoided. Moreover, if the approach would have also taken into consideration the actor/s participating in the use case steps, the boundaries of duplicate functionality might have been better detected.

We also observed that the approach fails to detect duplicate functionality between sequences that omit some "irrelevant" use case steps. We say irrelevant, because there are some use case steps that are optional from a functional perspective (for instance, the confirmation of an operation). Although *ReqAligner* is prepared to handle gaps between two sequences, we noticed that occasionally the tool fails to spot duplicate functionality if there are many single and scarce irrelevant steps missing. An interesting way of addressing this problem is to weight the gap-scoring mechanism by analyzing what type of action is missing. This means paying more attention to gaps of "relevant" actions and less attention to gaps of "irrelevant" actions.

Furthermore, despite the evaluation covered five real case studies, their specifications were small in comparison to large-scale developments. We exercised our tool with a sixth case study that comprised 22 use cases and approximately 38 textual pages. Unfortunately, we did not reported this experiment due to the lack of a reference baseline for computing measurements. Still, we observed some peculiar responses of the tool in this larger case study. First, we were surprised by the speed of *ReqAligner*, which took approximately 20 seconds to align all the sequences and produced a reasonable output. Second, our tool suggested approximately 50 candidate deficiencies/refactorings related to duplicate functionality. Nonetheless, since many of the deficiencies detected involved the duplication of functionality among three, four, or even five use cases, the pairwise analysis of the tool produced a larger number of recommendations than the ones really needed.

7.4 Threats to validity

The purpose of *ReqAligner* is to identify duplicate behavior in use case specifications in an automated fashion and to help analysts to improve the quality of the requirements. Despite a careful development of the experiments with case studies, there were threats that might compromise the results obtained.

Regarding construct validity, we believe that individually identifying the presence of duplicate behaviors is a good way to assess the problem at hand. The adopted IR metrics (like other related work [8]), such as *precision* and *recall,* are accurate vehicles for measuring the performance of our technique. Nonetheless, these metrics have no consideration for the relevance of the duplication found, that is, they are unable to tell whether a duplicate behavior contributes to the (poor) quality of the use cases.

Regarding internal validity, we identified three threats. First, the baseline solution used to compare the outputs of the technique may be biased. Determining whether the behaviors of two use case specifications are duplicate is a rather subjective activity and depends on the analysts' expertise and interpretation of the system requirements. While one analyst may think that a given duplication of behavior is acceptable and does not reduce the clarity of the specifications, another analyst may choose to apply a UML reuse mechanism and refactor such duplication. The exercise of developing a baseline solution was designed in such a way the duplicate behaviors were agreed by a group of experienced analysts, but still there may be cases in which a single analyst's opinion influenced on the rest of the group. Second, although the analysts that defined the baseline solutions were carefully selected (according to their expertise), there still may be a bias and maybe more experienced analysts could have produced baselines of better quality. Third, we have to consider that duplicate behavior is identified (by our technique) on a pairwise basis and the possible defects are attended individually (i.e., one by one). This means that the order in which the duplicate behavior was dealt with might have an effect on the results, hindering the detection of some behaviors or even introducing new duplicate behaviors. If we had applied instead a multiple-sequence alignment, this threat could had been mitigated, although at the expense of hurting the performance of our tool.

When it comes to external validity, we tested *ReqAligner* with small and medium-sized systems coming from publicly available sources, but our technique should be applicable to other software projects as well. The performance obtained by our technique in the experiments and the complexity of the sequence alignment algorithm led us to believe that *ReqAligner* should not have problems to deal with larger requirements specifications. However, we have to consider three threats to external validity. First, although the classifier of domain actions was trained to work on virtually any use case specification,[12] it may need to be adapted (by adding or removing classes) and re-trained (e.g., using another dataset) to operate on specialized software domains. Second, the current substitution matrix of the SA algorithm may need to be customized to accommodate particular software domains. Third and last, the reported case studies might not be large enough to make conclusions for bigger systems, and further experimentation is needed on systems with a large number of use cases (100 or more). For instance, we should have to assess if the number of false positives detected is still kept to an acceptable level.

8 Related work

Several researchers have studied the underlying causes that reduce the clarity of requirements specifications and makes them difficult to understand and/or communicate. In general, these works can be organized into two groups: (i) those that concentrate on the search of patterns and evidences of defects in requirements models and their textual specifications, and (ii) those that deal with the improvement of faulty specifications. While the former group characterizes the defects and provides heuristics to find them, the latter group is mainly focused on the provision of mechanisms for fixing faulty specifications and thus get requirements of better "quality". In the context of our work, "quality" refers to requirements that are written clearly and unambiguously, at the same time that they are essential (i.e., non-redundant) [22].

Regarding the first group, only a few works have addressed the automation of defect discovery by means of computer-

[12] Because it exploits peculiarities in the domain of use cases, especially of textual use case specifications.

supported techniques. Femmer et al. studied the application of light-weight static analyses to make instant checks on textual requirements and detect bad smells [18]. These smells include 8 types of deficiencies. The most relevant ones are: ambiguous adverbs and adjectives, vague pronouns, subjective language, and comparative phrases. The authors implemented a prototype tool for smell detection by using NLP techniques. However, the granularity of the smells identified by the tool is limited to individual sentences, disregarding duplicate functionality among documents. Juergens et al. investigated the performance of clone detection techniques to discover redundant functionality descriptions stemmed from copy & paste operations [24]. They argued that redundancy in requirements can hurt modifiability and automated RE. Unfortunately, their clone-based approach works by comparing words, without taking into account the meaning of the sentences.

Falessi et al. investigated several NLP techniques for retrieving equivalent requirements [17]. They aimed at finding duplicate functionality to avoid allocating the same requirement to two or more developers and thus implementing duplicate code. Although this work is very interesting, it was developed with non-structured requirements in mind, that is, it processes raw textual requirements. Our approach takes advantage of use cases structure, terminology and semantics, what allows it to perform complex analyses, such as deriving sequences of steps and identifying abstract actions. Ciemniewska et al. explored several automated techniques for discovering defects in textual use cases [8,9]. They classified use case defects at three different levels: *specifications*, *use cases* and *use case steps*. Unfortunately, the techniques for discovering specification-level defects, which focus on intentional or non-intentional duplication of functionality, are rudimentary and might not be effective enough for their application in industrial environments. In order to identify duplicate functionality, their technique calculates the similarity among use cases by comparing the words of use case steps. We argue that this technique is not well suited to analyze large textual requirements and could have trouble for identifying many instances of duplicate functionality, because it imposes a unique alignment for matching the use cases and does not deal with synonyms (the intention of a use case step). Conversely, our technique, which is prepared to operate on many use cases, takes into consideration any possible alignment and introduces semantic classes to compare semantically similar functionalities.

In the same line of work, Kalmalrudin et al. introduced an approach for assuring an adequate level of the consistency, completeness and correctness of requirements specifications [25]. Their approach, called *MaramaAI*, operates on use cases and relies on the concept of essential use cases (EUC). Initially, the approach links the textual use cases to a semiformal model of essential interactions. Once this linkage is performed, the retrieved model is compared to a library of essential interactions patterns. This library contains a set of well-defined interactions between an actor and the system and it is applicable to many environments (i.e., supporting a variety of application domains). A partial matching between a use case and the patterns of the library could unveil an inconsistent or incomplete specification. The way in which this work takes advantage of EUC interactions is conceptually similar to the domain actions used by *ReqAligner*. Furthermore, the comparison with a library of patterns can be though of as applying text-matching techniques over use case specifications.

Other researchers have focused on the improvement of use cases at the model level (that is, use case diagrams). El-Attar et al. presented an advanced technique for assessing and improving the quality of use case models via the identification of unsound structures [15]. They produced a series of anti-patterns written in the Object Constraint Language (OCL), which can detect potentially defective areas in use case models. The technique is supported by the ARBIUM tool, which allows analysts to define their own anti-patterns. Along this line, Enckevort presented a prototype that identifies defects in UML class models via the evaluation of quality metrics [16]. The metrics are formulated in OCL and, after the prototype detects a problem, it can recommend mending actions. The author suggests that their prototype can be effortlessly extended to other UML diagrams, such as use cases. Ren et al. proposed an assistive approach for refactoring use case models [40]. They developed a prototype tool for automating the requirements refactoring process based on a meta-model of use cases, greatly facilitating the tasks for reorganizing use case models. However, the approaches presented in [15,16] and [40] are limited in their enhancement of use case diagrams, leaving the analysis of duplicate functionality in textual specifications unresolved. This shortcoming is not a minor issue, because relevant functionalities can remain duplicate across multiple documents and the analysts will not notice it, mistakenly believing that the use cases are correctly specified.

Regarding the second group, several works exist for appraising the quality of requirements artifacts and carrying out the necessary actions to mend defects. Ramos et al. instantiated the GQM framework in the context of requirements artifacts to measure the quality of use case specifications [39]. Their approach, called *AIRDoc*, helps to improve the quality of use cases by means of systematic evaluations and refactorings. An evaluation permits to identify a problem, whereas a refactoring helps to solve the issue. AIRDoc defines a catalog of candidate problems to be detected in the evaluation, namely: *Duplicate Steps, or Complex Conditional Structures,* among others. In addition, AIRDoc also provides the analysts with possible refactoring solutions (e.g., *Extract Use Case, Move Use Case Steps,* or *Rename Use Case*, among others).

Rui et al. also explored the application of refactoring techniques in use cases [41]. Particularly, their work illustrates how different refactorings can be used in early development stages, enabling the transformation of a problematic use case model into a better one in a systematic fashion. To this end, they organize and describe the refactorings using a meta-model of use cases. This meta-model supports the formal definition of refactorings through a series of transformation actions. Unfortunately, in both [39] and [41], determining when and where a particular refactoring should be used is yet a call that must be made by the analysts. There is no tool assistance to guide analysts to choose "good" refactorings for improving the textual requirements. Yu et al. studied the subject of use case refactorings [48], developing a catalog of refactorings that focuses on frequent defects of use case specifications, and relies on the concept of *episodes*. An episode is defined recursively as an object that: (i) can be composed by a number of simpler episodes, which tell about a complex interaction between the system and the outside world, or (ii) it represents an atomic unit of behavior of a use case, which details an individual interaction. Refactorings are applied by matching the episode model to the current specification, and thus improving the structure of the use cases. Similarly to the previous two approaches, the aspect of discovering the location (in textual specifications) in which a defect occurs and specially where to apply the refactoring is out of the scope of Yu's proposal.

Another number of approaches have focused merely on the refactoring of use case models. For instance, Einarsson et al. presented an approach for synchronous refactoring of UML diagrams and the underlying UML model using Model-to-Model (M2M) transformations [13]. This approach is supported by a prototype tool built upon the Eclipse-based Papyrus UML editor. Dobrzanski et al. introduced an approach for dealing with "eroded" UML diagrams [12] via the application of refactorings. In order to automate this procedure, they take some decisions, such as the formalization of both system diagrams with Executable UML models and refactoring transformations. Their approach is driven by the so-called "bad smells", meaning that the refactorings are applied in response to the presence of a defect. This kind of approaches is useful for analyzing use case diagrams, but have no support for identifying quality-related problems in the associated specifications.

Finally, an interesting line of work related to the analysis of faulty requirements is that of discovering crosscutting concerns (also referred to as early aspects) [31]. A crosscutting concern (CCC) emerges when a requirement cannot be treated separately from other concerns [36] and ends up being described across multiple documents and mixed with other concerns. Frequently, these concerns are associated, to a greater or lesser extend, with software qualities [6]. Chernak, for example, studied the incorporation of a special kind of refactoring mechanism for dealing with CCCs in textual use cases [7], but leaving the detection of the problems unresolved. Several works have applied Information Retrieval (IR), Machine Learning (ML) and Natural Language Processing (NLP) techniques for identifying CCC in an automated fashion. For instance, EAMiner [42] and Theme/Doc [3] have successfully uncovered this kind of problems in use case specifications. Furthermore, we have also explored the detection of CCC in previous works, by combining clustering techniques with advanced NLP modules [35,36,38]. The discovery of CCC in textual requirements is similar to the task carried out by *ReqAligner,* because it must analyze multiple requirements sources in order to identify clues leading to CCC. However, searching for CCC is relatively easy in comparison to finding duplicate functionality. In general, CCC are confined to individual sentences and can be inferred by looking at certain terms. These terms either relate to some domain-specific functionality (e.g., login or access control features) or are associated with a particular quality attribute that crosscuts requirements (e.g., "time" or "second" for a PERFORMANCE concern).

9 Conclusions

In this article, we have presented a technique called *ReqAligner* that helps analysts to assess the quality of requirements specifications, by identifying duplicate behaviors in the documents and providing guidelines to mend those defects (duplications) in an automated fashion. The technique has been implemented in a prototype tool that supports the analysis of textual use cases based on a pipeline of text processing modules. In particular, *ReqAligner* leverages on a domain-specific classifier of *semantic actions* and, based on such knowledge, employs a *sequence alignment* technique for finding suspiciously similar functionalities in the use cases. Moreover, our technique is also able to suggest UML relationships that can help analysts to remove duplications and ultimately improve the overall requirements model. The novelty of *ReqAligner* lies in the identification of duplicate functionality within use cases using computer-assisted techniques. To the best of our knowledge, this line of research has not been explored yet by the Requirements Engineering community.

We have evaluated our technique with five publicly available case studies with promising results. The case studies come from both academia and industry, covering a varied range of system domains. *ReqAligner* achieved a very good *Recall*, meaning that it can recover the majority of the defects from the specifications of the case studies. This is an important achievement, because it confirms the potential of SA algorithms in SE-related artifacts. Furthermore,

the performance of the tool was reasonable, since the analyses performed by our technique only took a few minutes to complete. This performance is attributed to the Smith–Waterman's algorithm implemented in JAligner. Unfortunately, the *Precision* results were lower than those of *Recall*, reaching only an acceptable level. *ReqAligner* reported several instances that were not actually duplicate functionalities. In most cases, this situation was attributed to the usage of a very similar writing style among multiple use case documents, which our technique mistakenly considered as duplicate functionality. Still, we are confident that a human analyst will have no trouble to discard these false positives by just taking a quick look at the results. We also explored the quality of the alignments produced with our technique, comparing the boundaries of the duplicate functionality detected with those defined in the "correct" solution. We found that our technique is very accurate, although it can include more use case steps than necessary. At last, we examined whether the recommendations implemented in *ReqAligner* can steer the analysts to fix the defects. In our experiments, we found that the advice provided by the heuristics was generally correct, except for a few mistakes related to a poor demarcation of duplicate functionality.

Although our technique performed remarkably well on the experiments, there are some limitations that should be taken care of. One limitation is the detection of false positives. For instance, *ReqAligner* cannot notice that similarities between CRUD use cases are not a defect. In general, the technique has trouble to tell the difference between real duplication and very similar use case scenarios (lexically and syntactically speaking). Another limitation is the performance of our prototype. Even though performance was not an issue during the evaluation with case studies, we believe that the complexity of the SA algorithm will increase with larger systems (with more use cases). It is also worth-mentioning that the technique may need adaptations to function in specialized SE domains (e.g., financial developments, critical systems, or embedded software) regarding the current set of domain actions. In addition, the current implementation of the tool cannot process existing relations like explicit inclusions or extension points present in textual use cases. Finally, since *ReqAligner* currently produces pairwise alignments, there might be cases in which functionality being duplicated among more than two sequences gets lost in the analysis.

As future work, we are planning to run more evaluations with the specifications, as well as to incorporate additional case studies. We are also investigating ways for improving and refining the hierarchy of domain actions, in an attempt to better capture the semantics of use cases. This activity should improve the accuracy of *ReqAligner* in unknown environments and enable the analysis of other software development domains. Along this line, we are looking forward to making adjustments in the parameters of the SA technique. For instance, we think that a rigorous definition of the substitution matrix and gap schema (e.g., using a *variation of parameters* analysis) should lead to better alignments and consequently to a better detection of duplication. Another line of research is to include information about the actors that participate in a use case step and the objects affected by domain actions to increase the precision of our approach. Regarding the processing of existing relations, we are studying alternatives to support the automatic expansion of included and extended use cases during the assembly of sequences. It is also interesting to investigate whether multi-sequence alignment algorithms are feasible in requirements specifications (in terms of both performance and accuracy), as well as to analyze the tradeoffs brought by such algorithms. Finally, we intend to sort the duplicate functionalities reported by the tool according to multiple criteria, such as confidence of detection, magnitude of duplication, or consequences of the duplication, among others. This kind of ranking can serve to present not only the most critical defects first, but also to place unreliable detections last in the list, and therefore it is expected to facilitate the analyst's work.

Overall, we think *ReqAligner* is a compelling alternative for an analyst to scan a large set of use case specifications in an automated fashion, providing a quick and useful output to drive the improvement of the requirements. Nonetheless, analysts always have to make the final decisions on whether duplicate functionality should be refactored, considering an appropriate balance between understandability and modularity of the use case specifications.

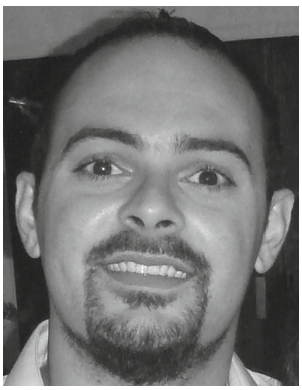
Acknowledgments The authors would like to thank Paula Frade and Miguel Ruival, who implemented the *ReqAligner* prototype and evaluated the technique as part of their final project for the degree of Bachelor in Systems Engineering at UNICEN University. Also, the authors are grateful to the analysts who defined the reference solution for the evaluation of the technique. The authors also thank the anonymous reviewers for their feedback that helped to improve the quality of the manuscript.

References

1. Adolph, S., Bramble, P., Cockburn, A., Pols, A.: Patterns for Effective Use Cases. The Agile Software Development Series. Addison-Wesley, Reading, MA (2003)
2. Alchimowicz, B., Jurkiewicz, J., Ochodek, M., Nawrocki, J.: Building benchmarks for use cases. Comput. Inf. **29**(1), 27–44 (2010)
3. Baniassad, E., Clarke, S.: Theme: an approach for aspect-oriented analysis and design. In: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, Scotland, UK, pp. 158–167 (2004)
4. Bell, R.: Course registration system. http://sce.uhcl.edu/helm/RUP_course_example/courseregistrationproject/indexcourse.htm

5. Booch, G., Rumbaugh, J., Jacobsen, I.: The Unified Modeling Language User Guide. The Addison-Wesley Object Technology Series. Addison Wesley, Reading (1999)
6. Chen, L., AliBabar, M., Nuseibeh, B.: Characterizing architecturally significant requirements. IEEE Softw. **30**(2), 38–45 (2013). doi:10.1109/MS.2012.174
7. Chernak, Y.: Building a foundation for structured requirements. Aspect-oriented re explained—part 1. Better Software (2009)
8. Ciemniewska, A., Jurkiewicz, J.: Automatic detection of defects in use cases. Master's thesis, Poznan University of Technology—Faculty of Computer Science and Management—Institute of Computer Science (2007)
9. Ciemniewska, A., Jurkiewicz, J., Olek, L., Nawrocki, J.: Supporting use-case reviews. In: Proceedings of the 10th International Conference on Business Information Systems (BIS'07), Springer, Berlin, Heidelberg, pp. 424–437 (2007)
10. Cockburn, A.: Writing Effective Use Cases, vol. 1. Addison-Wesley, Reading (2001)
11. Dekhtyar, A., Dekhtyar, O., Holden, J., Hayes, J., Cuddeback, D., Kong, W.K.: On human analyst performance in assisted requirements tracing: statistical analysis. In: 2011 19th IEEE International on Requirements Engineering Conference (RE), pp. 111–120 (2011). doi:10.1109/RE.2011.6051649
12. Dobrzanski, L., Kuzniarz, L.: An approach to refactoring of executable UML models. In: Proceedings of the 2006 ACM Symposium on Applied Computing, ACM, New York, NY, USA, SAC '06, pp. 1273–1279 (2006). doi:10.1145/1141277.1141574
13. Einarsson, H.T., Neukirchen, H.: An approach and tool for synchronous refactoring of UML diagrams and models using model-to-model transformations. In: Proceedings of the Fifth Workshop on Refactoring Tools. ACM, New York, NY, USA, WRT '12, pp. 16–23 (2012). doi:10.1145/2328876.2328879
14. Eissen, S.M., Stein, B.: Intrinsic plagiarism detection. In: Advances in Information Retrieval. Lecture Notes in Computer Science, vol. 3936, Springer, Berlin Heidelberg, pp. 565–569 (2006)
15. El-Attar, M., Miller, J.: Improving the quality of use case models using antipatterns. Softw. Syst. Model. **9**(2), 141–160 (2010). doi:10.1007/s10270-009-0112-9
16. Enckevort, Tv.: Refactoring UML models: using openarchitectureware to measure uml model quality and perform pattern matching on UML models with OCL queries. In: Proceedings of the 24th ACM SIGPLAN conference companion on OOPSLA '09. ACM, New York, NY, USA, pp. 635–646 (2009). doi:10.1145/1639950.1639959
17. Falessi, D., Cantone, G., Canfora, G.: Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques. IEEE Trans. Softw. Eng. **39**(1), 18–44 (2013). doi:10.1109/TSE.2011.122
18. Femmer, H., Fernández, D.M., Juergens, E., Klose, M., Zimmer, I., Zimmer, J.: Rapid requirements checks with requirements smells: two case studies. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (RCoSE'14) Held at the 36th International Conference on Software Engineering (ICSE'14). ACM, Hyderabad, India, pp. 10–19 (2014). doi:10.1145/2593812.2593817
19. Gotoh, O.: An improved algorithm for matching biological sequences. J. Mol. Biol. **162**(3):705–708 (1982). http://view.ncbi.nlm.nih.gov/pubmed/7166760
20. Greenwood, P.: Tao: A testbed for aspect oriented software development. (2011). http://www.comp.lancs.ac.uk/~greenwop/tao/
21. Horton, R., Olsen, M., Roe, G.: Something borrowed: Sequence alignment and the identification of similar passages in large text collections. Digital Studies/Le Champ Numérique **2**(1) (2010). http://www.digitalstudies.org/ojs/index.php/digital_studies/issue/view/25
22. Hull, E., Jackson, K., Dick, J.: Requirements Engineering. Springer, Berlin (2010). http://books.google.com.ar/books?id=5xREIrqnDQEC
23. Issa, A., Odeh, M., Coward, D.: Using use case patterns to estimate reusability in software systems. Inf. Softw. Technol. **48**(9), 836–845 (2006)
24. Juergens, E., Deissenboeck, F., Feilkas, M., Hummel, B., Schaetz, B., Wagner, S., Domann, C., Streit, J.: Can clone detection support quality assessments of requirements specifications? In: Proceedings of the 32 ACM/IEEE International Conference on Software Engineering (ICSE'10), vol. 2. ACM, New York, NY, USA, pp. 79–88 (2010). doi:10.1145/1810295.1810308
25. Kamalrudin, M., Hosking, J., Grundy, J.: Improving requirements quality using essential use case interaction patterns. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), Waikiki, Honoulu, Hawaii, pp. 531–540 (2011). doi:10.1145/1985793.1985866
26. Kamata, M.I., Tamai, T.: How does requirements quality relate to project success or failure? In: Proceedings of the 15th IEEE International Requirements Engineering Conference (RE'07). IEEE Computer Society, New Delhi, India, pp. 69–78 (2007). doi:10.1109/RE.2007.31
27. Kulak, D., Guiney, E.: Use Cases: Requirements in Context, 2nd edn. Pearson Education, New Jersey (2012)
28. Larman, C.: Applying Uml and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd edn. Pearson Education, New Jersey (2012)
29. Manning, C., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, Cambridge (2008). http://books.google.com.ar/books?id=t1PoSh4uwVcC
30. Mich, L., Franch, M., Novi, I.P.L.: Market research for requirements analysis using linguistic tools. Requir. Eng. **9**(2), 151 (2004). doi:10.1007/s00766-004-0195-3
31. Moreira, A., Chitchyan, R., Araujo, J., Rashid, A. (eds.): Aspect-Oriented Requirements Engineering, vol. XIX. Springer, Berlin Heidelberg (2013). doi:10.1007/978-3-642-38640-4
32. Niazi, M., Shastry, S.: Role of requirements engineering in software development process: an empirical study. In: 7th International Multi Topic Conference (INMIC 2003), pp. 402–407 (2003). doi:10.1109/INMIC.2003.1416759
33. Palmer, M., Gildea, D., Xue, N.: Semantic Role Labeling. Synthesis Lectures on Human Language Technologies. Morgan & Claypool (2010). http://books.google.com.ar/books?id=6C1Ag3NUqNEC
34. Polanski, A., Kimmel, M.: Bioinformatics. Springer, Berlin (2007). http://books.google.com.ar/books?id=oZbR3GEdmVMC
35. Rago, A., Abait, E., Marcos, C., Diaz-Pace, A.: Early aspect identification from use cases using NLP and WSD techniques. In: Proceedings of the Workshop on Early Aspects held at the 15th International Conference on Aspect-Oriented Software Development (AOSD'09). ACM, Charlottesville, Virginia, USA, pp. 19–24 (2009). doi:10.1145/1509825.1509830
36. Rago, A., Marcos, C., Diaz-Pace, A.: Uncovering quality-attribute concerns in use case specifications via early aspect mining. Requir. Eng. **18**(1), 67–84 (2013). doi:10.1007/s00766-011-0142-z
37. Rago, A., Marcos, C., Diaz-Pace, A.: Assisting requirements analysts to find latent concerns with reassistant. Autom. Softw. Eng. (2014). doi:10.1007/s10515-014-0156-0
38. Rago, A., Marcos, C., Diaz-Pace, A.: Una comparación de técnicas de nlp semánticas para analizar casos de uso (in spanish). In: Proceedings of the 2nd IEEE Biennial Congress of Argentina (ARGENCON'14), IEEE Argentina, Bariloche, Argentina (2014)
39. Ramos, R., Castro, J., Alencar, F., Araújo, J., Moreira, A., de Engenharia da Computacao, C., Penteado, R.: Quality improvement for use case model. In: XXIII Brazilian Symposium on Software Engineering, 2009. SBES'09. IEEE, pp. 187–195 (2009)

40. Ren, S., Butler, G., Rui, K., Xu, J., Yu, W., Luo, R.: A prototype tool for use case refactoring. In: Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS'04), Porto, Portugal, pp. 173–178 (2004)
41. Rui, K., Butler, G.: Refactoring use case models: the metamodel. In: Proceedings of the 26th Australasian computer science conference-Volume 16, Australian Computer Society Inc, pp 301–308 (2003)
42. Sampaio, A., Rashid, A., Chitchyan, R., Rayson, P.: EA-Miner: towards automation in aspect-oriented requirements engineering. In: Transactions on Aspect-Oriented Software Development III. Lecture Notes in Computer Science, vol. 4620, Springer, Berlin, pp. 4–39 (2007)
43. Sateli, B., Angius, E., Rajivelu, S.S., Witte, R.: Can text mining assistants help to improve requirements specifications? In: Mining Unstructured Data (MUD 2012), Kingston, Ontario, Canada, (2012). http://sailhome.cs.queensu.ca/mud/res/sateli-mud2012.pdf
44. Schneider, G., Winters, J.P.: Applying Use Cases: A Practical Guide. Object Technology Series, 2nd edn. Addison Wesley, Reading, MA (2001)
45. Sinha, A., Paradkar, A., Kumanan, P., Boguraev, B.: An analysis engine for dependable elicitation of natural language use case description and its application to industrial use cases. IBM Research Report RC24712 (2008)
46. Smith, T., Waterman, M.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
47. Tsoumakas, G., Katakis, I., Vlahavas, I.: Mining multi-label data. Data Mining and Knowledge Discovery Handbook, pp. 667–685 (2010)
48. Yu, W., Li, J., Butler, G.: Refactoring use case models on episodes. In: Proceedings of the 19th International Conference on Automated Software Engineering. IEEE, pp. 328–335 (2004)



Alejandro Rago is currently a PhD. student in the Computer Science Program of Faculty of Ciencias Exactas at UNICEN University (Tandil, Argentina). Alejandro's research interests are on the automation of software engineering activities, especially at early development stages. Alejandro holds a Bachelor's and Master's degrees in Systems Engineering both from UNICEN University. Contact him at arago@exa.unicen.edu.ar

Claudia Marcos has been a Professor at UNICEN University (Tandil, Argentina) since 1991. She is a research fellow of the Commission for Scientific Research (CIC) of Buenos Aires, Argentina. From 2000 to 2005, she was co-director of the ISISTAN Research Institute. Her main research interests are: software evolution, requirements engineering and agile development. She teaches several undergraduate and postgraduate courses at UNICEN University, and also has published several articles on those topics. She regularly advises Master and PhD students. Dr Marcos received a Ph.D. in Computer Science from UNICEN University in 2001. Contact her at cmarcos@exa.unicen.edu.ar

J. Andres Diaz-Pace is currently a professor at UNICEN University (Tandil, Argentina), and also a research fellow of the National Council for Scientific and Technical Research of Argentina (CONICET). From 2007 to 2010, he was a member of the technical staff at the Software Engineering Institute (SEI, Pittsburgh, USA), with the Research, Technology, and System Solutions Program. His primary research interests are: quality-driven architecture design, AI techniques in design, architecture-based evolution and conformance. He has authored several publications on topics of design assistance and object-oriented frameworks. He also participated, as an architecture evaluator or as a lead architect, in several technology transfer projects with industry. Mr. Diaz-Pace received a Ph.D. in Computer Science from UNICEN University in 2004. Contact him at adiaz@exa.unicen.edu.ar