

A Complete Traceability Methodology Between UML Diagrams and Source Code Based on Enriched Use Case Textual Description

Wiem Khelif, Dhikra Kchaou and Nadia Bouassida

E-mail: Wiem.khlif@gmail.com, Dhikra.Kchaou@fsegs.rnu.tn, nadia.bouassida@isimsf.rnu.tn

Sfax University, Mir@cl Laboratory, Tunisia

Keywords: traceability, UML diagrams, use case, enriched textual description, control structure.

Received: September 9, 2020

Abstract: Traceability in software development proves its importance in many domains like change management, customer's requirements satisfaction, model slicing, etc. Existing traceability techniques trace either between requirement and design or between requirement and code. However, none of the existing approaches achieved reliable results when dealing with traceability between requirements, design models and source code. In this paper, we propose an improvement and an extension of our design traceability approach in order to tackle the traceability between design, requirement and code. The fine-tuning of our methodology stems from considering an expanded textual description. A pre-treatment step is added in order to divide the textual description of system functionalities into different parts, each of which represents a specific goal. In fact, the extension consists in extracting an expanded textual description from a natural language text in order to trace between related elements belonging to requirement, design and code while using an information retrieval technique. The proposed method is based on different scenarios (nominal, alternatives and errors), particularly on concepts related to control structures to establish the traceability between artefacts. Furthermore, we implemented our method in a tool allowing the evaluation of its performance. The evaluation is performed on real existing applications that consist in comparing results found by our approach with results found by experts. Our method achieves an average precision of 0.84 and a recall of 0.91 in traceability between requirement, design and code. Besides its promising performance outcomes, our automated method has the merit of generating a traceability report describing the correspondence between different artefacts.

Povzetek: Prispevek opisuje novo metodo za sledenje povezavam med UML diagrami in izvorno kodo.

1 Introduction

Traceability quality is defined as the degree to which existing artefacts of a software development project are traceable as mandated by the project's traceability stakeholders. The Unified Modelling Language (UML) is used for specifying, constructing, and documenting these artefacts. It is composed of a set of diagrams grouping structural and semantic dependencies between UML elements [1]. Based on the unified process, UML diagrams are produced iteratively and incrementally from use case diagram (UCD) to code. An iteration generates a baseline that comprises a partially complete version of the final system. Each one results in an increment, which is a release of the system that contains added or improved functionality compared with the previous release. Each iteration goes through five activities that specify what needs to be done: requirements, analysis, design, implementation and test. Requirements are modelled by (UCD) and their textual descriptions while the design is modelled through UML diagrams (class, sequence, etc.). These diagrams are strongly related either within one iteration or between iterations and consequently the lack of traceability between them makes any change difficult and expensive. Determining and keeping traceability between UML models is important for many reasons. For

instance, in the context of change impact analysis, a change in one iteration often leads to changes in the following iterations. Certainly, the major challenge when developing a requirement change consists in creating traceability links between heterogeneous artefacts produced at different abstraction levels [2]. For example, adding data and actions in a use case (UC) description leads to add the corresponding methods and attributes in the class diagram and in the code. In fact, tracing change inter-UML diagrams into the source code is crucial to maintain the consistency and coherence. However, creating accurate and complete traceability is costly and remains a practical challenge [2]. In fact, we focus in this paper on determining traceability by considering structural and behavioural aspects. Furthermore, it is crucial to keep traceability between UML models since it allows checking the conformance between safety requirements and design decisions through model slicing. Thus, traceability definition is used to extract design slices that filter out irrelevant design details and keep information to inspect compliance between requirements and design [3, 4]. The recent literature on traceability shows two trends of approaches: those centred on traceability inter-UML models [4, 5, 6, 7, 8, 9], and those based on traceability from requirement and design to code [10, 11, 12, 13, 14, 15]. The first type of approaches

tackles the traceability within a set of models elements, particularly from requirements modelled by a UCD to design diagrams [6]. For instance, [16] deals with traceability between software architectural models and extra-functional results such as performance and security. Kchaou et al., [6] present traces between requirements modeled by a UCD and UML design diagrams. On the other hand, [4] illustrates the traceability between Use Case Maps and UML diagrams and [8] identifies the traceability between requirement and design models modeled with SysML. The second type of traceability approaches defines links between different models (requirements, design, test cases, etc.) and source code. These works differ in terms of the used techniques. These traceability approaches use exclusively either information retrieval techniques [17, 18, 19], a meta-model [20], Natural Language Processing (NLP) techniques [21] or machine learning techniques [22]. However, none of the existing approaches deals with traceability between requirements, design models and source code by covering all the concepts that can be determined in all levels (control structures, how activities are carried out, etc.). That is, the so-far proposed approaches neglect additional semantic and/or structural information that can be extracted. The lack of this information may reduce the scope of possible analyses that can be made and possible traceability links that may be found. In addition, in the literature, traceability from requirement and design to the source code is based generally on the class diagram, which does not produce all necessary information such as control structure. Consequently, class diagrams allow engineers to understand its structure but it does not show the behavior of the software [5]. To understand its behaviour, dynamic models are needed, such as sequence, activity or state transitions diagrams [1]. Moreover, while the existing approaches use a semantic technique to compute similarities between different artifacts based on specific and common terms (e.g., actors, actions, etc), they do not cover all kinds of terms like behavioral elements (Parallel, alternative, loop, etc.), type of result, functional call, etc.

In this paper, we first show how the approach, initially presented in [6], that traces the elements of design diagrams, can be improved, fine-tuned and automated in order to discover correlated structural and semantic information and to trace between different UML diagrams, and between these diagrams and the source code. So, we have improved our previous work by defining an Enriched Textual Description (ETD) of a UC. The latter is extracted from a text written in a natural language and describing a software. In addition, the defined ETD allows tracing between the design and code. Unlike existing works (e.g. [4, 8, 21]), we propose a method called TRADIAC Quality (TRAcability for UML DIAGrams and Code) that proceeds in three phases: “Pre-processing Natural language”, “Traceability Inter-UML diagrams” and “Traceability from requirement and design to code”. The “Pre-processing” phase receives as input the whole textual description of a software written in natural language. Then, the textual description is split into parts that achieve a specific goal expressing each one a

functionality (use case). After that, each part is specified by using an enriched template that encapsulates the semantic information pertinent to the functional and behavioural aspects. In this work, we enrich the used textual description template [6] by basic control structures (BCS) (loop, if, switch, etc.) and a set of key words (e.g. PARALLEL expressing how activities are carried out) which take into account many important concepts in the design and code. This template is used for the requirements specification as a mean to document a UC. Compared to the presented template in [6], the enriched one provides more comprehensive traceability. For instance, in [6], the proposed approach does not determine which UC corresponds to which function in the code. In addition, it does not focus on details in alternative behavioural elements such as control structures. The second phase of our method “Traceability process inter-UML diagrams” is composed of traceability rules identification and similarity calculation. Traceability rules detect the relationships between requirements and design models. They distinguish between two traceability levels: structural and semantic. Structural traceability determines structural relationships between UML diagrams. Semantic traceability, which discriminates our method, is useful by considering that use case diagrams and their textual descriptions are based on a well-structured text. It searches the meaning of words contained in these descriptions and their synonyms to find similarities with terms used in the rest of UML diagrams. We note that the semantic traceability between the enriched textual description associated to a UC and other diagrams is based on an information retrieval technique. More specifically, it uses the Latent Semantic Indexing (LSI) similarity measure to estimate the similarity between corresponding elements. The choice of this measure is based on evaluations presented in [6] which showed that LSI is better suited to measure the semantic similarity. In its third phase, our method determines the traceability from requirement and design to code. It allows keeping traceability links from requirements into design and code by adding implementation details. To do so, it uses the traceability process from requirement to code which applies the defined traceability rules specific to details in the source code and calculates the similarity between the selected fragment in the textual description of a UC and code.

To show the advantages and limits of our method, we conduct an experimental evaluation thanks to TRADIAC (TRAcability for UML DIAGrams and Code) tool, which implements all of the method phases. For the herein presented evaluation, we applied a set of measurements (precision, recall, F-measure) to examine the conformity degree between corresponded elements generated by our method with the corresponded elements where traceability is evaluated by experts. This experimentation aims at proving that these models have similar quality values. For these quantitative evaluations, we used two case studies related to different domains. Our method shows an average precision of 84,1%, and an average recall of 91%. The results showed the efficiency of our method in terms of finding correct traceability reports.

The remainder of this paper is organized as follows: Section 2 overviews existing works that define traceability relationships between requirement and other diagrams, and from requirement and design to code. Section 3 presents our method in two subsections: the first subsection presents the pre-processing phase and the enriched textual description to document use cases based on basic control structures. The second subsection is composed of two parts: the first one identifies the traceability rules to facilitate first the transition from the requirement to design level by deriving other diagrams, particularly dynamic diagrams, and then derive code. The second part illustrates the LSI similarity which determines traceability between UML diagrams. To show the improvements gained by applying the traceability rules, we evaluate in section 4 our method and we consider threats to validity of the study and the results. Section 5 presents the tool support and illustrates the method through an example. Finally, Section 6 summarizes the presented work and outlines its extensions.

2 Related work

Several works cope with traceability based on different axes: covered artefacts (e.g. Horizontal vs. vertical) [17, 24, 25] representing the purpose of the traceability (e.g. finding inconsistency among artefacts, impact analysis, knowing the dependencies among artefacts, reuse) [25, 26, 27, 28], challenges and solutions [14, 29, 30, 31], etc.

As highlighted in the introduction, existing traceability approaches adopt either horizontal or vertical approaches. Horizontal traceability determines artifact dependencies at the same abstraction level (requirement, or design or code), while vertical traceability traces artifacts between different models at different abstraction levels. In this paper, we focus on vertical traceability which is classified into two categories: The first one focuses on traceability inter-UML models (requirement and design) and the second one determines traceability between requirements, UML models and code.

2.1 Traceability inter-UML models

Traceability inter-UML models approaches tackles the traceability within UML diagrams elements, particularly from requirements to design diagrams.

Adopting this type of approach, [4] considers the traceability relationships between Use Case Maps (UCMs) and UML diagrams. The proposed approach generates UML diagrams from UCMs notation to describe the system at high abstraction level. This work neglects several concepts that relate UML diagrams such as repetitive and conditional treatment.

In [14], the authors present an approach that supports the automatic maintenance of traceability relations between requirements, analysis and design models of a software systems expressed in UML. It followed two major phases: Recognition phase and maintenance. The first phase consists in capturing elementary changes to model elements and recognizing the compound development activity applied to the model element. The

second phase, “Maintenance” consists on updating the traceability relations associated with the changed model element. A prototype called Trace Maintainer has been implemented to evaluate the approach.

In [32], an approach is presented to specify semantic relationships between system-level requirements, functional specifications, and architectures in terms of their subsystem specifications. This approach is based on logic predicate to present artifacts and their relations at different abstraction levels (Requirements, specification and architecture). The logical representation of each artifact is used by the authors to formalize relationships between these artifacts.

Adopting an abstract approach in defining traceability between software requirements and UML design, [20] proposes FUTOR (From Uml TO Requirement) guideline, which includes meta-model and process step. The meta-model expresses relationships between requirements and the UML model at the meta-level. For each meta-requirement, the author adds a “REQTYPE” attribute to decide which UML diagram shall be used for the traceability. Steps of the FUTOR guideline include: (1) writing requirements (2) annotate the requirement (3) start software design based on requirements, (4) check the traceability between requirements and UML models. This approach neglects information existing between requirements presented as textual documents and UML diagrams at the instance level.

In addition, [8] proposes a hybrid approach that combines graphs and information retrieval techniques to identify the requirement change impact on design models modeled with Systems Modeling Language (SysML). This approach is limited to traceability between requirements modeled with SysML and behavioral diagram modeled with the activity diagram. In addition, many behavioral aspects in the activity diagrams are not assigned like Join node, Fork node, etc.

For the purpose of reuse, [33] depicts an approach that derives systematically a standard functional model from a use case diagram, a structure diagram and a transition diagram. By decomposing the existing functional model into model components, traceability links are recovered based on guidelines that allow a mapping of model components to non-functional requirements. This approach is limited to use cases names without referring to use case descriptions.

Adopting an Information Retrieval (IR) technique to identify traceability between requirement and design, [34] proposes a method that uses graphs to model the structural dependencies. The Information Retrieval technique is used to handle the semantic traceability between the use case documentation and the sequence diagram. This approach is based on a structural textual description of a use case to express requirements. However, this description lacks structural controls which are used in UML behavioural diagrams.

On the other hand, [6] proposes a method that uses graphs to model the structural dependencies and an information retrieval technique to handle the semantic traceability between the use case documentation and the sequence diagrams. This approach is based on a structural

textual description of a use case diagram to express requirements; however, this description lacks structural controls which are used frequently in the UML behavioural diagrams. In fact, it is not possible to trace between behavioural elements in design and control structures in code functions such as loop, switch, etc. Additionally, the limitation of this approach lies in its incapability to determine the nature of functions/ methods that corresponds to a use a case textual description.

Furthermore, several approaches adopt a Natural language processing approach (NLP). For instance, [35] determines basic elements of a class diagram from natural language requirements. Requirements are presented in English and the designed tool (Natural language Processing for Class NLPC) applies NLP methods to analyze the given input. Natural language text is semantically analyzed to obtain classes, data members and member functions. NLPC uses pre-processing, Part of Speech (POS) Tagging, Class Identification, Attribute and Function identification to plot the classes.

[5] extracts class diagrams from natural language requirements using NLP techniques such as WordNet, OpenNLP parser, class extraction engine, etc. Moreover, the authors proposed a system based on rules to extract details related to the object oriented concepts like generalization, association and dependency from natural language requirements specification.

Furthermore, [35] adopts a NLP approach to show that natural language requirements are semantically analysed to obtain classes, data members and member functions.

Based on a combination between NLP and artificial neural networks, [36] proposes a new approach to automatically identify actors and actions in a natural language based requirements description of a system. They used an NLP parser with a general architecture for text engineering, producing lexicons, syntaxes, and semantic analyses. An artificial neural networks (ANN) was developed using five different use cases, producing different results due to their complexity and linguistic formation.

2.2 Traceability from requirement and design to code

Besides traceability inter UML models, the vertical traceability approaches tackle also the relationships between requirement, design and code [20, 37, 38, 39].

In this context, to support traceability between requirement and source code, [20] proposes a meta-model based approach that defines traceability links between different artifacts (requirements, test cases, etc.) and source code. The authors propose an editor to visualize traceability between the source code stored as an Abstract Syntax Tree (AST) and other possible artifacts. However, the use of an AST causes foreign problems like the existence of syntax errors and comments in the source code which loses traceability links.

In [37], the focus is on the traceability between requirement and source code in the context of version control system. Specifically, the authors study the link

between issues (i.e. new requests), commits (change set), and source code files. They train a classifier to identify missing issue tags in commit messages to generate missing links.

Besides, in the purpose of supporting traceability between requirement and source code, [40] introduces a solution for automating the evolution of bidirectional trace links between source code classes or methods and requirements. The solution depends on a set of heuristics coupled with refactoring detection tools and informational retrieval algorithms to detect predefined change scenarios that occur across contiguous versions of a software system.

To trace between requirements documents, UML class diagrams, and source code, [41] [42] use graph and XML format to capture links between artifact elements.

Based on a set of policies, [38] [39] describe an approach which allows maintaining traceability of evolving architecture to implementation links. They develop a tool “ArchTrace” which maintain existing traceability link. These links have to be created manually by the developers or by a traceability recovery method. In addition, the authors distinguish between four classes of rules depending on the level where the change occurs. For instance, architectural element evolution policies trigger when an architect makes modifications to an architecture. An example of an architectural policy is illustrated in the case of creating a new version of an architectural element [39]. This new version of this element should inherit all traceability links from its ancestor based on a copy of all traceability links from its previous version.

By referring to machine learning techniques, [22] presents a process to recover traceability links between Java programs entities and elements in a use case diagram. This solution, which is called LEarning and ANALyzing Requirements Traceability (LeanArt), combines program analysis, run-time monitoring, and machine learning to search similarities between the names and values of program entities, and the elements names of use case diagrams. This work is only based on traceability between use case name and source code. Nonetheless, it does not take into account the different scenarios that can be found in a use case textual description.

Likewise, [14] proposes an approach called TRAIL (TRAcability lInk cLassifier) that applies Traceability Link Recovery (TLR) as a binary classification problem for automating traceability maintenance. It uses historically collected traceability information (i.e., existing traceability links between pairs of artifacts) to train a machine learning classifier which is then able to classify the link between any new or existing pair of artifacts as valid (i.e., the two artifacts are related) or invalid (i.e., the two artifacts are unrelated) [29]. To determine the validity of the link between two artifacts, TRAIL introduces three types of features: IR Ranking, Query Quality, and Document Statistics.

[43] proposes a neural network architecture that utilizes word embedding and Recurrent Neural Network (RNN) technique to automatically generate trace links. Word embedding learns word vectors that represent knowledge of the domain corpus and RNN uses these

word vectors to learn the sentence semantics of requirements artifacts. The authors use an existing training set of validated trace links from the domain to train the RNN to predict the likelihood of a trace link existing between two software artifacts. For each artifact (i.e. requirement, source code file, etc.), each word is replaced by its associated vector representation learned in the word embedding training phase and then sequentially fed into the RNN. The final output of RNN is a vector that represents the semantic information of the artifact. The tracing network then compares the semantic vectors of two artifacts and outputs the probability that they are linked.

IR techniques are used also to define traceability between models and the source code. [17, 18, 19] use the Latent semantic indexing (LSI) to recover traceability between different artifacts. For instance, [17] uses LSI to recover traceability links between software artefacts produced during the different phases of a development project (use case diagrams, interaction diagrams, test cases and code). [7] utilizes comments and identifier names within the source code to match them with sections of corresponding documents. [13] establishes traceability between requirement and other software elements (code elements, API documentation, and comments) by taking into account the change frequency, and the semantic similarity (TF-IDF) between the requirement description and the software element.

In order to improve IR-based traceability recovery, [44] combines IR techniques with closeness analysis. Specifically, the work quantifies and utilizes the “closeness” for each call and data dependency between two classes to improve rankings of traceability candidate lists. In [45], the authors propose an improvement of the previous approach by introducing user feedback into the closeness analysis on call and data dependencies in code. Specifically, the approach iteratively asks users to verify a chosen candidate link based on the quantified functional similarity for each code dependency (which they called closeness) and the generated Information Retrieval values. The verified link is then used as the input to re-rank the unverified candidate links.

Based on NLP techniques, [25] defines an enhanced framework of software artefact traceability management which is implemented in the “SATAnalyzer” tool. NLP techniques are used to extract information from artefacts produced during software development process. The tool supports the traceability between requirements, UML class diagrams, and corresponding Java code. [15] extends the SAT-Analyzer tool to consider traceability among other stages of development life cycle such as testing and deployment with enhanced visualization suitable for DevOps practices and continuous integration.

In order to evaluate their graph-based traceability approach, [46, 47] use also the SAT-Analyser tool with a “Sale system Point” case. They present phases such as software artefact identification, data preprocessing, data extraction and traceability establishment methodologies presented with a graph. The tool traces software requirement artifact in natural language, only UML class diagram as design artefact and the Java source code artifact. The traceability graph construction is based on similarity algorithms (Jaro Winkler Distance and Levenshtein Distance) between requirements, classes, methods, attributes and the relationships inheritance, association and generalization.

Using a model-based approach, [25, 48] derive a quality model to present traceability (Traceability Assessment Model (TAM)) that specifies per element (class, link, path) the acceptable state (Traceability Gate) and unacceptable deviations (Traceability Problem) from this state. The authors describe how both, the acceptable states and the unacceptable deviations can be detected to systematically assess their project’s traceability. In order to improve the previous works, [2] defines a system allowing to ensure that the software delivered meets all requirements and thus avoids failures by using data traceability management.

In summary, existing works tackled the traceability either between UML diagrams at the same abstraction level (or similar notations) or between UML models (requirements, design, etc.) and the source code, at different abstraction levels. However, none of the existing approaches deal with traceability between requirements presented with an enriched template that covers the whole

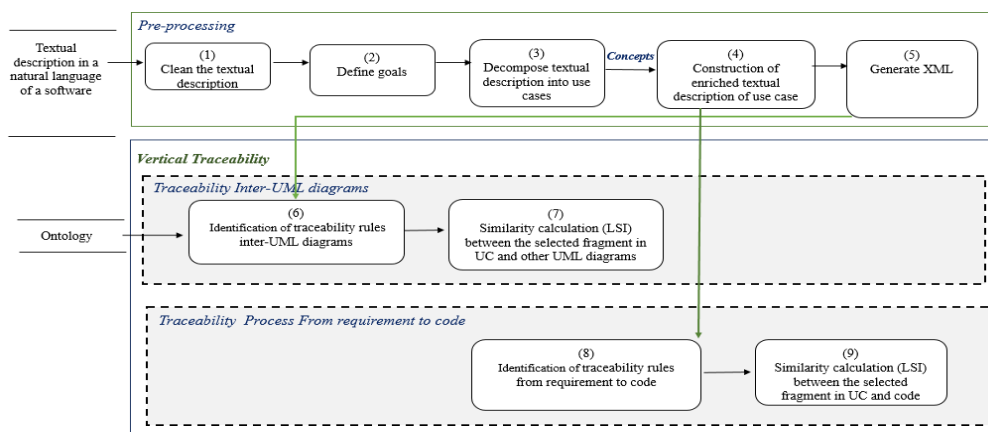


Figure 1: The proposed method for tracing UML code based on textual description of use cases.

concepts, design models and source code. In addition, all traceability techniques [49, 50] rely on either the structural and/or semantic information. For example, [20, 21, 41] determine traceability between heterogeneous terms existing in models (text in requirements, classes name, methods name, etc.). These works are purely structure-based; they ignore the remaining aspects of UML diagrams elements, which do affect the traceability between them.

The purpose of the proposed method focus on enriching the requirement template presented in [6] to cope with the control structures and orient our traceability. Furthermore, it combines both structural and semantic aspects in order to determine the traceability between all elements at different abstraction levels and detects the relationships between the requirements, design (modelled with sequence (SD), class (CD), activity (AD) and state transition (STD) diagrams (first phase), and the source code (second phase).

3 A new traceability method

Figure 1 depicts our method for determining vertical traceability. It followed three major phases: “Pre-processing Natural language”, “traceability inter-UML diagrams” phase and “traceability from requirement and design to code” phase.

The “Pre-processing Natural language” phase during which the software analyst receives a textual description of a software written in a natural language. The description is cleaned based on simple NLP technique (i.e. Stanford CoreNLP tool) [51]. Then, the software analyst uses the output to identify the goals that are used to divide the textual description into different parts. The proposed decomposition guides and improves the generation of description parts and the corresponding fragments related to design diagrams in a more systematic, rigorous, and consistent way. For each description part, the software analyst prepares its textual description according to a specific template. To handle this requirement, we define an enriched template that can be written in a specific format. The template is used to generate its corresponding XML file. The second phase, “Traceability inter-UML diagrams” receives the produced file which will be considered as the input to the traceability process.

The latter is composed of traceability rules identification and similarity calculation between the selected fragment in the use case and its corresponding in UML design diagrams (class, sequence, activity and state transition). This process uses the identification of traceability rules and semantic traceability results. The identification of traceability rules explicitly represents the relationships (structural aspect) among the diagrams' elements. It is based on an ontology for the semantic analysis of the textual description template. To identify the semantic traceability between the structured textual documentation and UML design diagrams, traceability process inter-UML diagrams apply the LSI technique.

The third phase is based on the traceability process from requirement to code which apply the traceability rules defined in the first phase on the code and calculate

the similarity between the selected fragment in UC and the code.

3.1 Natural language pre-processing

The most important challenge we are facing when trying to generate the enriched format from the textual description is the complexity of natural language. Consequently, we used natural language processing concepts that are syntax parsing.

The syntax parsing consists in obtaining a structured representation of the software knowledge. Therefore, the software analyst has first to clean the textual description by using the Stanford CoreNLP tool [51] and second to organize it according to a specific template's structure. Stanford CoreNLP tool is used to obtain a more manageable and readable text. The tool relies on the following methods:

- Tokenization is the task of breaking a character sequence up into pieces (words/phrases) called tokens, and perhaps at the same time throw away certain characters such as punctuation marks [52].
- Filtering aims to remove some stop words from the text. Words, which have no significant relevance and can be removed from the documents [53].
- Lemmatization considers the morphological analysis of the words, i.e. grouping together the various inflected forms of a word so they can be analysed as a single item.
- Stemming aims at obtaining stem (root) of derived words. Stemming algorithms are indeed language dependent [54].
- Part of Speech Tagging tags for each word (whether the word is a noun, verb, adjective, etc.), then finds the most likely parse tree for a piece of text.

The cleaned file is then used to identify the goals. By goal, we mean a collection of functionalities that are related to describe a functional process of the software. Each goal will correspond to a textual description of a use case.

To guide and improve the generation of a software in a more systematic way, the software analyst associates to each textual description of a part, a template that is described by a set of linguistic patterns. The template is easy to understand and validated by stakeholders. It covers the semantic, behavioural, functional and organizational information. It is composed of three blocks (See Table 1).

The first block gives an executive summary of the textual description block in terms of the name of the UC, purpose of the use case and actors. The second block describes the main, alternative, and error scenarios. The use case description contains also pre-condition for execution, post-condition (success/failure), and relationships with parts successors. These scenarios respect a linguistic syntax pattern:

```
<NumAction><From Actor><To Actor> <Type of
Result> <Action Description> <In-Parameter><Out-
Parameter> <IsConsidered><IsIgnored> <IsNegative>
```

Table 1 depicts the expanded description template with alternative behavioral elements based on control structures such as IF-THEN statement and iterative

elements, e.g. <for><number of iterations>). In addition, the extended template expresses how the actions are executed: in a parallel <Parallel>, or sequence way <Sequential>, etc. Besides the common elements, we proposed an extension of the UC textual description with behavioral elements and keywords, such as:

- <In-Parameter> and <Out-Parameter> expressing the input and output of the action.
- <Type of Result > which determines if the result has a simple value or it represents an entity. In the case of a simple value, it can be represented as an attribute. However, in the case of an entity, it can be transformed to a class in the class diagram.
- <From Actor><To Actor> which represents the sender and receiver of the action.
- <If><Else if><Else>represents a choice or behaviour alternatives.
- <Parallel> expresses parallel execution of the actions.
- <For> <number of iterations> represents the loop which is repeated a number of times.
- <Loop>: an iterative behaviour that englobes one or several actions.
- <Break> represents an exceptional situation corresponding to a scenario of rupture.
- <Functional Call> is an action that calls another action or use case.
- <IsIgnored> reflecting that the actions types can be considered insignificant and are implicitly ignored.
- <IsConsidered> determines which actions should be considered within this textual description, meaning that any other action will be ignored.
- <IsNegative> describes actions of traces that are defined to be negative (invalid). Negative traces occur when the system has failed. It can represent an exception.

The added behavioral elements and keywords are organized according to the use case scenario. The main scenario contains sequential or parallel actions. It can also contain a functional call; while the alternative and error scenario are based on conditional (opt, If Else, etc.) or iterative (Loop) control structures that can be expressed in one or more levels (nested levels). For instance, it is possible to determine an iterative block nested in a conditional block and vice versa. These control structure types can be followed by parallel or sequence blocs.

<p>Name of the Use Case (UC):<unique name assigned to a use case> Purpose of the use case:< a summary of a UC purpose> Actors:<Primary actor>: actor that initiates the use case> <Secondary actor>: actor that participate within the use case></p> <p>Pre-condition for execution:<A list of conditions that must be true to initialize the UC> Post-condition (success/failure):<state of the system if the goal is achieved/abandoned> Relationships: <include>: <UC in relation with this UC by include> <Extend>: < use cases in relation with this use case by "extend"> <Super use case>: <list of subordinate uses cases of this use case> <Sub use case>: <list of all uses cases that specialize this use case></p> <p>Begin ***Main scenario*** <steps of the scenario to goal> Begin //sequential actions <NumAction><From Actor><To Actor> <Type of Result> <Action Description> <In-Parameter><Out-Parameter> <IsConsidered><IsIgnored></p>
--

<p><IsNegative> //parallel actions Parallel <NumAction><From Actor><To Actor><Type of Result ><Action Description><In-Parameter><Out-Parameter><IsConsidered> <IsIgnored> <IsNegative> <NumAction><From Actor><To Actor><Type of Result ><Action Description><In-Parameter><Out-Parameter> <IsConsidered> <IsIgnored> <IsNegative> // Functional call Functional Call <NumAction><From Actor><To Actor><Type of Result> <Action Description> <In-Parameter> <Out-Parameter> <IsConsidered> <IsIgnored> <IsNegative> End ***Alternative scenario*** SA1 Begin<Event, condition> begin at <Num "action number"> <Return "action number"> List of actions //sequential actions <NumAction><From Actor><To Actor><Type of Result> <Action Description> <In-Parameter><Out-Parameter> <IsConsidered> <IsIgnored><IsNegative> //parallel actions <NumAction><From Actor><To Actor><Type of Result> <Action Description> <In-Parameter><Out-Parameter> <IsConsidered> <IsIgnored><IsNegative> <NumAction><From Actor><To Actor><Type of Result> <Action Description> <In-Parameter><Out-Parameter> <IsConsidered> <IsIgnored><IsNegative> //alternative control structure in the first level <IF><condition> <NumAction><From Actor><To Actor><Type of Result ><Action Description><In-Parameter><Out-Parameter> <IsConsidered> <IsIgnored> <IsNegative> End IF //iterative control structure in the first level <Loop><Min Number of Iterationxfcyws, Max Number of Iterations > <NumAction><From Actor><To Actor><Type of result ><Action Description><In-Parameter><Out-Parameter><IsConsidered> <IsIgnored> <IsNegative> End Loop End SA1 SA2 Begin<Event, condition> begin at <Num "action number"> <Return "action number"> List of actions // Loop nested in an alternative control structures < IF><condition> <NumAction> <From Actor><To Actor><Type of result ><Action Description><In-Parameter><OutParameter><IsConsidered> <IsIgnored> <IsNegative> <Else> <Loop><Min Number of Iterations, Max Number of Iterations > <NumAction> <From Actor><To Actor><Type of result ><Action Description><In-Parameter> <Out-Parameter><IsConsidered> <IsIgnored> <IsNegative> End Loop End IF <Functional Call> <NumAction> <From Actor><To Actor><Type of Result ><Action Description> <In-Parameter> <Out-Parameter> <IsConsidered> <IsIgnored> <IsNegative> End End SA2 ***Error scenario*** SE1// Treat the error and return to the action Begin<Event, condition> begin at <Num "action number"> <Return "action number"> List of actions <NumAction> <From Actor><To Actor><Type of Result ><Action Description><In-Parameter><Out-Parameter><IsConsidered> <IsIgnored> <IsNegative></p>
--

Parameter<<OutParameter><IsConsidered> <IsIgnored> <IsNegative> End SE1 End Use case
Critical situations of execution of the activity Special requirement: <non Functional requirement><Project requirement and constraints>

Table 1: Enriched textual description of a use case.

3.2 Traceability process

In this subsection, we define traceability rules which are applicable to the first and second phase of our method. They are used to determine correspondences between the requirement modeled with the use case diagram based on the enriched textual description and design diagrams modeled with SD, CD, AD and STD.

3.2.1 Traceability rules

R1: For each <In-Parameter> and <Out-Parameter> expressing the input and output of the action, there is:

- **SD:** an object in a sequence diagram.
- **CD:** a class corresponding to each parameter, and an attribute corresponding to an argument.
- **AD:** an object node that corresponds to InputPin and OutputPin. We note that InputPin and OutptPin can be related to the same or more than one objectNode.
- **Code:** a class corresponding to <In parameter> and an attribute corresponding to an argument.

R2: For each action’s sequence in a use case, there is:

- **SD:** a sequence of sent or received message which preserves the action order in the scenario.
- **AD:** a sequence of ordered activities.
- **STD:** a sequence of ordered states in the state diagram. If the action in a STD respects the renaming pattern: « Action verb + DataObject| NominalGroup », then the state of the action will be: Data object + past participle.
- **Code:** a sequence of lines of code that respect the ordered actions.

We note that this rule cannot be expressed in the CD.

R3: For each actor expressing the sender and the receiver of the action in the use case scenario, there is:

- **SD:** an object corresponding to each participant (actor) in the SD.
- **CD:** a class corresponding to each participant in the CD.
- **AD:** a swimlane having the actor name which performs a group of activities.
- **STD:** the actor has no corresponding in the STD.
- **Code:** a class in the code.

R4: For each action in the use case scenario, there is:

- **SD:** a message in a SD having a synonym name.
- **CD:** a method in a class corresponding to the action.
- **AD:** an executable node represented by ‘Action’ having the same name and the same parameters.
- **STD:** If the action in a textual description respects the renaming pattern: « Action verb + Object | Nominal Group », then the state will be : object + past participle.
- **Code:** a method in the code having the synonym name, the same parameters.

R5: For each pre-condition/post-condition of the use case scenario, there is:

- **SD:** a precondition/post-condition of the first message sent by an object in the sequence diagram.
- **AD:** a guard of the corresponding action [55]
- **STD:** a pre-condition associated to a transition which is necessary to define a state.
- **Code:** a precondition under which a method may be called and expected to produce correct results [56].

We note that the precondition and the post-condition have no corresponding in the class diagram.

R6: For each parallel scenario (PARALLEL), there is:

- **SD:** a parallel combined fragment in a sequence diagram.
- **AD:** a set of parallel actions between a fork node and a join node.
- **STD:** a fork pseudo state vertices and a join state.
- **Code:** a multi-threaded program in java.

We note that the parallelism is not expressed in the CD.

R6 is illustrated in Table 2.

R7: For each alternative scenario in a use case where instructions begin with alternative behavioural elements (IF-THEN Statement ELSE Statement), there is:

Use case	Sequence Diagram	Activity Diagram	State transition diagram	Code
PARALLEL < <NumAction><Pre-condition> <From Actor><To Actor><Action Type><Type of Result ><Action Description> <In-Parameter> <Out-Parameter> <IsConsidered> <IsIgnored><IsNegative> <NumAction><Pre-condition> <From Actor><To Actor><Action Type><Type of Result ><Action Description> <In-Parameter> <Out-Parameter> <IsConsidered> <IsIgnored><IsNegative>>				<pre> public class MyClass implements Runnable{ Thread UnThread ; MyClass () { //..initialisation of myClass constructor UnThread = new Thread (this , "thread secondaire"); UnThread.start(); } public void run () { //...second thread actions here } } </pre>

Table 2: R6 illustration.

Use case	Sequence Diagram	Activity Diagram	State transition diagram	Code
<p><IF><condition> <NumAction><Pre-condition> <From actor><To actor><Action Type><Type of result ><Action Description> <In-Parameter> <Out-Parameter> <IsConsidered> <IsIgnored><IsNegative> <Else > <NumAction><Pre-condition> <From actor><To actor><Action Type><Type of result ><Action Description> <In-Parameter> <Out-Parameter> <IsConsidered><IsIgnored> <IsNegative> END</p>				<pre>If (condition) { operation 1; } else { operation 2; }</pre>

Table 3: R7 illustration.

- **SD:** an ALT combined fragment with the interaction operator “ALT” and two alternative interactions in a SD.
- **AD:** a decision node with two outgoing edges with guards in the activity diagram or a conditional node is a structured activity that represents an exclusive choice between two alternatives.
- **STD:** a decision point leading to two different states in the state transition diagram.
- **Code:** a basic control structure corresponding to “IF condition THEN treatment 1 ELSE treatment2”.

R7 is illustrated in Table 3.

R7.1: For each alternative Scenario where instructions begin with the alternative behavioral elements (<if > condition <else if >.....<else if>...<else>...), (SWITCH), there is:

- **SD:** an Alt Combined Fragments: Interaction operator “alt” with more than two alternatives in a SD.
- **AD:** a decision node with more than two outgoing edges in an activity diagram.
- **STD:** a decision point leading to n different states in a STD or a conditional node is a structured activity that represents an exclusive choice among some number of alternatives.
- **Code:** a basic control structure corresponding to switch.

R7.2: For each alternative scenario in a use case where instructions begin with the alternative behavioral elements (<if > condition <then> treatment.), there is:

- **SD:** an opt combined fragment in a sequence diagram. We recall that the opt (optional) operator is a non-alternative (otherwise) test statement.
- **AD:** a decision node with two outgoing edges: one to execute an action and the second is related to the final activity in the activity diagram.
- **STD:** a decision point leading to one state and one final state.
- **Code:** a basic control structure corresponding to “IF condition THEN treatment”.

R7.3: For each alternative scenario in a use case where instructions contain the alternative behavioral elements (<if > condition <break>) in an iterative bloc, there is:

- **SD:** a break combined fragment in a loop fragment that belongs to a sequence diagram
- **AD:** a decision node which one is related to a final activity by an outgoing edge and another outgoing

edge which is related to a final node in the alternative scenario

- **STD:** a decision point leading to 1 state and one final state (Transition to terminate pseudostate).

R7.4: For each error scenario in a use case where instructions begin with the alternative behavioral elements <IF><condition> Return, there is:

- **SD:** a break combined fragment in a sequence diagram which can be used to express an error scenario.
- **AD:** an interruptible region which contains activity nodes in the error scenario
- **STD:** a decision point leading to 1 state and one final state (Transition to terminate pseudostate). A break can be also expressed by an Exit point pseudostate which is an exit point of a state machine or composite state.

The exit point is typically used if the process is not completed but has to be escaped for some error or other issue.

R7.4 is illustrated in Table 4.

R8: For each alternative/error Scenario in a use case where instructions begin with the iterative behavioural elements (<For><[num of iterations]>...), there is:

- **SD:** a loop combined fragment in a sequence diagram.
- **AD:** a decision node with one of the outgoing edges is a precedent activity in an activity diagram.
- **STD:** a reflective transition or transition path.
- **Code:** a basic control structure corresponding to For-do, DO while (post-test), While do (pre-test).

R8 is illustrated in Table 5.

R9: For each functional call (an action that calls another action or use case), there is:

- **SD:** a ref fragment expressing the reference to an interaction in another sequence diagram.
- **AD:** a call Behaviour: An activity is invoked by using the ‘Call Behavior Action’ node, which means that the invoked activity is defined in more details in another AD.
- **STD:** A Composite state which encloses refinements of the given state. We note that the composite state corresponds to the object that can realize the functional call or an entry point of a state machine or composite state which allows you to specify an activity that occurs when you enter the state.

Use case	Sequence Diagram	Activity Diagram	State transition diagram	Code
<IF><condition> <NumAction><Pre-condition> <From actor><To actor><Action Type><Type of result ><Action Description> <In-Parameter> <Out-Parameter> <IsConsidered> <IsIgnored><IsNegative> <Else> return				<pre> If (condition1) operation 1; Else return; </pre>

Table 4: R7.4 illustration.

Use case	Sequence Diagram	Activity Diagram	State transition diagram	Code
<For><Min Number of Iterations, Max Number of Iterations > <NumAction><Pre-condition> <From actor><To actor><Action Type><Type of result ><Action Description> <In-Parameter> <Out-Parameter> <IsConsidered> <IsIgnored> <IsNegative> End For				<pre> For(i=1, i<=5,i++) operation 1; } </pre>

Table 5: R8 illustration.

– **Code:** a call of a class or a method.

R10: For each action that represents an invalid interaction/exception <IsNegative>, there is:

- **SD:** A Negative combined fragment in the sequence diagram which defines invalid traces.
- **AD:** An event representing an error (exception) that interrupts the flow or a break which are most commonly used to model exception handling.
- **STD:** a transition to an error state. This error state may be terminal, i.e. aborts further event handling.
- **Code:** a basic control structure corresponding to “Exception”. This corresponds to a try-catch.

R11: For each action that should be considered (respectively ignored) within the scenario, there is:

- **SD:** messages that are considered as significant (respectively insignificant) within the “consider” (respectively ignore) combined fragment.
- **AD:** considered (respectively ignored) messages are shown in the activity diagram.
- **STD:** The states corresponding to the considered (respectively ignored) actions.
- **Code:** The method should be considered (ignored) as significant in the code.

3.2.2 Similarity calculation

Based on the proposed rules, we apply the similarity measure “Latent Semantic Indexing” (LSI) which is defined to the traceability process inter-UML diagrams and to the traceability process from requirement to code.

The first step in calculating the LSI is to assign term weights and construct the term-document matrix A and query matrix. The m by n document-matrix A is presented as follows where:

$$a_{ij} = w_{ij} = \text{term weights} \quad (1)$$

In the second step, LSI applies singular value decomposition (SVD) to the A matrix which consists in decomposing the A matrix into three matrices: the U, S and V. One component matrix describes the original row entities as vectors of derived orthogonal factor values, another describes the original column entities in the same way, and the third is a diagonal matrix containing scaling values such that when the three components are matrix-multiplied, the original matrix is re-constructed. The third step represents the dimensionality reduction, which consists in computing U_k , S_k , V_k and V_k^T . For instance, implementing a rank 2 Approximation ($K=2$) by keeping the first two columns of U and V and the first two columns and rows of S. The fourth step consists in finding the new document vector coordinates in this reduced 2-dimensional space. Rows of V hold eigenvector values. These are the coordinates of individual document vectors. The fifth step finds the new query vector coordinates in the reduced 2-dimensional space as follows:

$$q = q_T U_k S_k^{-1} \quad (2)$$

Finally, the last step ranks documents in order to decrease the order of query-document cosine similarities using the following equation:

$$\text{sim}(q,d) = \frac{q \cdot d}{|q| |d|} \quad (3)$$

The document which has a higher score is closer to the query vector than the other vectors.

We note that, in this paper, LSI is used to compute similarities between the selected fragment in a use case and the corresponding ones in other UML diagrams (SD, CD, AD and STD), and then the corresponding fragment in the code while in [6] the LSI is used only to compute similarities between actions in UC and messages in sequence diagrams. The choice of LSI amongst other similarity measures is justified by its capacity in retrieving hidden, semantic relations between terms when searching

for similar terms between queries extracted from a fragment in the UC and the documents containing the information in other UML diagrams. In fact, LSI does not rely on words but rather on concepts; that is, words having same contexts can be revealed similar. This propriety expresses the difference between LSI and other IR techniques. Henceforth, the similarity measure can be properly calculated between queries and documents even when they do not share enough words.

4 Traceability evaluation

The evaluation phase expresses the performance of the proposed method revealed by two steps: experimental evaluation and result interpretation. The first step in the evaluation phase compares corresponded elements generated by our method with the corresponded elements where traceability is evaluated by experts. Particularly, we present two UML projects containing a set of UML diagrams (including use cases and their textual descriptions) and the source code (projects are implemented using JAVA language) to five experts having years of experience studying and developing UML projects. The expert should determine traceability by detecting the corresponding elements. The solution presented by these experts was compared with our solution (constructed by our tool). The projects source codes are available as well as their design (i.e. UML diagrams). Table 6 provides some information about these projects. Besides, for experimental evaluation purposes, we refer to the recall and precision measures:

$$\text{Precision} = \text{TP}/(\text{TP}+\text{FP}) \quad (4)$$

$$\text{Recall} = \text{TP}/(\text{TP}+\text{FN}) \quad (5)$$

where:

- True positive (TP) is the number of existing real corresponded elements generated by our tool;
- False Positive (FP) is the number of non existing real corresponded elements generated by our tool;
- False Negative (FN) is the number of existing real corresponded elements not generated by our tool.

4.1 Evaluation results and interpretation

High scores for both ratios show that our traceability approach returns both accurate corresponding elements of UML diagram (high precision) and the majority of all relevant corresponding elements (high recall). It means that the generated traceability links cover the whole domain precisely in accordance to the experts' perspective.

As illustrated in Table 7, precision, whose average is 0.84, indicates that we found some false positive corresponding elements (i.e. incorrect detected corresponding elements). The false positives corresponding elements are not significant value when we compare them to the true positives found by our method. The recall, whose average value is 0.91, expresses that there are also some false negatives corresponding elements (i.e. true corresponding elements are not detected). These false negatives can be explained by the fact that our method uses "threads" to detect parallelism in

the source code however parallelism in JAVA can be implemented using different ways (fork/join framework, threads, Agregate operations, etc.). Source code in the used projects uses the aggregate operations and parallel streams to express parallelism and our method uses threads to detect parallelism. This is why parallel fragments are not traced and we found some false negatives.

The true positives and the false negatives are equal to the total number of actual corresponding elements. All the false negatives are corresponding elements associated to elements in UC textual description diagram that have corresponding impacts on other UML diagrams which are not detected.

4.2 Threats to validity

This section discusses the potential issues that may threaten the validity of our study, including the internal and external validity [57].

The internal validity threats in the case of traceability identification are related to user requirements [58]. They are related to three issues: The first issue is due to the use of the enriched textual description of a use case which may not always be available. The second problem is addressed when there is a diversity of requirements description. In this case, which one can be used to describe the functional requirements?

Furthermore, if the functional requirements are clearly stated, then our method generates well matched elements; otherwise, the quality of the derived traceability elements is not guaranteed in terms of dependencies between elements. The third issue is related to the impact of an error-prone generation of UML diagrams and code. This case may lead to inconsistency between the requirement, design models and source code.

The external validity threats deal with the possibility to generalize this study results to other case studies. The limited number of case studies used to illustrate the proposed approach could not generalize the results. In addition, the traceability between all levels increases the detection and localization of consistency errors.

5 TRADIAC tool

To facilitate the application of our method, we have developed a tool for determining the traceability at different abstraction levels, named TRADIAC Quality (TRAcability for UML DIAGrams and Code). Our tool is implemented as an Eclipse™ plug-in [59]. It is composed of four main modules (see Figure 2): Pre-processing Natural language, Traceability inter-UML diagrams, Traceability from requirement to code, and traceability evaluator.

5.1 Pre-processing natural language module

The pre-processing engine is composed of the cleaner and the XML generator.

PROJECT NAME	#Use cases	# CLASSES	# METHODS	KLOC
Car rental system	9	98	252	108
Customer Relationships system	7	65	124	96

Table 6: Characteristics of the studied projects.

Evaluation Measures	TP	FP	FN	Precision= $\frac{TP}{TP+FP}$	Recall= $\frac{TP}{TP+FN}$
Results	62	9	5	0.84	0.91

Table 7: Evaluation results.

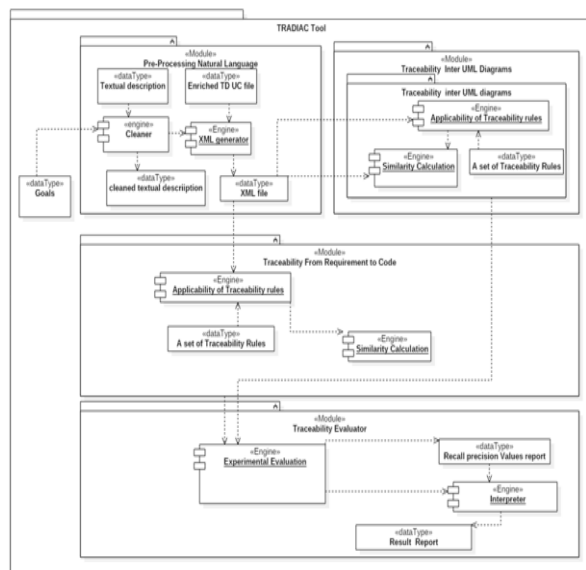


Figure 2: Software architecture of TRADICAC Quality Tool.

Goals: The purpose is to make a reservation by a customer from a car rental branch.

Textual description: The use case begins when a customer decides to make a reservation and introduce himself in the car rental branch to an available Clerk. The clerk asks the customer for his/her ID and introduces it.

The system checks if the customer is a person who has had contact with EU-Rent. If he/she exists, the system verifies that the customer is not in the black list otherwise it introduces a new EU-rent customer/driver. The clerk introduces the reservation ID, the period desired and countries planned to visit. He specifies and verifies the period validity and that there is no overlap with other customer reservations and 3) the availability of the specified car model for the period indicated.

If there are no cars to rent corresponding to the desired model in the selected period, the system displays an error message to the user and suggests if it is possible to change the reservation period or the car type. The clerk asks the customer to validate the reservation.

If the customer validates the reservation, the clerk creates the reservation agreement and offers a discount to the customer. The rental is confirmed and a new rental agreement is created with the indicated parameters.

Figure 3: Goal definition.

5.1.1 Cleaner

The cleaner uses as input the textual description of the software written in a natural language. It cleaned the file using the Stanford CoreNLP tool. The cleaned file is used by the software analyst to define manually goals. Then, the latter associates each goal to its corresponding textual description part.

In order to illustrate the functioning of this module, we apply it to the “make a reservation” textual description. For instance, Figure 3 illustrates the goal definition and its description. The software analyst creates the enriched template corresponding to each textual description part. Table 8 illustrates enriched textual description for the use case “UC-ETD” “make a reservation” from a car rental system [60].

5.1.2 XML generator

XML generator takes as input the enriched textual description of UC introduced by the user. The purpose interface of “UC-ETD” is presented in Fig. 4. It is composed respectively of five tabs illustrating the identification purposes “identification purpose”, the nominal scenario “Main Scenario”, the alternative scenario (s) “Alternative Scenario”, the error scenario (s) “Error scenario (s)” and the generator of the XML file corresponding to the textual description. The “identification purpose” tab contains the name of the UC, its purpose, the primary and secondary list of the actors, the pre-condition and the post-condition of the UC in the textual description and the use case’s relationships: include, extend and generalize. The list expresses use cases in relation with the corresponding one by “include”, use cases in relation with the corresponding use case by “extend”, subordinate uses cases of the super UC and the list of all uses cases that specialize the sub use case. The three other tabs express the details of the different UC scenarios being documented. The last tab expresses the XML file corresponding to the textual description of the whole UC. In the rest of the section, we detail these tabs through the use case “make a reservation” from the case study “Car Rental” [60]. The enriched textual description for the use case “make a reservation” is presented in Table 8 describing the purpose (See Figure 4) of the UC, Figure 5, Figure 6 and Figure 7 presenting respectively the main, alternative and error scenarios, and Figure 8 illustrates the corresponding XML file based on the enriched template of the “make a reservation” UC.

- Addition a nominal scenario NS: The “Main Scenario” (see Figure 5) shows the list of actions in the main scenario which can be classified on two blocs: sequential or parallel actions. Each bloc indicates how these actions are executed. It is composed of seven columns representing respectively: a) NumAction that indicates an automatic number identifying an action, b) Fom actor and c)To actor which allows to specify who is responsible for the action, d)Type of result which determines if the result is a simple value or it represents an entity, and e)Action description representing a field specifying the action text, f) In-

<p>Name of the UC : <Make a reservation > Purpose: <A customer makes a reservation from an EU-Rent branch > Principal Actors : < Clerk > , <Secondary actor> : < Customer></p> <hr/> <p>Pre-condition for execution:< when a customer decides to make a reservation and inform the clerk> Post-condition (success): < The rental is confirmed and a new rental agreement is created with the indicated characteristics> Post-condition (failure): <The indicated characteristics are not satisfied and a rental is canceled> Relationships: <include>: < Offer discount>< Offer special advantages> <Extend>: < -- >; <Super use case>: < -->; <Sub use case>: <--> Begin</p> <p style="text-align: center;">***Normal scenario***</p> <p><steps of the scenario of the trigger to goal> Begin NS <NumAction 1> < From Actor Clerk >< To Actor Customer> < Type of Result Simple: Integer> < Action Description Asks the customer for hisID > <In-Parameter: IDCustomer > <Out-Parameter IDCustomer > < IsConsidered 1>< IsIgnored 0>< IsNegative 0> <NumAction 2> < From Actor The Clerk >< To Actor The Customer > < Type of Result Simple: Boolean>< Action Description Checks if the customer is a person who had contact with EU-Rent> <In-Parameter IDcustomer > <Out-Parameter Exists: Yes> < IsConsidered 1>< IsIgnored 0>< IsNegative 0> <NumAction 3>< From Actor Customer> < To Actor clerk>< Type of Result Entity>< Action Description Tells information about the reservation to the clerk><In-Parameter Reservation (IDRes,StartResDat, End ResDat, DepartureCity, Arrival city)> <Out-Parameter Reservation (IDRes , StartResDat, End ResDat, Departure/Arrival City, registration number car) >< IsConsidered 1>< IsIgnored 0><IsNegative 0> <NumAction 4> < From Actor Clerk >< To Actor The reservation> < Type of Result Entity>< Action Description Introduces the reservation ID, the period desired and countries planned to visit ><In-Parameter IDRes, StartResDat, End ResDat, DepartureCity, Arrival city, registration number car ><Out-Parameter Reservation (IDRes, StartResDat, End ResDat, Departure/ArrivalCity, registration number car) >< IsConsidered 1>< IsIgnored 0>< IsNegative 0> <NumAction 5> < From Actor The Clerk >< To Actor The reservation><Type of Result Simple: boolean><Verify that the period is correct, that there is no overlap with other reservations of the customer and the availability of the specified car model for the period indicated> <In-Parameter: Reservation> <Out-Parameter availability car > < IsConsidered 1>< IsIgnored 0>< IsNegative 0> <Parallel> <NumAction 6>< From Actor The Clerk >< To Actor The agreement> < Type of Result Entity>< Action Description Create the reservation agreement ><In-Parameter rental agreement: ID customer, price, ID reservation> <Out-Parameter Rental agreement >< IsConsidered 1>< IsIgnored 0>< IsNegative 0> <NumAction 7> < From Actor The Clerk >< To Actor The agreement>< Type of Result Entity>< Action Description Offer a discount to the customer ><In-Parameter Discount rental agreement : Discount,ID agreement, ID customer> <Out-Parameter Discount rental agreement >< IsConsidered 1>< IsIgnored 0>< IsNegative 0> End</p> <p style="text-align: center;">***Alternative scenario***</p> <p>AS1 Begin <Event, begin at Num 2 in SN> <IF>< the customer does not exists > <NumAction 1> < From Actor Clerk >< To Actor customer> < Type of Result Customer (name, ID, birthdate, address, phone)>< Action Description Introduce a new customer> <In-Parameter Customer (name, ID, birthdate, address, telephone)> <Out-Parameter Customer > < IsConsidered 1>< IsIgnored 0>< IsNegative 0> <Else > <restart at num 3 in SN> End AS1</p> <p>AS2 <begin at Num 3 in SN> <Do> <NumAction 1>< From Actor clerk ><To Actor reservation><Type of Result Simple>< Action Description Specifies the period ><In Parameter period > <Out-Parameter period >< IsConsidered 1>< IsIgnored 0>< IsNegative 0> <NumAction 2> < From Actor clerk><To Actor reservation> <Type of Result correct yes/no>< Action Description Verify the period>] [<In-Parameter period>] [<Out-Parameter correct yes/no >]< IsConsidered 1>< IsIgnored 0> < IsNegative 0> <While><period is correct &does not overlaps with other reservations > <restart at Num “6” in SN> End AS2</p> <p>AS3 <begin at Num 5 in SN> <Opt><needs confirmation> <NumAction 6> < From Actor Clerk ><To Actor Customer><Type of Result Simple: Boolean > <Action Description Asks the customer if he validates the reservation> <In-Parameter Reservation><Out-Parameter IsValidated > < IsConsidered 1>< IsIgnored 0>< IsNegative 0> <restart at Num “6” in SN> End AS3</p> <p>*** Error scenario ***</p> <p>ES1 <begin at Num “5”> <IF><There is no cars to rent having the desired model in the selected period> <NumAction 6> < From Actor The system >< To Actor The Clerk >< Type of Result Entity >< Action Description displays an error message to the user and suggests if it is possible to change the reservation period or the car type ><In-Parameter: Car model, period > <Out-Parameter Error message> < IsConsidered 0> < IsIgnored 0> < IsNegative 1> Return End</p>
--

Table 8: Enriched textual description for the use case “make a reservation”.

Enriched Textual Description

Purpose | Main Scenario | Alternative Scenario(s) | Error Scenario(s) | Generate XML

Name: make a reservation

Purpose: A customer makes a reservation from an EU-Rent branch

Principal Actor: Clerk

Secondary Actor: Customer

Pre Condition: When a customer decides to make a reservation and inform the clerk

Relationships

- Generalize**: <-->
- Include**: < Offer discount>, < Offer special advantages>
- Extend**: <-->

Figure 4: UC-ETD “make a reservation” purpose interface.

N° Action	From Actor	To Actor	Type of Results	In-Parameters	Out-Parameters
5	The Clerk	The reservation	Simple: boolean	Reservation	availability car

Action Description: Verify that the period is correct, that there is no overlap with other reservations of the customer and the availability of the specified car model for the period indicated

Is Considered Is Ignored Is Negative

Parallel Actions

Add Action

N° Action	From Actor	To Actor	Type of Results	In-Parameters	Out-Parameters
6	The clerk	The agreement	Entity	ement: ID customer, price, ID reservation	Rental agreement
7	The Clerk	The agreement	Entity	ent: Discount, ID agreement, ID customer	Discount rental agreement

Action Description: Create the reservation agreement

Action Description: Offer a discount to the customer

Is Considered Is Ignored Is Negative

Post condition: a new rental agreement is created with the indicated characteristics 0 / 0

Add Parallel Actions **Add Sequential Actions**

Figure 5: Main scenario of the "make a reservation" use case.

Parameter expressing the input of the action g)Out Parameter expressing the output of the action, and h) boolean value corresponding to each state of the action that can be Considered, IsIgnored, IsNegative. To add a nominal scenario in a specified bloc, click

- on the "Add Parallel Actions" button or "Add Sequential Actions" in the corresponding bloc.
- Addition of an alternative scenario and /or errors: Figure 6 and Figure 7 illustrate, respectively, the alternative and error scenarios. Each alternative or error scenario is composed of two blocks where the

The screenshot shows the 'Enriched Textual Description' window with the 'Alternative Scenario(s)' tab selected. The 'Alternative Scenario's Title' is 'AS1'. Under the 'Event' section, the 'Condition' is 'the customer does not exists'. The 'Action's Number' is 2 and the 'Return Action's Number' is 3. The 'List of Actions' section has buttons for 'Sequential', 'Parallel', 'Iterative Control Structure', and 'Conditional Control Structure'. The 'AS1(Sequential)' section contains an 'Add Action' button and a table with the following data:

N° Action	From Actor	To Actor	Type of Results	In-Parameters	Out-Parameters
1	The Clerk	The customer	Entity Customer	(name, ID, birthdate, address, telephone)	Customer (name, ID, birthdate, address, telephone)

The 'Action Description' is 'Introduce a new costumier'. There are checkboxes for 'Is Considered' (checked), 'Is Ignored', and 'Is Negative'. At the bottom, there are buttons for 'Add Iterative Control Structures' and 'Add Conditional Control Structures'.

Figure 6: Alternative scenario of the "make a reservation" use case.

The screenshot shows the 'Enriched Textual Description' window with the 'Error Scenario(s)' tab selected. The 'Error Scenario's Title' is 'ES1'. Under the 'Event' section, the 'Condition' is 'There is no cars to rent having the desired model in the selected period'. The 'Action's Number' is 5 and the 'Return Action's Number' is empty. The 'List of Actions' section has an 'Add Action' button and a table with the following data:

N° Action	From Actor	To Actor	Type of Results	In-Parameters	Out-Parameters
6	The system	The Clerk	Entity	Car model, period	Error message

The 'Action Description' is 'displays an error message to the user and suggests if it is possible to change the reservation period or the car type'. There are checkboxes for 'Is Considered' (checked), 'Is Ignored', and 'Is Negative'. At the bottom, there is a button for 'Add Error Structures'.

Figure 7: Error scenario of the "make a reservation" use case.

user enters the following information: The first bloc contains the scenario title, guard condition of the event triggering the scenario, the start action at the alternative scenario level and the return action number if it exists. We note that the alternative scenario may contain conditional and/or iterative control structures. In addition, it is possible to depict nested blocs. For instance, an iterative control structures can be nested in an alternative control structures and vice versa. Besides, each bloc includes the list of actions executed in a parallel or sequential way. For each action, the user enters the corresponding information as presented in the nominal scenario (from a to h). To add a new alternative scenario, click on the "Add Conditional Control Structures" button or "add Iterative Control Structures". Similarly, to add an error scenario, click on the "Add Error Structures" button.

After entering the enriched textual description of the make a reservation use case, the XML generator module produces the XMI document as illustrated in Figure 8. To generate the XML file corresponding to the obtained XMI document, this module uses the standard template of Star UML definition. For example, we present as follows the generated XML file corresponding to the documentation of the "Make a reservation" use case of our "Car rental" case study.

5.2 Traceability process module

The traceability module is composed of two engines: applicability of traceability rules and calculation similarity.

5.2.1 Applicability of Traceability rules

In the traceability detection module, the user firstly imports the UML project. In this step, the designer chooses a UC from a list of use cases. Then, the designer can choose a specific fragment to be traced from the selected UC. The presented fragments represent specific

concepts which we added in the UC textual description (e.g., parallel, sequence, loop, conditional, break, etc.). For instance, the designer needs to trace the 'parallel' fragment in the enriched textual description by checking the list of parallel fragments which are available in a list box as shown in Figure 9. Next, we apply the similarity measure LSI between the XML of the selected parallel fragment and the related UML diagrams; and the source code based on the defined traceability rules.

5.2.2 Similarity calculator

The similarity calculator uses the XML files to determine the traceability inter-UML diagrams where the module computes the similarity between the selected fragment and other UML diagrams (CD, AD, STD and SD), and the traceability between UML diagrams and code where the module detects the corresponding elements between the UML diagrams and the code.

We offer to the designer a pairwise traceability (two by two) from the use case diagram to the other diagrams. For instance, when the designer chooses Use case-sequence, the system calculates the similarity between the parallel fragment in the use case diagram and each parallel fragment in the sequence diagrams. To end this purpose, the similarity calculator determines the score of resemblances between the fragment elements in the enriched description and all the corresponding parallel fragments in the sequence diagram (i.e., actor/action in a use case diagram and object/message in the sequence diagrams).

The fragment having a higher score is considered as the most similar one. To decide upon the obtained score value, the constant threshold of 0.70 is widely used in the literature [17]. Consequently, we assume that a similarity value greater than or equal to 0.7 indicates a high similarity between fragments. Otherwise, the designer should verify the quality of the corresponding UML-diagrams. Besides, we calculate in the same manner the similarity between the selected fragment in the use case and the source code.

Figure 8: The generated XML file corresponding to the documentation of the "Make a reservation" use case.

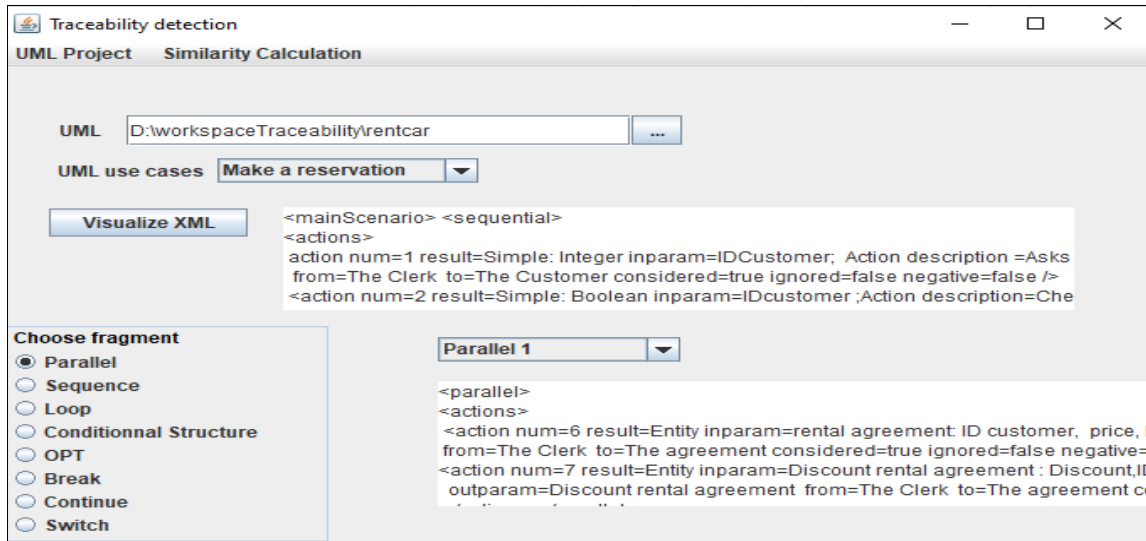


Figure 9: Traceability detection interface.



Figure 10: Traceability inter-UML Diagram.

Table 9 presents the correspondence between control structure fragments (OPT, ATL, WHILE, BREAK, ...) in the use case, activity, sequence, state transition diagram and code. Figure 10 shows traceability between the selected parallel fragment in the main scenario which includes the 6th and the 7th actions in the use case “make a reservation” and its corresponding one in the sequence, activity, class and state transition diagrams as well as the source code.

6 Conclusion

In this paper, we proposed a new method that determines the traceability at different abstraction levels. The traceability is based on the mapping between an enriched textual description of a use case and UML diagrams (class, sequence, activity and state transition diagrams) and between UML diagrams and the code. This correspondence is focused on the control structures defined in the use case textual description and the combined fragment used in the sequence diagrams.

In our future works, the following points will be taken into consideration:

- Representing data in textual description to derive directly the object and class diagram.

- Studying the possibility to derive the implementation diagrams from textual description.
- Determine the traceability from code to functional requirements based on the code-Requirement Traceability Matrix (CRTM) information.

References

- [1] Y. Wang, Formal description of the UML architecture and extendibility, in: journal L’object: Software, Databases, Networks, 2000, Vol.6, No.3.
- [2] P. Rempel, P. Mader, Continuous Assessment of Software Traceability, in: 38th IEEE/ACM Conference on Software Engineering Companion, May, Austin, TX, USA, 2016, pp. 747-748, DOI: <http://dx.doi.org/10.1145/2889160.2892657>
- [3] L. Briand, D. Falessi, S. Nejati, M. Sabetzadeh, T. Yue, Traceability and SysML Design Slices to Support Safety Inspections: A Controlled Experiment, Simula Research Laboratory, in: journal of ACM Transactions on Software Engineering and Methodology, February, 2014, No.9. <https://doi.org/10.1145/2559978>

- [4] A. Lawgali, Traceability of unified modeling language diagrams from use case maps, in: international Journal of Software Engineering & Applications (IJSEA), Vol.7, No.6, November, 2016, pp.89-100. doi:10.5121/ijsea.2016.7607
- [5] V. Adhav, D. Ahire, A. Jadhav, D. Lokhande, Class Diagram Extraction from Textual Requirements Using NLP, in: Second International Conference on Computer Research and Development, (2015), vol.17, No 2, pp. 27-29. DOI: 10.1109/ICCRD.2010.71
- [6] D. Kchaou, N. Bouassida, H.Ben-Abdallah, Uml models change impact analysis using a text similarity technique. In journal of IET Software, Vol 11, Issue 1, No 2, February, 2017, pp. 27-37. DOI: 10.1049/iet-sen.2015.0113
- [7] P. Mader, O. Gotel, Towards automated traceability maintenance, in: Journal of Systems and Software, vol. 85, no. 10, 2012, pp. 2205–2227. <https://doi.org/10.1016/j.jss.2011.10.023>
- [8] S. Nejati, M. Sabetzadeh, C. Arora, L.C.Briand, F.Mandoux, Automated change impact analysis between sysml models of requirements and design, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, USA, 2016, pp 242-253. <https://doi.org/10.1145/2950290.2950293>
- [9] Min, H.S.: 'Traceability Guideline for Software Requirements and UML Design'. in: Journal of Software Engineering and Knowledge Engineering, 26, (01), 2016, pp. 87-113.
- [10] M. Rahimi, J. Cleland-Huang, Evolving software trace links between requirements and source code, in: international journal of Empirical Software Engineering, Vol. 23, 2018, pp.2198–2231. DOI: <https://doi.org/10.1007/s10664-017-9561-x>
- [11] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia and E. Merlo, "Recovering traceability links between code and documentation," in IEEE Transactions on Software Engineering, vol. 28, no. 10, pp. 970-983, Oct. 2002, doi: 10.1109/TSE.2002.1041053.
- [12] A. Ghabi, A. Egyed, Exploiting traceability uncertainty among artifacts and code, The Journal of Systems and Software, Vol. 108, October 2015, pp. 178–192. <http://dx.doi.org/10.1016/j.jss.2015.06.037>
- [13] A. Ghannem, H. Mohamed Salah, M. Kessentini, H.A. Hany, Search-Based Requirements Traceability Recovery: A Multi-Objective Approach, in: IEEE Congress on Evolutionary Computation (CEC), San Sebastian, Spain, 5-8 June, 2017, DOI: 10.1109/CEC.2017.7969440
- [14] C. Mills, C., J. Javier Escobar-Avila, S. Haiduc, Automatic Traceability Maintenance via Machine Learning Classification, in: IEEE International Conference on Software Maintenance and Evolution, Madrid, Spain, 2018, pp. 369-380. DOI: 10.1109/ICSME.2018.00045
- [15] S. Palihawadana, C. H. Wijeweera, M. G. T. N. Sanjitha, V. Liyanage, I. Perera, D. Meedeniya, Tool support for traceability management of software artefacts with DevOps practices, in: Proceedings of the Moratuwa Engineering Research Conference, IEEE, 2017, pp. 129-134. DOI: 10.1109/MERCon.2017.7980469
- [16] C. Trubiani, A. Ghabi, A. Egyed, Exploiting traceability uncertainty between software architectural models and extra-functional results, in: Journal of Systems and Software, Vol 125, March 2017, , 2017, pp.15-34. <https://doi.org/10.1016/j.jss.2016.11.032>
- [17] A. D. Lucia, , F.Fasano, , R.Oliveto, , G.Tortora, Recovering traceability links in software artifact management systems using information retrieval methods, in: ACM Transactions on Software Engineering and Methodology, Vol 16, No4, 2007, pp.13-63. <https://doi.org/10.1145/1276933.1276934>
- [18] M. Lormans, A. van Deursen, Can LSI help Reconstructing Requirements Traceability in Design and Test? In: Proceedings of the 10th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2006, pp. 47-56. DOI: 10.1109/CSMR.2006.13
- [19] A. Marcus, and J. I. Maletic, Recovering documentation-to-source-code traceability links using latent semantic indexing, in: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, Washington, USA, May, 2003, pp.125–135.
- [20] M. Eyl, C. Reichmann, and K. Müller-Glaser, Traceability in a Fine Grained Software Configuration Management System, in: international conference on software quality, LNBP 269, 2017, pp. 15–29. DOI: 10.1007/978-3-319-49421-0_2
- [21] A. Shanmugathan, S., Ratnavel, S., Thiagarajah, V., Perera, I., Meedeniya, D., Balasubramaniam, D.: 'Support for traceability management of software artefacts using Natural Language Processing'. Moratuwa Engineering Research Conf., 2016. pp. 18-23.
- [22] M. Grechanik, K.S. McKinley, D.E. Perry, Recovering and using use-case-diagram-to-source-code traceability links, in: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, September, 2007, pp.95–104, <https://doi.org/10.1145/1287624.1287640>
- [23] A. Cockburn, *j. Highsmith*, Agile Software Development: The People Factor. IEEE Computer, Volume 34, 2001, pp. 131-133.

- [24] O. S. Dawood, A. E. K. Sahraoui, From Requirements Engineering to UML using Natural Language Processing – Survey Study. *European Journal of Engineering Research and Science*, 2, (1), January, 2017, pp. 44-50.
- [25] K. Swathine, N. Sumathi, Study on Requirement Engineering and Traceability Techniques in Software Artefacts, in: *international Journal of Innovative Research in Computer and Communication Engineering*, Vol. 5, Issue 1, January 2017. DOI: 10.15680/IJIRCC.2017. 0501016
- [26] H. Kaiya, A. Hazeyama, S. Ogata, T. Okubo, N. Yoshioka, H. Washizaki, Towards A Knowledge Base for Software Developers to Choose Suitable Traceability Techniques, in: *Proceedings of the 23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems, 2019*, pp. 1075-1084, <https://doi.org/10.1016/j.procs.2019.09.276>
- [27] H. Kaiya, R.Satoa, A.Hazeyamab, S.Ogatac, T.Okubod, T.Tanakae, N.Yoshiokaf, H. Washizakig, Preliminary Systematic Literature Review of Software and Systems Traceability, in: *2th International Conference on Knowledge Based and Intelligent Information and Engineering of the 10th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2006*, pp. 47-56. DOI: 10.1109/CSMR.2006.13
- [28] C. Trubiani, A. Ghabi, A. Egyed, Exploiting traceability uncertainty between software architectural models and extra-functional results, in: *Journal of Systems and Software*, Vol 125, March 2017, , 2017, pp.15-34. <https://doi.org/10.1016/j.jss.2016.11.032>
- [29] S. Maro, A. Anjorin, R. Wohlrab, J.P. Steghöfer: Traceability maintenance: factors and guidelines, in: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, September 3-7, 2016*, pp. 414-425.
- [30] S. Maro, J.P. Steghöfer, M. Staron, Software traceability in the automotive domain: Challenges and solutions, in: *Journal of Systems and Software, Vol 141, 2018*, pp.85-110. <https://doi.org/10.1016/j.jss.2018.03.060>
- [31] R.Wohlrab, J.-P. Steghöfer, E. Knauss, S. Maro, A. Anjorin: Collaborative Traceability Management: Challenges and Opportunities, in: *24th IEEE International Requirements Engineering Conference, Beijing, China, September 12-16, 2016*, pp. 216-225. DOI: 10.1109/RE.2016.1
- [32] M., Broy, A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability - from requirements to functional and architectural views, in: *journal of Software and Systems Modeling*, pp.365-393, Vol.17, Issue 2, 2018, pp. 365-393. <https://doi.org/10.1007/s10270-017-0619-4>
- [33] Yazawa, Y. , Ogata, S., Okano, K., Kaiya, H., Washizaki, H.: 'Traceability Link Mining - Focusing on Usability'.41st IEEE Annual Computer Software and Applications Conference, Italy, 2, 2017, pp 286-287.
- [34] K.S. Divya, R. Subha, , S. Palaniswami, Similar words identification using naive and tf-idf method'. *Information Technology and Computer Science Journal*, pp. 42-47, 2014.
- [35] Kothari, P.R.: 'Processing Natural Language Requirement to Extract Basic Elements of a Class'. *Journal of Applied Information Systems*, USA 3, (7), 2012, pp. 39-42.
- [36] A. T. Imam, A. A. Hroob , R. A. Heisa, The use of artificial neural networks for extracting actions and actors from requirements document'. *journal of Information and Software Technology*, 2018, pp.1-15.
- [37] Rath, M., Rendall, J., Guo, J. L.C., Cleland-Huang, J., Mader, P., 'Traceability in the Wild: Automatically Augmenting Incomplete Trace Links'. *ICConf on Software Engineering*, May 27-June 3, Sweden, 2018, pp. 834–845.
- [38] L. G. P. Murta, A. van der Hoek, C. M. L. Werner, Archtrace: policy-based support for managing evolving architecture-to implementation traceability links, in: *21st IEEE/ACM International Conference on Automated Software Engineering, Tokyo, Japan, 2006*, pp. 135–144. DOI: 10.1109/ASE.2006.16
- [39] L. G. P. Murta, A. van der Hoek, C. M. L. Werner, Continuous and automated evolution of architecture-to-implementation traceability links, in: *Automated Software Engineering Journal*, vol. 15, no. 1, 2008, pp. 75–107. <https://doi.org/10.1007/s10515-007-0020-6>
- [40] I. D. D. Rubasinghe, A. Meedeniya, I. Perera, Towards Traceability Management in Continuous Integration with SAT Analyser, in: *Proceedings of the 3rd International Conference on Communication, and Information Processing, 2017, ACM, Tokyo*. DOI: 10.1145/3162957.3162985
- [41] I, Pete, D., Balasubramaniam, Handling the Differential Evolution of Software Artefacts A Framework for Consistency Management, in: *22nd IEEE International Conference on Software Analysis Evolution and Reengineering, 2015*, pp.599-600, doi: 10.1109/SANER.2015.7081889
- [42] M. Grechanik, KS. McKinley, DE. Perry, Recovering and using use-case-diagram-to-source-code traceability links, in: *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, September,*

- 2007, pp.95–104,
<https://doi.org/10.1145/1287624.1287640>
- [43] Guo, J., Cheng, J. Cleland-Huang, J.: 'Semantically Enhanced Software Traceability Using Deep Learning Techniques'. Conf on Software Engineering, May 2017, pp. 3-14.
- [44] Kuang, H., Nie, J., Hu, H., Rempel, P., Lü, J., Egyed, A., Mäder, P. : 'Analyzing Closeness of Code Dependencies for Improving IR-Based Traceability Recovery'. Software Analysis Evolution & Reengineering, 2017, pp. 68-78.
- [45] Kuang, H., Gao, H., Hu, H., Ma, X. , Lü, J., Mäder, P. , Egyed, A.: 'Using Frugal User Feedback with Closeness Analysis on Code to Improve IR-Based Traceability Recovery'. IEEE/ACM 27th Inter. Conf. on Program Comprehension, pp. 369-379, 2019.
- [46] I. D. D. Rubasinghe, A. Meedeniya, I. Perera, Towards TraceabilityManagement in Continuous Integration with SAT Analyser, in: Proceedings of the 3rd International Conference on Communication, and Information Processing, 2017, ACM, Tokyo. DOI: 10.1145/3162957.3162985
- [47] I. D. D. Rubasinghe, A. Meedeniya, I. Perera, Software Artefact Traceability Analyser: A Case-Study on POS System, in: Proceedings of the 6th International Conference on Communications and Broadband Networking, February 24 - 26, 2018, pp.1-5, DOI:10.1145/3193092.3193094
- [48] H. Tufail, M. F. Masood, B. Zeb, F. Azam, A Systematic Review of Requirement Traceability Techniques and Tools, in: 2nd International Conference on System Reliability and Safety (ICSRS), 20-22 December, Milan, Italy, 2017, DOI: 10.1109/ICSRS.2017.8272863.
- [49] O. Rahmaoui, K.Souali, M. Ouzzif, Improving Software Development Process using Data Traceability Management, in: international Journal of Recent Contributions from Engineering, Science & IT, 2019, pp.52-58.
<https://doi.org/10.3991/ijes.v7i1.10113>.
- [50] K. Souali, O. Rahmaoui, M. Ouzzif, An overview of traceability: Definitions and techniques. 4th IEEE Colloquium on Information Science and Technology, Morocco, October, 2016, pp.789-793.
- [51] C.D Manning, M. Surdeanu, J. Bauer, J.R Jenny Rose Finkel, S. Bethard, D. McClosk, The Stanford CoreNLP Natural Language Processing Toolkit. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, June 22-27, 2014, pp.55-60.
- [52] J.J. Webster, C. Kit, Tokenization as the initial phase in nlp. In *Proceedings of the 14th conference on Computational linguistics*, Association for Computational Linguistics, Volume 4, 1992, pp. 1106-1110.
- [53] H. Saif, M. Fernandez, Y. He, H. Alani, On stopwords, filtering and data sparsity for sentiment analysis of twitter'. In LREC'14 Proceedings of the Ninth International Conference on Language Resources and Evaluation, European Language Resources Association, Reykjavik, Iceland, May 26-31, 2014, pp. 810-817.
- [54] J.B. Lovins, Development of a stemming algorithm. In *Mechanical Translation and Computational Linguistics*, Vol 11, No.1-2, March, June, 1968, pp. 22-31.
- [55] OMG-UML :OMG-UML, 2015. *OMG Unified Modeling Language (OMG UML)*. formal/2015-03-01. [Online].
- [56] D. Bailey, Java Structures: Data Structures in Java for the Principled Programmer, 2nd edition, (2007) pp. 528, McGraw-Hill Science/Engineering/Math.
- [57] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén 'Experimentation in Software Engineering: An Introduction, 2000.
- [58] N. Mustafa, Y. Labiche, D. Towey, Mitigating Threats to Validity in Empirical Software Engineering: A Traceability Case Study. 43rd Annual Computer Software and Applications Conference, USA, July, 2019, pp. 324-329.
- [59] Eclipse Specification. 2011, Available from: <http://www.eclipse.org/>
- [60] L. Frias, A. Queralt, A. Oliv, EU-Rent car rentals specification. Technical report, 2003.

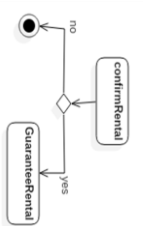
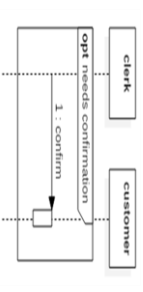
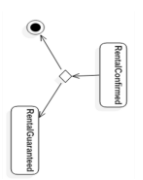
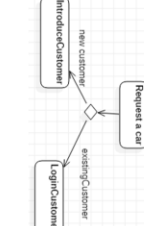
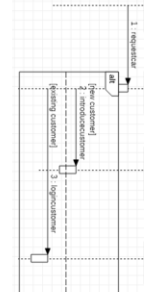

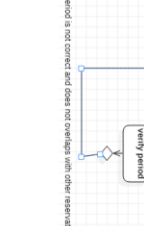
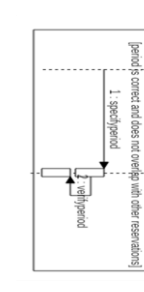
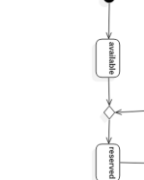
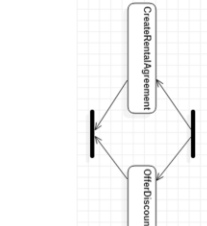
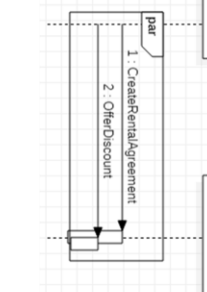
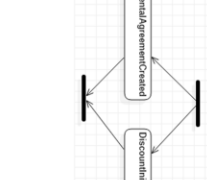
USE CASE DIAGRAM	ACTIVITY DIAGRAM	SEQUENCE DIAGRAM	STATE TRANSITION DIAGRAM	Code
<p>OPT</p> <p><Op><needs confirmation> <NumAction 6>< From Actor The Clerk ><To Actor Customer><Type of Result Simple: Boolean ><Action Description Asks the customer if he/she wants to guarantee the reservation><In-Parameter Reservation><Out-Parameter Is Validated >< IsConsidered 1>< IsIgnored 0>< IsNegative 0> <restart at Num "6" in SN></p>				<pre> If (rentalConfirmed) { guaranteeRental(); } </pre>
<p>ALT</p> <p><begin at Num 2 in SN> Begin <Event, begin at Num 2 in SN> <IF>< the customer does not exists> <NumAction 1>< From Actor The Clerk ><To Actor The customer>< Type of Result Customer (name, ID, birthdate, address, telephone)>< Action Description Introduce a new customer><In-Parameter Customer (name, ID, birthdate, address, telephone)><Out-Parameter Customer >< IsConsidered 1>< IsIgnored 0>< IsNegative 0> <Else ><restart at num 3 in SN> <begin at Num 3 in SN><Do> <NumAction 1>< From Actor the clerk ><To Actor the reservation><Type of Result Simple>< Action Description Specifies the period ><In-Parameter period ><Out-Parameter period >< IsConsidered 1>< IsIgnored 0>< IsNegative 0> <NumAction 2>< From Actor The clerk><To Actor The reservation><Type of Result correct yes/no>< Action Description Verify the period> <In-Parameter period> <Out-Parameter correct yes/no > < IsConsidered 1>< IsIgnored 0>< IsNegative 0> <While><period is correct and does not overlaps with other reservations > <restart at Num "6" in SN></p>				<pre> Do SpecifyPeriod(); VerifyPeriod(); While (period is correct and does not overlaps with other reservations) reservations) </pre>
<p>Do while</p> <p><Parallel> <NumAction 6>< From Actor The Clerk ><To Actor The agreement>< Type of Result Entity>< Action Description Create the reservation agreement ><In-Parameter rental agreement: ID customer, price, ID reservation><Out-Parameter Rental agreement >< IsConsidered 1>< IsIgnored 0>< IsNegative 0> <NumAction 7>< From Actor The Clerk ><To Actor The agreement>< Type of Result Entity>< Action Description Offer a discount to the customer ><In-Parameter Discount rental agreement : Discount, ID agreement, ID customer><Out-Parameter Discount rental agreement >< IsConsidered 1>< IsIgnored 0>< IsNegative 0></p>				<pre> public class rentalAgreementThread implements Runnable { Thread rentalAgreementThread; public rentalAgreementThread (int agreement, int IDloc, int IDMAT, date dat) {this.agreement= agreement; This .IDloc=IDloc; This .IDMAT; This .dat=dat;} rentalAgreementThread = new Thread (this , " OfferDiscountThread"); UnThread.start (); } //....second thread actions here public OfferPointsPayment(); } </pre>
<p>PARALLEL</p> <p><Parallel> <NumAction 6>< From Actor The Clerk ><To Actor The agreement>< Type of Result Entity>< Action Description Create the reservation agreement ><In-Parameter rental agreement: ID customer, price, ID reservation><Out-Parameter Rental agreement >< IsConsidered 1>< IsIgnored 0>< IsNegative 0> <NumAction 7>< From Actor The Clerk ><To Actor The agreement>< Type of Result Entity>< Action Description Offer a discount to the customer ><In-Parameter Discount rental agreement : Discount, ID agreement, ID customer><Out-Parameter Discount rental agreement >< IsConsidered 1>< IsIgnored 0>< IsNegative 0></p>				<pre> public class rentalAgreementThread implements Runnable { Thread rentalAgreementThread; public rentalAgreementThread (int agreement, int IDloc, int IDMAT, date dat) {this.agreement= agreement; This .IDloc=IDloc; This .IDMAT; This .dat=dat;} rentalAgreementThread = new Thread (this , " OfferDiscountThread"); UnThread.start (); } //....second thread actions here public OfferPointsPayment(); } </pre>

Table 9: Traceability between control structure fragments in the use case, Activity, sequence state transition diagrams and code

