Seventh Framework Programme

Call FP7-ICT-2013-10

Project Acronym: **S-CASE**

Grant Agreement N^{o}: **610717**

Project Type: **COLLABORATIVE PROJECT**

Project Full Title: **Scaffolding Scalable Software Services**

D2.4 Mining models for SE-related associations

Nature:	**R**
Dissemination Level:	**PU**
Version #:	**1.3**
Date:	**25 February 2015**
WP number and Title:	**WP2 Automated model-driven mapping and transformation**
Deliverable Leader:	**AUTH**
Author(s):	**Themistoklis Diamantopoulos (AUTH)**
Revision:	**Konstantinos Giannoutakis (CERTH), Ciro Formisano (ENG), Andreas Symeonidis (AUTH)**
Status:	**Submitted**

Document History

Version[1]	Issue Date	Status[2]	Content and changes
0.1	10 July 2015	Draft	TOC
0.2	17 July 2015	Draft	Added introduction, executive summary
0.3	20 July 2015	Draft	Added state-of-the-art analysis
0.4	21 July 2015	Draft	Added section 3
0.5	22 July 2015	Draft	Added section 4
0.6	23 July 2015	Draft	Added section 5
0.7	24 July 2015	Draft	Added conclusions
0.75	30 July 2015	Draft	Internal review
0.8	18 August 2015	Peer-Reviewed	Made corrections noted by reviews
0.9	20 August 2015	Peer-Reviewed	Added caption lists noted by reviews
1.0	21 August 2015	Final	Sent to Commission
1.1	5 February 2016	Revised	Added comparison section on UML diagrams (subsection 5.4)
1.2	18 February 2016	Revised	Added improved literature review (section 2) and added evaluation section on UML diagrams (subsection 5.5) and summary of changes (subsection 1.4)
1.3	22 February 2016	Revised	Internal review - Made final modifications
1.4	26 February 2016	Submitted	Submitted to PO

Peer Review History[3]

Version	Peer Review Date	Reviewed By
0.75	30 July 2015	Konstantinos Giannoutakis (CERTH)
0.75	30 July 2015	Ciro Formisano (ENG)
0.9	30 July 2015	Andreas Symeonidis (AUTH)
1.2	21 February 2016	Andreas Symeonidis (AUTH)

[1]Please use a new number for each new version of the deliverable. Use “0.#” for Draft and Peer-Reviewed. “x.#” for Submitted and Approved”, where x>=1.Add the date when this version was issued and list the items that have been added or changed.

[2]A deliverable can be in one of these stages: Draft, Peer-Reviewed, Submitted and Approved.

[3]Only for deliverables that have to be peer-reviewed

Table of Contents

List of Tables

List of Figures

Abbreviations and Acronyms

CIM	Computationally Independent Model
CRUD	Create, Read, Update, Delete
CSE	Code Search Engine
LCS	Longest Common Subsequence
MDE	Model-Driven Engineering
M2M	Model-to-Model
PIM	Platform Independent Model
PSM	Platform Specific Model
RE	Requirements Engineering
RAML	RESTful API Modeling Language
REST	REpresentational State Transfer
RSRE	Recommendation System in Requirements Engineering
RSSE	Recommendation System in Software Engineering
UML	Unified Modeling Language

Executive Summary

The objectives of WP2 include the design and development of a Model-Driven Engineering mechanism as well as the research and development of mining techniques for software artefacts. As part of task T2.4, this deliverable comprises novel methodologies for mining models for Software Engineering related associations.

In specific, the work on this task involves applying data mining techniques in functional requirements, source code of RESTful web services, and UML diagrams, in order to extract useful associations.

Concerning functional requirements, our work includes a novel recommendation system that can recommend functional requirements in the form of semi-structured natural language text. Using the requirements of past software projects, association rule mining techniques and heuristics are employed to determine whether the requirements of a project are complete and suggest new requirements.

Concerning the source code implementation of RESTful web services, this work focuses on the task of providing examples of algorithmic resources. In specific, we present a system that uses code search engines to retrieve algorithmic implementations and employs mining techniques to conform to the resource models of RESTful web services.

Finally, our methodology on UML models involves reusing UML use case and activity diagrams in order to improve the process of requirements elicitation. Upon parsing these types of diagrams into models, we employ matching techniques in order to find similar diagrams.

1 Introduction

1.1 WP2 Objectives

The overall objective of WP2 is to allow the successful storing and querying of meta-information for software artefacts as well as the transformation of this meta-information to RESTful web services upon developer request. The most important objectives of WP2 are summarized as follows:

1. Define the structure and constraints of the Platform Independent Model (PIM) given the output of multimodal user requirements.
2. Define the structure and constraints of the Platform Specific Model (PSM) in order to specify the abstract PIM design to a concrete set of technologies.
3. Design the interfaces to the S-CASE ontology to support automated storing and retrieval of software artefacts.
4. Develop an automated CIM to PIM Model-to-Model (M2M) transformation mechanism.
5. Develop an automated PIM to PSM M2M transformation mechanism.
6. Develop mechanisms for transforming 3rd party services into models and S-CASE services.
7. Design and develop mining mechanisms for discovering associations between models.

This deliverable concerns the design and development of methodologies of mining models from software artefacts. Thus, the work described in this deliverable focuses on objective 7 of the above list.

1.2 Scope of Task 2.4

The scope of task T2.4, which is described by this deliverable, involves applying data mining techniques in order to extract useful associations from three types of input. These types include:

- Functional requirements in the form of semi-structures natural language text
- Source code implementations of RESTful web services
- UML use case and activity diagrams

Our work on functional requirements concerns the problem of recommending new requirements given the requirements of past software projects. Association rule mining techniques and heuristics are used to determine whether the requirements of a project are complete and suggest new requirements to the user.

Concerning the source code of RESTful web services, we focus on providing examples of algorithmic resources. Note that tasks T2.1, T2.2, and T2.3 ensure that the source code provided to the developer as the output of the MDE engine is complete in terms of modelling resources. In specific, the source code extracted from the engine involves a connection to a database and a complete implementation for all CRUD resources. Therefore, the work in this deliverable focuses on providing examples of algorithmic resources. We present a system that uses code search engines to retrieve algorithmic implementations and employs mining techniques to conform to the resource models of RESTful web services.

Finally, the work on UML diagrams concerns the models that are given by the requirements engineer/developer as part of multimodal input. Thus, the process of requirements elicitation can benefit from reusing UML models of use case and activity diagrams. Upon parsing these types of diagrams into models, we employ matching techniques in order to find similar diagrams.

1.3 Structure of this Deliverable

This document is structured as follows. Section 2 describes the state-of-the-art on mining functional requirements, source code mining, and mining UML diagrams and models. Section 3 presents the details of a system for creating a recommendation system for functional requirements. Section 4 describes a methodology for finding useful source code fragments for the development of algorithmic resource implementations. Section 5 describes our methodology for reusing UML use case and activity diagrams. Finally, Section 6 summarizes our main contributions on this task.

1.4 Summary of Changes

Following the first review of this deliverable, further work has been performed in order to comply with the guidelines provided by the reviewing team, aiming to improve the overall quality of the work performed as part of task T2.4. Major changes include:

- Augmenting state-of-the-art analysis on mining UML models (subsection 2.4), focusing on representing UML models as ordered trees.
- Revision of the main contributions of this task with regard to UML models (subsection 2.5), in order to refer to the progress made with respect to the updated state-of-the-art section.
- Further assessment our UML mining methodology against the current state-of-the-art, in order to illustrate the effectiveness of our approach for recommending UML use case and activity diagrams (subsection 5.4).
- Evaluation of our UML mining methodology against the current state-of-the-art on a UML models dataset, comprising 65 Use Case diagrams and 72 Activity diagrams, in order to determine whether our approach can effectively detect semantically similar UML diagrams (subsection 5.5).

2 State-of-the-art Analysis

2.1 Overview

The area of Requirements Engineering covers the activities of discovering, defining, analyzing, and maintaining the requirements of software projects [1]. The process of requirements elicitation according to the needs of the stakeholders can be a particularly challenging task, especially today that software systems have grown to be more and more complex. Additionally, the need for proper identification of software requirements is crucial, since reengineering costs as a result of poorly specified requirements are considerably high [2].

In this context, several research efforts in the field of RE have focused on improving various aspects of the requirements elicitation process. Additionally, the rise of the open source community and the need for designing and developing large and complex software have attracted the attention of several researchers and practitioners. The potential of utilizing the vast available amounts of existing information of software projects in order to extract useful recommendations for software projects is explored in several fields in order to facilitate the reuse of software artefacts.

In the following subsections, we provide a state-of-the-art analysis for the main research axes explored in the task of this deliverable. Subsections 2.2, 2.3, and 2.4 provide a review of research efforts on mining functional requirements, source code, and UML models, respectively. Finally, subsection 2.5 summarizes our main contributions indicating the progress beyond the state-of-the-art.

2.2 Background on Mining Functional Requirements

Several researchers have proposed data mining techniques and recommendation systems technologies in order to facilitate the process of requirements elicitation. The area of Recommendation Systems in Requirements Engineering (RSREs) comprises systems that aim to provide valuable information that may refer to any process and/or artefact related to software requirements. RSREs span along various processes of RE, including project management (i.e. decision support) and release planning [3], [4], requirements elicitation and analysis [5]–[14], quality assurance and validation [15], [16], etc. In the following paragraphs, we review the current literature for requirements elicitation systems and particularly systems that are used to recommend requirements (not only stakeholders as in [17]).

Early research efforts in RSREs for requirements elicitation were mainly involved with domain analysis, using linguistics (vocabularies, lexicons) in order to determine the domains of projects and possibly identify missing entities and relations at the requirements level. One of the first systems to support this kind of analysis is the Domain Analysis and Reuse Environment (DARE), a Computer Aided Software Engineering (CASE) tool designed and developed by Frakes et al. [5]. DARE utilized multiple sources of information, including not only the requirements of a project, but also its architecture and source code. Upon extracting entities and relations from several projects, DARE used clustering techniques in order to identify common entities and recommend similar domain artefacts and architectural schemata for each project.

In the same context, Kumar et al. [6] utilize ontologies that store software requirements and possible project domains in order to improve the specifications of agile requirements. Subsequent work by Ghaisas and Ajmeri [7] involves using ontologies to develop a Knowledge-Assisted Ontology-Based Requirements Evolution (K-RE) method and toolset. The authors construct a domain knowledge repository in order to facilitate the creation of software requirements and especially resolve conflicts between change requests.

Another quite interesting line of work involves identifying the features of a system, using possibly its requirements, and recommending new ones. Chen et al. [8] used requirements from several software projects in order to construct requirements relationship graphs and subsequently extract domain information using clustering techniques. In specific, their system can identify features, such as e.g. reading from a file, and use them to create a feature model for projects of a specific domain. In the same context, Alves et al. [9] constructed a domain feature model, employing the vector space model and using latent semantic analysis to find similar requirements and cluster them into domains.

Using requirements to identify features is an idea explored also by Dimitru et al. [10]. The system described by the authors analyzes requirements of software projects using association rule mining and groups them using clustering techniques. The groups are then used to identify similar projects (using the k-Nearest Neighbors algorithm) and recommend new features.

Apart from the feature recommendation techniques discussed so far, RSREs can also be used for the non-functional characteristics of software projects. A notable research work by Romero-Mariona et al. [11] involves creating a model for several different approaches in security requirements engineering. Thus, the developer may define the importance of certain characteristics for each approach, and the system returns a ranked list of the most compliant approaches.

Finally, the relation between requirements and stakeholders has also been explored in the context of RSREs. Lim and Finkelstein [12] introduce StakeRare as an extension to their previous work in stakeholder identification [17]. StakeRare can be used to prioritize and reuse requirements in a project with multiple stakeholders. Upon providing a rating for each requirement, the system uses collaborative filtering to recommend new requirements and prioritize requirements according to the preferences of stakeholders and their influence in the project. A similar line of research followed by Castro-Herrera et al. [13] and Mobasher and Cleland-Huang [14] involves employing collaborative filtering to recommend requirements to the most relevant stakeholders.

2.3 Background on Source Code Mining

Ever since its introduction, the REST architectural style has been constantly preferred by several developers for its simplicity and scalability, thus it has now grown to be the state-of-the-practice for creating web services. The main building blocks of a RESTful web service are *resources*. Each resource provides an object of the system that can be addressed using one of the four CRUD operations: Create, Read, Update, and Delete. Thus, given for example a library management system, we could have a resource “book”. The CRUD operations would then be used to add, delete or update books to the library.

The API of a RESTful service is usually on top of a database system. In the library management system example, the underlying database would handle most of these simple operations. However, sometimes the operations required by a web service may be much more complex. For example, consider the following queries to the service:

1) Return the text of the book with id 4.
2) Return the text of the book with id 4, wrapped left.
3) Return the number of books that contain a specific word.

The first query is quite simple, requiring a GET request to return a book and its text, as long as it exists. Queries 2 and 3, however, are much more complex. In specific, query 2 requires not only performing some request on the data, i.e. retrieving the text of Hamlet, but also performing some other operation on the returned text. Correspondingly, query 3 requires searching, i.e. an operation on the data. Although query 3 requires searching the database while query 2 requires only retrieving a specific text, both queries require some kind of algorithm. Since in RESTful web services, any API operations including algorithms have to be modeled as resources, algorithms are resources too. In specific, algorithms in RESTful web services are called algorithmic resources [18].

In the context of S-CASE, and particularly WP2, the MDE module shall handle the automatic source code generation of CRUD resources. Thus, the problem analyzed in this task is the recommendation of source code that can be reused in the construction of algorithmic resources. Similar systems that search for reusable code can be traced in the area of Recommendation Systems in Software Engineering (RSSEs). In the following paragraphs, we analyze certain code-reuse RSSEs that find example source code for the developers.

In the context of code reuse, XSnippet [20] is one of the first RSSEs to provide examples to the developer, however its recommendations focus on the SDK of the Eclipse IDE. PARSEWeb [21] and MAPO [22] provide recommendations based on the results of CSEs in order to aid the developers use APIs. In specific, these two systems recommend API usages given queries from a source to a destination object.

Code Conjurer [23] and CodeGenie [24] follow a test-driven approach to provide useful components. The systems search for useful components in CSEs and use tests to confirm that the functionality of the components complies with the requirements of the developer. Although the problem of component reuse is quite similar to the scope of this work, the components recommended by these services are mostly structured implementations of interfaces, thus they do not comply with the characteristics of algorithms or the properties of RESTful web services.

2.4 Background on Mining UML Models

During the latest decades, several research efforts have been directed towards the idea of applying data mining techniques on UML models and diagrams. Most early efforts in the field employed Information Retrieval techniques. Despite not using UML models, the work of Alspaugh et al. [25] is largely relevant since it is among the first works in the direction of scenario reuse. Scenarios are defined as a set of *events* triggered by *actions* of *actors*, while *authors* and *goals* are also defined for them. Blok and Cybulski [26] employ the vector space model in order to represent the flow of events for every use case. The similarity between use cases is then computed using the cosine similarity of the vectors of events.

Another interesting line of work involves extracting graphs from UML diagrams and employing graph matching techniques to find similar graphs. Woo and Robinson [27], [28] encoded UML elements of use case, sequence, and class diagrams as vertices and their associations as edges. After that, the authors applied a graph matching algorithm to the generated graphs. In a similar context, Park and Bae [29] convert sequence diagrams to Message Object Order Graphs (MOOGs) where the messages are represented as nodes and the edges denote the flow between the objects, either message flow or temporal flow. Salami and Ahmed [30] extract the adjacency matrix of the directed graph of class diagrams, where classes and represented as nodes and relationships as edges. The authors use an inexact graph matching technique to find similar diagrams in order to reduce the complexity of the problem. As part of the EU-funded project ReDSeeDS, graph matching and information retrieval techniques are also employed to find similar use case, sequence, and activity diagrams [34].

Although graph based methods can be quite effective for certain types of structured models, their application to differencing UML diagrams lacks semantically, as noted also by Kelter et al. [31], since they only employ arbitrary graph similarity metrics without actually focusing on the construction of a descriptive model. As an alternative, several researchers have focused on representing UML diagrams (and other types of XML structures in general) as ordered trees [31]–[33]. Ordered trees can be quite effective for capturing the structure of UML diagrams, while they also result in more efficient implementations than those of graph models. The main focus of this line of work is the design of a data model that covers the elements and the structure of UML diagrams. Upon properly defining and populating such a data model, the problem is reduced to computing the similarity of ordered trees.

Finally, the rise of semantic web technologies during the latest decade has inspired several researchers to use ontologies in order to detect semantically similar UML diagrams. Gomes et al. [35] were among the first to extract terms from class diagrams and semantically match them using WordNet [39]. Additionally, the authors demonstrated how the use of domain specific ontologies can result in improved matching. Robles et al. [36] use an ontology for measuring the similarity between classes and relationships in class diagrams and then semantically relate one class diagram with another using an ontology for measuring the semantic distance between their class names. Bonilla-Morales et al. [37] create a database of ontology instances for use case diagrams. Each instance contains actors and use cases, which can be searched through a query interface.

2.5 Task Contributions and Progress beyond the State-of-the-art

In this section we summarize our main contributions with regard to the current state-of-the-art. In the following paragraphs, we indicate the shortcomings of current systems, and illustrate how our methodology is better oriented towards the three tasks at hand: mining and recommending functional requirements, recommending source code components for the implementation of algorithmic resources, and finding similar UML diagrams to support UML model reuse.

Concerning functional requirements mining, we may notice that there are several domain-centered approaches [5]–[9]. This is not entirely unexpected since using domain knowledge can lead to better understanding of the software project at hand. However, as noted also in

[10], these domain analysis techniques are usually not applicable due to the lack of available annotated requirements in a particular domain. Feature-based recommendation techniques, either functional [10] or non-functional [11] do not face these issues, however their scope is usually too high-level, especially for fine-grained requirements artefacts, such as functional requirements. Finally, the scope of the stakeholder-aware techniques [12]–[14] is mostly oriented towards the process of the requirements elicitation. In this work, we focus on functional requirements and create a system that can provide low-level recommendations, maintaining a semantic, yet domain-agnostic outlook.

Finding source code examples is a quite popular research area [20]–[24]. Code-reuse RSSEs have been widely used in the latest decades. However, most systems do not comply with the RESTful architectural styles, and thus do not fit the problem of finding algorithmic resources. Due to its clear resource model, the RESTful paradigm has been the subject of several services that allow automating parts of the development process, such as the MDE module of tasks 2.1, 2.2, and 2.3. The MDE module aims on creating a skeleton and a database schema for resources. In this deliverable we facilitate reuse in the concept of algorithmic resources. To the best of our knowledge, there is no RSSE oriented towards providing algorithmic resource implementations for RESTful web services. Therefore, in Section 4, we provide a system that conforms to the special characteristics of REST and provides algorithms in the form of class and method components.

Concerning UML diagrams, although current literature is oriented towards several directions, there is a common methodology that involves extracting models from UML diagrams and mining these models using a variety of methods. Graph based methods [27]–[34] are effective for the structured models of class diagrams, however they do not conform to the simple nature of use case and activity diagrams, since the latter have much simpler structures. Ordered tree structures [31]–[33] provide simpler models, while at the same time preserving the structure of UML diagrams (and other types of XML structures in general) as ordered trees. However, given that the key aspect of all structure-based approaches is to define a complete data model for all types of diagrams, they usually focus on the static aspects of diagrams, omitting their dynamic characteristics. As a result, they are not effective for representing data flows or flows of actions. Furthermore, the string similarity methodologies of most aforementioned approaches are limited to string difference methods, thus they are not applicable to complex scenarios with multiple sources of diagrams. Ontology based, i.e. semantics enabled, methods [35]–[37] are proven to be quite useful when domain knowledge is taken into account, however most of the time domain specific information is limited. Finally, although Information Retrieval techniques [25], [26] are not limited to domain information, their lexical approach is not effective for diagrams that incorporate structure or flow information, such as activity diagrams.

In this work, we propose two methodologies, one for reusing use case diagrams and one for reusing activity diagrams. Both approaches are domain agnostic in order to ensure their broad applicability. The use case diagram matching algorithm uses distance metrics in order to avoid the strict structural limitations of graph based algorithms, while at the same time handling use cases as discrete nodes in contrast to Information Retrieval techniques. The activity diagram matching algorithm represents diagrams are sequences of action flows, thus ensuring that the dynamic features of the diagrams are kept intact, without however over-engineering their structure using graph based methods. This algorithm actually resembles ordered tree approaches [31]–[33], as these approaches offer a middle ground between unstructured and heavily structured methods, and hence are preferred by current UML tools. As a result, we are also inclined to assess our algorithms against a state-of-the-art ordered tree approach [31]. Finally, both diagram matching algorithms also use a semantic, yet domain-agnostic, scheme for strings in order to provide more effective recommendations, especially in cases where the diagrams originate from different sources and diverse types of projects.

3 Mining Functional Requirements

3.1 Overview

As noted in the previous Section, finding domain-specific requirements is a challenging task, except perhaps for large organizations that are active in a specific domain and thus may have several relevant projects in their private repositories. Thus, in this Section, we present a methodology that can be applied to requirements of projects which are relevant to diverse domains. In the following subsection we present an example for the annotation scheme of task 3.1 and illustrate how the annotated requirements can be mined in order to extract useful association rules and finally use these rules to recommend functional requirements.

3.2 A Recommendation System for Functional Requirements

3.2.1 Annotating Requirements

For the needs of annotating functional requirements, we used the annotation scheme defined in deliverable D3.1.2 of WP3. The annotation scheme developed as part of this task originated from an ontology that was created for storing functional requirements and effectively representing the static view of a software project. The design of the ontology involves the concept of an actor performing some action(s) on some object(s), including also various properties that may act as modifiers or as elements of the actor or the object.

The annotation scheme includes four types of entities: `Actor`, `Action`, `Object`, and `Property`. Three relations are specified among entities, including: `IsActorOf` declared from `Actor` to `Action`, `ActsOn` defined from `Action` to `Object` or from `Action` to `Property`, and `HasProperty` defined from `Actor` to `Property` or from `Object` to `Property` or from `Property` to `Property`.

An example of an annotated functional requirement is shown in Figure 3.1.

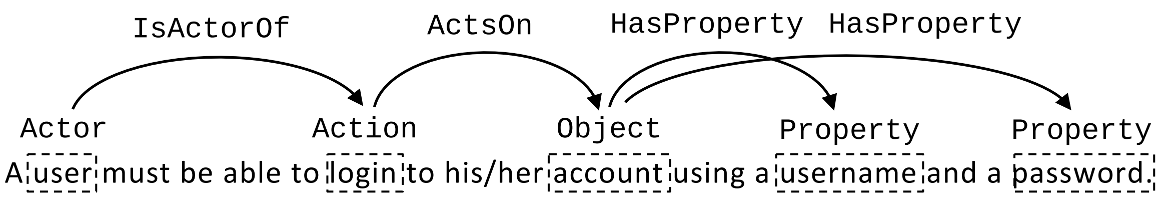
IsActorOf
ActsOn
HasProperty
HasProperty
Actor
Action
Object
Property
Property
A user must be able to login to his/her account using a username and a password.

Figure 3.1 Example depicting an annotated functional requirement

As shown in this Figure, the structure of the sentence follows the Subject-Verb-Object (SVO) motif, thus annotating it is intuitive. In addition, using the Natural Language Processing (NLP) parser constructed in task 3.1 (deliverables D3.1.1 and D3.1.2), the procedure of annotating one's project requires little effort. After annotating the functional requirements of a software projects, the annotations can be used to construct certain rules that correspond to relations. For the annotated requirement of Figure 3.1, the four relations (indicated with arrows) correspond to the four items shown in Figure 3.2.

```
user_IsActorOf_create
create_ActsOn_account
account_HasProperty_username
account_HasProperty_password
```

Figure 3.2 Annotated items of the requirement of Figure 3.1 that correspond to relations

The preprocessing of these items and their use for recommending requirements are analyzed in the following subsections.

3.2.2 Semantically Relating Terms of Requirements

As already noted, our system should be able to recommend requirements given projects of different domains. Since we do not use any domain-specific information, the semantics of our system have to be generic enough to support relating domain-agnostic terms between different projects. Having extracted the items for each requirement, we can identify a set of terms for each requirement and subsequently for each project. For example, for the items shown in Figure 3.2, we may extract the terms `user`, `create`, `account`, `username`, and `password`. Thus, assume we have another project that also involves an account for each user, yet the term used for the user account is `profile`. In this case, the two terms have to be marked as semantically similar.

Marking two terms as similar requires a database of words and their semantics, as well as a similarity measure. We use WordNet [39] as our database, and specifically interface with it using the MIT Java Wordnet Interface (JWI) [40]. JWI supports similarity metrics between terms using the Java Wordnet::Similarity (JWS) library[4]. Although the similarity between two terms can be determined using a variety of methods [41], several of them do not employ semantics while others that do may not be well correlated to human judgments [42]. As a result, we decided to use the information-content measure introduced by Lin [42], since it is universal and fits well the generic human judgment for similar terms.

According to Lin [42], the similarity between two terms (WordNet classes) C_1 and C_2 is defined as:

$$sim(C_1,C_2) = \frac{2 \cdot \log P(C_0)}{\log P(C_1) + \log P(C_2)} \tag{3.1}$$

where C_0 is the most specific class that contains both C_1 and C_2. For example given account and profile, the most specific class that describes both terms is record. This example is also visualized in Figure 3.3.

[4] JWS, which is available at http://users.sussex.ac.uk/~drh21/, is based on the Perl module for WordNet similarity metrics [41].

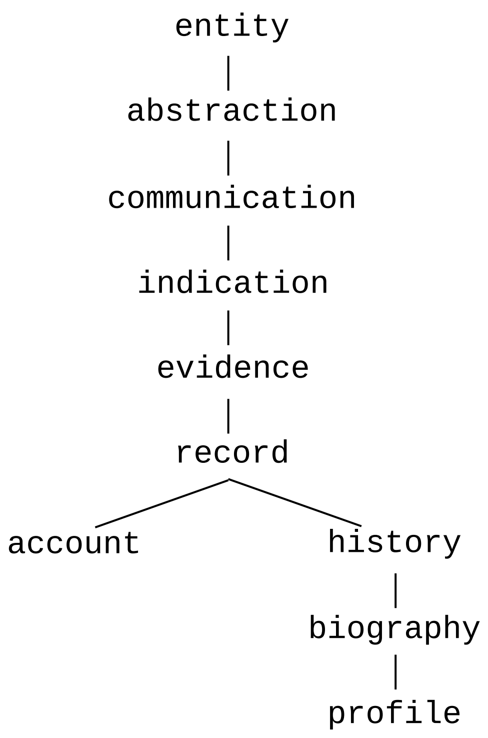

Figure 3.3 Example fragment of WordNet depicting paths to `account` and `profile`. `record` is the most specific class that describes both elements.

Upon having found C_0, the (maximum) value of equation (3.1) can be computed using the information content of each WordNet class. For each class, its information content is determined as the logarithm of the probability that a word in the corpus belongs to this class[5]. For example, for `record`, `account`, and `profile` these values are 7.874, 7.874, and 11.766 respectively, thus the similarity between `account` and `profile` is $2 \cdot 7.874 / (7.874 + 11.766) = 0.802$.

Finally, in the context of our RSRE, we first extract all terms and then we employ the aforementioned semantics before adding an item to any itemset (i.e. a set of items). In specific, we assume two terms have similar meaning if their similarity is more than a threshold t. This ensures that any semantic relations between the terms are taken into account, both when training the system and later on when using it for recommendations.

3.2.3 Extracting Association Rules from Requirements

Upon having extracted the annotations of several software projects, we now have a dataset consisting of one set of items, i.e. one itemset, per software project. In this subsection, we illustrate how we can extract useful association rules from these items using association rule learning [43]. Let $P = \{p_1, p_2, \ldots, p_m\}$ be the set of m software projects and $I = \{i_1, i_2, \ldots, i_n\}$ be

[5] For the information content of each class, we use the precomputed information content files of the Perl module for WordNet similarity metrics [41], available at http://www.d.umn.edu/~tpederse/.

the set of all n items. The support of an itemset X is defined as the number of projects in which all items of the itemset appear in:

$$\sigma(X) = \left|\left\{p_i \mid X \subset p_i, p_i \in P\right\}\right| \tag{3.2}$$

An association rule is expressed in the form $X \to Y$, where X and Y are disjoint itemsets. For example, an association rule that can be extracted from the items shown in Figure 3.2 is {`account_HasProperty_username`}→ {`account_HasProperty_password`}.

The strength of each association rule is determined by its *support* and its *confidence*. Given a rule $X \to Y$, its support indicates the number of projects for which the rule is applicable, and it is given as:

$$\sigma(X \to Y) = \frac{\sigma\left(X \cup Y\right)}{|D|} \tag{3.3}$$

The confidence of the rule indicates how frequently items in Y appear in X, and it is given as:

$$c(X \to Y) = \frac{\sigma\left(X \cup Y\right)}{\sigma\left(X\right)} \tag{3.4}$$

Thus, the aim of association rule learning is to extract the association rules of the dataset that have support and confidence values above certain thresholds. We use the Apriori algorithm for this task [43], since it is quite effective in finding rules requiring also relatively few computations.

3.2.4 Recommending Functional Requirements

Upon extracting the association rules, in this subsection we indicate how they can be used to recommend new requirements for a software project. At first, given the newly added software project, we create a new itemset p following the procedure defined in the previous subsection. After that, given this itemset and the set of rules, we can extract the set of activated rules R. A rule $X \to Y$ is activated in project itemset p if all items in X are contained in p (i.e. $X \subset p$).

Given the set of activated rules R, the next step is to transform them to rules containing single items as antecedents and consequents. This rule flattening procedure creates a rule for each combination of the antecedents and the consequents of the rule. For example, given the rule $X \to Y$ where the two itemsets X and Y contain the items $\left\{i_1, i_2, i_3\right\}$ and $\left\{i_4, i_5\right\}$ respectively, the new flattened rules are $\left\{i_1, i_4\right\}$, $\left\{i_1, i_5\right\}$, $\left\{i_2, i_4\right\}$, $\left\{i_2, i_5\right\}$, $\left\{i_3, i_4\right\}$, and $\left\{i_3, i_5\right\}$. We also propagate the support and confidence values of the original rules to these new flattened rules, so that they can be later used as criteria for the importance for each rule.

Our system receives as inputs the itemset of a project p and the flattened activated rules and outputs new requirements, using the heuristics shown in Table 3.1.

Table 3.1 Heuristics for the activated rule items of a software project

Antecedent	Consequent	Conditions	Result
$[Actor1, Action1]$	$[Actor2, Action2]$	$Actor2 \in p$	$[Actor2, Action2, Object]$, $\forall Object \in p : [Actor1, Action1, Object]$
$[Action1, Object1]$	$[Actor2, Action2]$	$Actor2 \in p$	$[Actor2, Action2, Object1]$
$[Actor1, Action1]$	$[Action2, Object2]$	$Object2 \in p$	$[Actor1, Action2, Object2]$
$[Action1, Object1]$	$[Action2, Object2]$	$Object2 \in p$	$[Actor, Action2, Object2]$, $\forall Actor \in p : [Actor, Action1, Object1]$
* (except for the above)	$[Action2, Object2]$	$Object2 \in p$	$[Actor, Action2, Object2]$, $\forall Actor, Action \in p : [Actor, Action, Object2]$
*	$[Any2, \mathrm{Pr}operty2]$	$Any2 \in p$	$[Any2, \mathrm{Pr}operty2]$

As shown in this Table, there are three types of heuristics corresponding to three different consequents. For the consequent $[Actor2, Action2]$, corresponding to an `Actor2_IsActorOf_Action2` item, we create a new requirement which includes the actor and the action of the consequent as well as an $Object$, which is determined by the antecedent. For example, given an antecedent $[create, bookmark]$ and a consequent $[user, edit]$, the system recommends a new requirement $[user, edit, bookmark]$.

Concerning the consequent $[Action2, Object2]$, which corresponds to an `Action2_ActsOn_Object2` item, we create a new requirement which includes the action and the object of the consequent while the actor is determined by the antecedent. For example, given an antecedent $[user, profile]$ and a consequent $[create, profile]$, the system recommends a new requirement $[user, create, profile]$. Finally, any consequent corresponding to `HasProperty` (with any antecedent) leads to a new requirements of the form $[Any, \mathrm{Pr}operty]$. An example of such a requirement is $[user, profile]$.

Finally, note that although natural language generation deviates from the scope of this work, the semi-structured form of functional requirements allows providing them to the user in a comprehensible pseudo-natural language format. Hence, the recommended requirements of our system are given in the form of sentences. This is accomplished using the following template sentences:

1) The $Actor$ must be able to $Action$ $Object$.
2) The Any must have $\mathrm{Pr}operty$.

Although the usage of these templates does not ensure that the syntax of the new requirement is totally correct, they are useful for presenting the requirements to the user in a comprehensible format, so that he/she can later rephrase them and add them to the requirements of his/her project.

3.3 Evaluation

3.3.1 Dataset

Since an important part of our hypothesis involves the notion of creating a generic RSRE, we used a very diverse dataset to evaluate our methodology. Our dataset consists of 30 projects, including student projects[6], pilot requirements (Giftcase), as well as the RESTAPPS [44] (Restmarks, Restmaps). In total, there are 30 projects with 514 requirements (i.e. approximately 17 requirements per project). Upon annotating, we have a dataset of 7234 entities and 6626 relations among them. After the preprocessing step of subsection (where the threshold for two similar terms is set to 0.5), we end up with 1512 items for all 30 projects, 1162 of which are distinct.

As one may notice, generating the set of all possible association rules is computationally prohibitive. Using, however, the Apriori algorithm we are able to find a set of the most useful rules (determined by support and confidence) in a few seconds. The minimum support of the rules was set to 0.1, indicating that any rule contained in at least 10% of the projects (i.e. 3 projects) is interesting. The minimum confidence was set to 0.5, indicating that the rule must be confirmed at least half of the time that its antecedents are found. The execution resulted in 1372 association rules. A fragment of them is shown in Table 3.2.

Table 3.2 Sample association rules extracted by the dataset

Association Rule	Support	Confidence
$[provide, product] \rightarrow [system, provide]$	0.167	1.0
$[system, validate] \rightarrow [user, login]$	0.1	1.0
$[user, buy] \rightarrow [system, provide]$	0.1	1.0
$[administrator, add] \rightarrow [administrator, delete]$	0.167	0.833
$[user, logout] \rightarrow [user, login]$	0.167	0.833
$[user, add] \rightarrow [user, delete]$	0.133	0.8
$[user, access] \rightarrow [user, view]$	0.1	0.75
$[edit, product] \rightarrow [add, product]$	0.1	0.75
$[administrator, delete] \rightarrow [administrator, add]$	0.167	0.714
$[user, contact] \rightarrow [user, search]$	0.133	0.5

[6] Mainly from a software development course organized jointly by several European universities, available at http://www.fer.unizg.hr/rasip/dsd

As shown in Table 3.2, several of these rules are quite rational. For example, the first rule indicates that if a product were to be provided, then it would be provided by the system. An example of two actions of which one implies the other is the fifth rule; if a user may log out of a system, he/she would probably also have to log in first. Finally, note that several rules may be bidirectional. For instance, if the administrator is able to add an object, then he/she could probably delete it (fourth rule), and vice versa (ninth rule). In the following subsection we illustrate how the association rules are used to recommend new requirements for a software project.

3.3.2 An Example of Recommending Requirements

In this subsection, we provide an example of recommending new requirements for a software project. We use project Restmarks as our example. Restmarks is a service that allows users to store their bookmarks online, and effectively share them with the community and search for bookmarks using tags. Thus, Restmarks can be seen as a social service for bookmarks. The functional requirements of Restmarks are shown in Figure 3.4.

A user must be able to create a user account by providing a username and a password.

A user must be able to login to his/her account by providing his username and password.

A user that is logged in to his/her account must be able to update his/her password.

A logged in user must be able to add a new bookmark to his/her account.

A logged in user must be able to retrieve any bookmark from his/her account.

A logged in user must be able to delete any bookmark from his/her account.

A logged in user must be able to update any bookmark from his/her account.

A logged in user must be able to mark his/her bookmarks as public or private.

A logged in user must be able to add tags to his/her bookmarks.

Any user must be able to retrieve the public bookmarks of any community user.

Any user must be able to search by tag the public bookmarks of a specific RESTMARKS's user.

Any user must be able to search by tag the public bookmarks of all RESTMARKS users.

A logged in user must be able to search by tag his/her private bookmarks as well.

Figure 3.4 Functional requirements of project Restmarks

At first, our methodology for creating association rules is applied to the 29 projects of our dataset (excluding Restmarks). After that, the rules that are activated by the annotated requirements of Restmarks are isolated and presented to the user along with the corresponding support and confidence values. Using the heuristics of Table 3.1, the recommended requirements for Restmarks are shown in Figure 3.5.

+ The user must be able to edit bookmark. ($\sigma = 0.138, c = 1.0$)

+ The user must be able to view bookmark. ($\sigma = 0.103, c = 1.0$)

+ The user must be able to view account. ($\sigma = 0.103, c = 1.0$)

+ The user must be able to edit tag. ($\sigma = 0.103, c = 1.0$)

+ The user must be able to edit account. ($\sigma = 0.103, c = 1.0$)

+ The user must be able to logout account. ($\sigma = 0.172, c = 0.62$)

– The user must be able to contact account. ($\sigma = 0.172, c = 0.62$)

– The user must be able to contact bookmark. ($\sigma = 0.138, c = 0.5$)

– The user must be able to stop account. ($\sigma = 0.138, c = 0.5$)

+/–: Correctly/Incorrectly Recommended Requirement
σ: Support, c: Confidence

Figure 3.5 Recommended requirements of our system for project Restmarks

As shown in this Figure, several of these requirements are actually quite rational. For example, the ability of the user to edit his/her tags (e.g. rename a tag) or log out of his/her account certainly seem to have been omitted by the engineers/stakeholders that originally wrote the functional requirements of Restmarks.

Finally, an interesting correlation can be observed between the quality of recommended requirements and the support and confidence values of the rules that they were derived from. Figure 3.6 visualizes the number of recommended requirements for each combination of support and confidence value, as well as the percentage of correctly recommended requirements for these combinations.

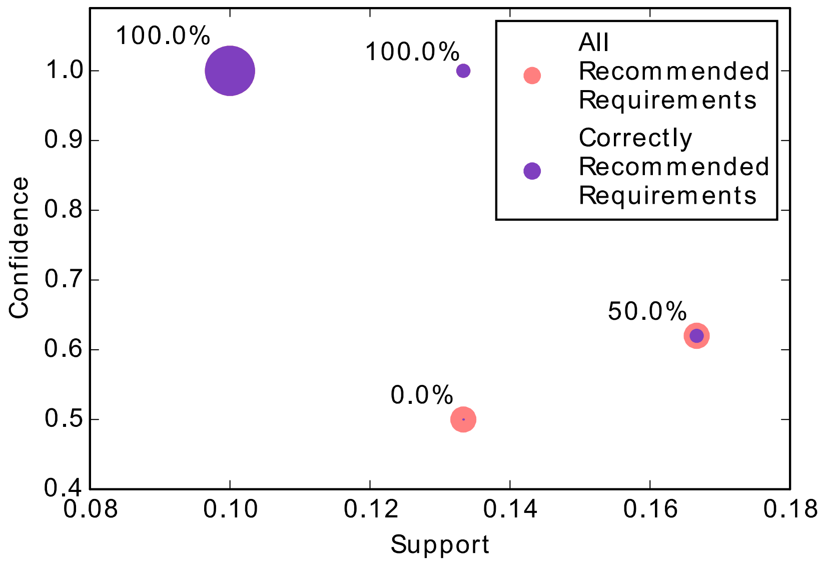

Figure 3.6 Visualization of the recommended requirements for project Restmarks including the percentage of the recommended and the correctly recommended requirements given their support and their confidence

As we can see in this Figure, most recommended requirements have high confidence values. Furthermore, requirements with high confidence and high support seem to be correctly recommended for Restmarks.

3.3.3 Experimental Results

In this subsection, we use a cross-validation scheme in order to further evaluate our method and explore the influence of support and confidence on the quality of the recommended requirements. At first, the dataset of subsection 3.3.1 is randomly split into 6 folds, such that each of them has 5 projects. For each of the 6 folds, we remove the 5 projects of the fold from the dataset, we extract the association rules from the remaining 25 projects, and finally our system recommends new requirements for the 5 removed projects.

Upon having the recommendations for each project, we examined the requirements and determined whether each recommended requirement is sensible. Note that the main scope of our evaluation is to determine whether the recommendations add some sensible functionality to the system under development. Hence, the developer could regard a recommendation as useful, yet decide not to add it to the final set of requirements. The decision to add a new requirement to a project may depend on several factors, including stakeholder preferences, required manpower to meet the requirement, etc.

The accumulated results of our analysis for all folds are shown in Table 3.3.

Table 3.3 Evaluation results for the recommended requirements

Support	Confidence	# Correctly Recommended Requirements	# Recommended Requirements	% Correctly Recommended Requirements
0.2	1.0	1	2	50.0%
0.133	1.0	23	37	62.16%
0.1	1.0	43	76	56.58%
0.133	0.8	2	4	50.0%
0.2	0.75	0	1	0.0%
0.1	0.75	64	92	69.57%
0.167	0.71	0	1	0.0%
0.133	0.67	13	14	92.86%
0.167	0.62	1	1	100.0%
0.1	0.6	9	17	52.94%

0.133	0.57	4	7	57.14%
0.167	0.56	4	5	80.0%
0.1	0.5	13	40	32.5%
	Total	**177**	**297**	**59.6%**

As shown in this Table, our system recommended 297 requirements in total, out of which 177 were recommended correctly. The results are actually quite satisfactory since almost 60% of the recommendations can lead to useful requirements that may be otherwise omitted. This, in the case of a single project, the requirements engineer would be presented with a set of 10 requirements on average, out of which he/she would select 6 to add to the project.

Note, however, that the number of recommended requirements per project depends on the number of the already existing requirements of the project (and possibly also their size, in the sense of the number of entities). This is quite expected, given the heuristics of Table 3.1 are largely based on the entities and relations of the project at hand. That is of course a desired feature, since it allows making project-centric recommendations, such as the ones of Figure 3.5.

The effect of the support and the confidence of the rules on the quality of the final recommendations is further explored by visualizing the recommended requirements in Figure 3.7.

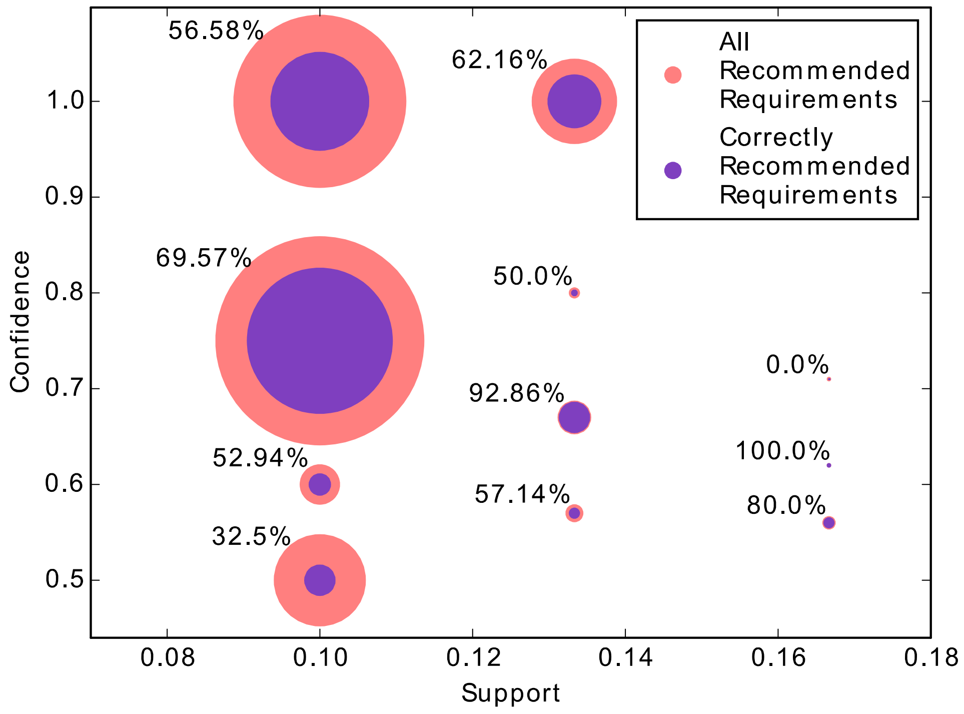

Figure 3.7 Visualization of the recommended requirements including the percentage of the correctly recommended requirements given support and confidence

The size of each circle indicates the number of recommended (in red color) and correctly recommended requirements (in blue color) for each combination of values of support and confidence. At first, we may note that most recommended requirements are extracted from rules with low support. This is actually expected since our dataset is largely domain-agnostic. However, note that low support rules do not necessarily result in low quality recommendations, as long as confidence values are large enough.

The confidence of the rules is highly correlated with the quality of the recommendations. Indicatively, 2 out of 3 recommendations extracted from rules with confidence equal to 0.5 may not be useful. However, setting the value of confidence to 0.75 ensures that more than 2 out of 3 recommended requirements can be added to the project. Rules with high support and high confidence values also lead to useful recommendations, however these rules may be limited.

4 Source Code Mining

4.1 Overview

In this Section, we provide an overview of an algorithm recommender that can be used for RESTful web services. Our system consists of three components, a downloader, a parser, and a matcher. The downloader handles the retrieval of source code from online repositories, and the parser extracts useful information from the downloaded files. After that, the matcher can be used to rank the files according to their compliance to the RESTful resource. These three components of our system are described in detail in subsection 4.2, while subsection 4.3 illustrates the applicability of our system in a case study.

4.2 Recommending algorithms for RESTful resources

4.2.1 Downloader

The downloader component uses AGORA, a CSE that allows syntax-aware queries, to retrieve useful results for the algorithm that is given as a query[7]. AGORA offers a powerful API, enabling developers to use complex queries to return relevant results. In our work AGORA is employed as a useful input resource for source code implementations.

The query for an algorithm requires two inputs: the name of the resource and the method name of the algorithm. For instance, given a resource “document”, where its model may have the properties “id”, “title” and “text”, the developer may require an algorithm “wordWrap” that would wrap the text of the document. The structure of this query is shown in Figure 4.1.

```
{
   "query": {
      "bool": {
         "must": [{
            "bool": {
               "should": [{
                  "match": {
                     "code.class.name": "wordWrap"
                  }
               },
               {
                  "nested": {
                     "path": "code.class.methods",
                     "query": {
                        "match": {
```

Figure 4.1 Example query for resource named “user” and method named “sort”

[7] The reader is referred to deliverable 5.1 of S-CASE for more information about AGORA.

```
                           "code.class.methods.name": "wordWrap"
                        }
                     }
                  }
               }]
            }
         },
         {
            "bool": {
               "should": [{
                  "nested": {
                     "path": "code.class.variables",
                     "query": {
                        "match": {
                           "code.class.variables.name": "document"
                        }
                     }
                  }
               },
               {
                  "nested": {
                     "path": "code.class.methods",
                     "query": {
                        "match": {
                           "code.class.methods.name": "document"
                        }
                     }
                  }
               },
               {
                  "nested": {
                     "path": "code.class.methods.parameters",
                     "query": {
                        "match": {
                           "code.class.methods.parameters.name": "document"
                        }
                     }
                  }
               }]
            }
         }]
      }
   }
}
```

Figure 4.1 (continued)

The query is a boolean query consisting of two subqueries. Since it is a must type of query, both subqueries have to be matched. The first subquery refers to the algorithmic resource that is searched. As shown in Figure 4.1, the algorithmic resource part is considered match if either the class name or a method name (or both) of the returned file is equal to the resource. For the RESTful resource part of the query, the name of the resource has to be matched in one or more of the following: the name of a variable of the file, the name of a method of the file, or the name of a parameter of a method.

Upon executing the query on AGORA, the results of the query are downloaded. These results comply with the resource name and algorithm name that were given by the user, however they may not comply with the properties of the resource, and the methods may not provide useful algorithms. The next subsections indicate how these results are parsed and matched to extract the useful method implementations.

4.2.2 Parser

The parser component is used for two different purposes. At first, it is used to form the query of the user and extract useful resource properties for matching. This is accomplished by parsing a model file of the RESTful web service. An example RESTful resource model file is shown in Figure 4.2.

```
...
XmlRootElement
@Entity
@Table(name = "document")
public class DocumentModel {

   @Id
   @GeneratedValue
   @Column(name = "documentId")
   private int documentId;

   @Column(name = "documentTitle")
   private String documentTitle;

   @Column(name = "documentText")
   private String documentText;
...

   public void setDocumentId(int documentId) {
      this.documentId = documentId;
   }

   public int getDocumentId() {
      return this.documentId;
   }
...
}
```

Figure 4.2 Example model file for resource “document”

As shown in this Figure, the model file includes the properties of the resource as columns (i.e. using the “Column” annotation), as well as setters and getters for these properties. In our case, the useful information to be parsed is the names and types of the properties. Thus, our specialized parser receives a model file such as the one of Figure 4.2 and extracts the name of the resource, the types of the properties, and the names of the properties in JSON format, shown in Figure 4.3.

```
{
   "name": "document",
   "variables": [{
      "name": "documentId",
      "type": "int"
   },
   {
      "name": "documentTitle",
      "type": "String"
   },
   {
      "name": "documentText",
      "type": "String"
   }],
}
```

Figure 4.3 Example model properties for the resource model of Figure 4.2

In the case of result files, we focus on method implementations since they are better suited for simple algorithms. In specific, we assume each method of the result files represents an algorithm that may suit the user. Hence, our parser extracts the methods of each class file. The excerpt for each method is given in JSON format in Figure 4.4.

```
{
   "body": "...",
   "bodyvariables": [{
      "name": "result",
      "type": "String[]"
   },
   {
      "name": "line",
      "type": "int"
   },
   ...
   ],
   "name": "wordWrap",
   "returntype": "String[]",
   "modifiers": ["public"],
   "throws": [],
```

Figure 4.4 Example extracted information for method “public String[] wordWrap(String text, double width)”

```
   "parameters": [{
      "name": "text",
      "type": "String"
   },
   {
      "name": "width",
      "type": "double"
   }]
}
```

Figure 4.4 (continued)

The extracted information for each method includes its name, its return type, any modifiers and thrown exceptions, as well as name and type of each of its parameters. Additionally, the body of the method is saved as a string value in the JSON representation. Although the body of the method contains quite useful information, using it as is to match result methods to desired algorithms is not straightforward. In our system, we parse the method body and extract all its statements. After that, the variable declaration and variable assignment/instantiation statements are parsed so that the variable names and types of the method are extracted. Although this representation may seem simplistic, it is actually quite reasonable, since most algorithms may use several properties of objects that are not however given as parameters. For instance, in the case of a "document" with "id", "title" and "text", a "wordWrap" method may receive as parameter a document object, and then use the text variable only in the body implementation.

4.2.3 Matcher

Upon having extracted the properties of the resource and the structural information from the result files, the matcher component is used to rank the results according to their compliance to the query. This is accomplished by assigning a similarity value to each result file. For this purpose, we have created a similarity scheme between the properties and a method. The similarity scheme performs a 1-1 matching between the properties of the resource and the parameters and variables of the method.

Thus, we can define the problem as a matching problem between two objects A and B. Object A has name equal to the resource name and its variables are the properties of the resource. Object B has name equal to a method name, and its variables is the union of the parameters of the method and the variables that are defined in the method. Given these two objects A and B, we define their similarity as:

$$S(A,B) = c_{name} \cdot StringSimilarity\left(name^{A}, name^{B}\right) + VariableSimilarity\left(V_{A}, V_{B}\right) \qquad (4.1)$$

The first term refers to the string similarity between the names of the objects. Since Java naming conventions include camelCase, the two strings are initially split according to camelCase and underscores, and then the two sets of strings are compared. The function *StringSimilarity* returns the size of the union of these two sets divided by the maximum of the sizes of the sets. For instance, given strings "addBookmark" and "createBookmark", their derived sets are {"add", "bookmark"} and {"create", "bookmark"}, thus their string similarity is 1/2 = 0.5. Finally, c_{name} is a weighting parameter indicating the importance that the two objects have similar names.

VariableSimilarity refers to the best possible matching between the variables of the two objects, V_A and V_B. Two variables u and v, corresponding to the objects V_A and V_B respectively, are matched using the function:

$$score(u,v) = \begin{cases} c_{vname} \cdot ns(u,v) + c_{vtype} \cdot ts(u,v) & \text{if } ns(u,v) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

where

$$ns(u,v) = StringSimilarity(name^u, name^v)$$
$$ts(u,v) = StringSimilarity(type^u, type^v)$$

and c_{vname} and c_{vtype} are weighting parameters indicating the importance of similar names and types of the variables, respectively. Thus, given two sets of variables V_A and V_B, the function *VariableSimilarity* finds the best possible matching according to the algorithm of Figure 4.5.

```
$VariableSimilarity(V_A, V_B)$
    $MatchedV_A = \{\}$
    $MatchedV_B = \{\}$
    $MatchedPairs = \{\}$
    $Pairs = \{(u, v, score(u,v)) \forall u \in V_A, v \in V_B\}$
    Sort $Pairs$ according to the highest score
    $TotalScore = 0$
    for each $(u, v, score(u,v)) \in Pairs$ :
        if $u \notin MatchedV_A$ and $v \notin MatchedV_B$
            $MatchedPairs = MatchedPairs \cup \{(u, v, score(u,v))\}$
            $TotalScore = TotalScore + score(u,v)$
    return $TotalScore$
```

Figure 4.5 Algorithm that computes the similarity between two sets of variables

The algorithm receives the sets V_A and V_B as input and outputs the *TotalScore*, i.e. the score of the best matching between the sets. At first, two sets are defined, $MatchedV_A$ and $MatchedV_B$, to keep track of the matched variables of the two objects. After that, all possible scores for the combinations of the variables of the two methods are computed and stored in the *Pairs* set in the form $(u, v, score(u,v))$, where u and v are variables of objects A and B respectively. The *Pairs* set is sorted in descending order, and then the algorithm iterates over each pair of the set. For each pair, if neither of the variables is already present in the $MatchedV_A$ and $MatchedV_B$ sets, then the pair is matched, added to the $MatchedPairs$ set, and the $TotalScore$ is incremented by the score of the pair.

Finally, for each result file the method with the maximum score is presented to the user as a candidate algorithm. Additionally, all scores are normalized in the range $[0,1]$ by dividing them with the maximum score, i.e. the score $S(A,A)$. The parameters c_{name}, c_{vname}, and c_{vtype} can be set by the user. In our analysis, we set the parameters to values 4, 1, and 0:5 respectively.

4.3 Case Study

In this section, we provide a case study for a resource and possible algorithms for the given resource. We use the resource "document" with an "id", a "title", and a "text". In the following subsections we present the results of different algorithmic queries that could be useful for this resource.

4.3.1 Algorithm for Wrapping Text

One of the most useful text algorithms involves wrapping words to enhance the presentation of the text. In the context of our analysis, we use the algorithmic resource "wordWrap" and apply the methodology of the previous Section. The results for this algorithm are shown in Table 4.1.

Table 4.1 20 first results for the query "wordWrap"

#	Method	Score	Relevant
1	mxUtils.wordWrap	0.588	Yes
2	GroovyTemplateEngine.wrap	0.412	No
3	HtmlDomParserContext.wrapDocument	0.353	No
-	HtmlDomParserContext.wrapDocument	0.353	No
-	Impl.wrap	0.353	No
4	XmlDomParserContext.wrapDocument	0.294	No
-	XmlDomParserContext.wrapDocument	0.294	No
-	GoHyperlinkDetector.findWord	0.294	No
-	YamlHyperlinkDetector.findWord	0.294	No
-	XmlHyperlinkDetector.findWord	0.294	No

-	StringHyperlinkDetector.findWord	0.294	No
-	PyDoubleClickStrategy.selectWord	0.294	No
-	RPrintUtilities.printDocumentWordWrap	0.294	Yes
-	PythonWordFinder.findWord	0.294	No
-	WordFinder.findWord	0.294	No
-	WordFinder.findWord	0.294	No
5	LagartoParser.textWrap	0.275	Yes
-	XmlDocument.wrapJavaDocument	0.275	No
-	XmlDocument.wrapJavaDocument	0.275	No
6	OOXMLParserTest.testWord	0.235	No

Upon examining the results, we can see that 3 of them are highly relevant to the original query, while at the same time offering algorithmic implementations of value to the user. In this case, the ranking is also quite effective, given that the first result, i.e. the one with the highest score, is relevant. Notably, the other two relevant results are in the positions 4 and 5, however there are several results in the same positions.

4.3.2 Algorithm for Highlighting a Word

Another common algorithm for a document would be to highlight a word. The query involves the algorithmic resource "highlightWord". The results are shown in Table 4.2.

Table 4.2 20 first results for the query "highlightWord"

#	Method	Score	Relevant
1	HighlighterTest.highlightField	0.353	No
-	AnalyzingInfixSuggester.highlight	0.353	Yes
-	mxUtils.wordWrap	0.353	No
2	UIPrefsAccessor.highlightSelectedWord	0.314	No
3	BaseWebInspector.refresh	0.294	No
-	DOM.highlightNode	0.294	No
-	ReadTask.doLogic	0.294	No

-	GoHyperlinkDetector.findWord	0.294	No
-	YamlHyperlinkDetector.findWord	0.294	No
-	XmlHyperlinkDetector.findWord	0.294	No
-	StringHyperlinkDetector.findWord	0.294	No
-	PyDoubleClickStrategy.selectWord	0.294	No
-	PythonWordFinder.findWord	0.294	No
-	WordFinder.findWord	0.294	No
-	WordFinder.findWord	0.294	No
4	OOXMLParserTest.testWord	0.235	No
-	WebView.handleQueuedTouchEventData	0.235	No
-	WebView.handleQueuedTouchEventData	0.235	No
-	WebViewClassic.getMaxTextScrollX	0.235	No
-	VocabCreator.validWord	0.235	No

As in the previous query, the first relevant results is also ranked in the first position. In this case, however, we can see that there are not a lot of relevant results. This is due to the naming of the method; usually most algorithms would highlight a word, instead they would find a word in some text and then highlight the relevant fragments. Our methodology, however, managed to provide some implications as to the query. Given the results of Table 4.2, one may notice that several of them refer to functions for finding a word. In the following subsection, we examine the relevant query for word finding.

4.3.3 Algorithm for Finding a Word

Searching inside a document to find whether some word exists, or also its position, is also one of the most usual algorithms. The query includes the term "findWord". The results are shown in Table 4.3.

Table 4.3 20 first results for the query "findWord"

#	Method	Score	Relevant
1	GoHyperlinkDetector.findWord	0.529	Yes
-	YamlHyperlinkDetector.findWord	0.529	Yes

-	XmlHyperlinkDetector.findWord	0.529	Yes
-	StringHyperlinkDetector.findWord	0.529	Yes
-	WordFinder.findWord	0.529	No
-	PythonWordFinder.findWord	0.529	Yes
-	WordFinder.findWord	0.529	Yes
2	mxUtils.wordWrap	0.353	No
-	DependencyTree.findDependency	0.353	No
-	Cluster.find	0.353	No
3	SamlModel.findProtocolSignatureElement	0.333	No
4	DataKeeperServiceImpl.findDocumentByDocumentId	0.329	No
-	DataKeeperServiceImpl.findDocumentByDocumentId	0.329	No
5	InputHandler.actionPerformed	0.314	No
6	RewriteCustomTheme.findCustomThemeSheet	0.294	No
-	MongoCollection.findOne	0.294	No
-	MongoDBClient.find	0.294	No
-	StdCouchDbConnector.find	0.294	No
-	SLD3DParser.findElements	0.294	No
-	InlineLocalVariableActionHandler.findUsage	0.294	No

In this case, most of the results at the top of the list are indeed relevant. Note also that these results have rankings more than 0.5, indicating high similarity with the properties of the “document” resource.

As a concluding remark, our methodology seems effective for finding example algorithmic implementations. Furthermore, the algorithms conform to the model of the RESTful resource.

5 Mining UML Models

5.1 Overview

In this Section we present the procedure of parsing use case diagrams and activity diagrams into models as well as a matching scheme to find similar use case and activity diagrams. By examining similar diagrams, the requirements engineer can get useful ideas regarding the models of his/her project. In subsection 5.2 we describe our methodology for finding similar use case diagrams, while in subsection 5.3 we show how similar activity diagrams are found by our system. Both subsections include examples of matching diagrams.

5.2 Detecting Similar Use Case Diagrams

5.2.1 Parsing Use Case Diagrams

UML Use case diagrams have two types of elements: actors and use cases. Additionally, they have certain associations, including "extend" and "include" between use cases, "generalization" between actors, and relations between actors and use cases. An example use case diagram of project Restmarks is shown in Figure 5.1.

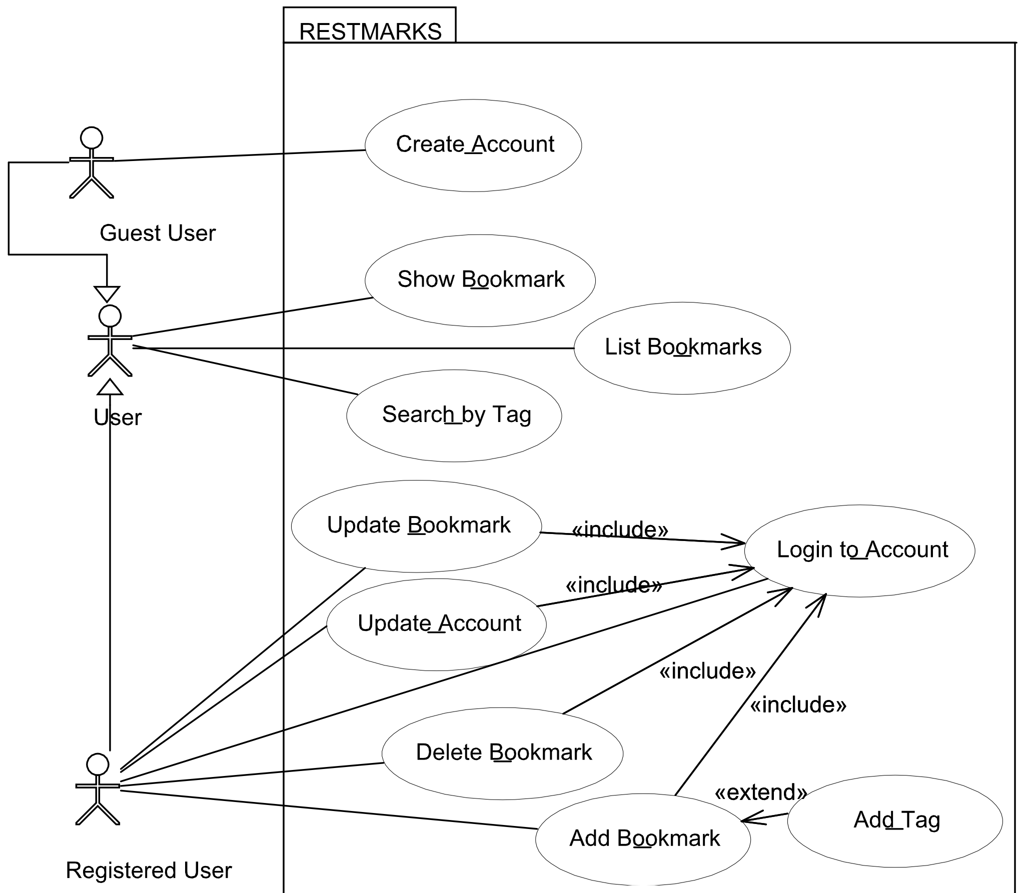

Figure 5.1 Example use case diagram for project Restmarks

The use case diagram of the above Figure contains 10 use cases and 3 actors. We created a parser in order to effectively read the XMI files of use case diagrams into a model. Our system supports the XMI format of the Papyrus UML tool [46].

Since use cases refer to static information, our model is mostly flat. In specific, it contains the actors and the use cases of the diagram. Thus, for the example of Figure 5.1 the model consists of two sets, the set of actors `{User, Registered User, Guest User}` and the set of use cases `{Add Bookmark, Update Bookmark, Update Account, Show Bookmark, Search by Tag, Add Tag, Login to Account, Delete Bookmark}`.

5.2.2 Matching Use Case Diagrams

Upon having parsed the use case diagrams, in this subsection we present a matching scheme for finding similar diagrams. Given two diagrams D_1 and D_2, the matching scheme involves two sets for each diagram, one set for the actors A_1 and A_2 respectively, and one for the use cases UC_1 and UC_2 respectively. The similarity between two diagrams is given as follows:

$$s(D_1,D_2)=\alpha\cdot s(A_1,A_2)+(1-\alpha)\cdot s(UC_1,UC_2) \tag{5.1}$$

where s denotes the similarity between two sets and α is a parameter that denotes the importance of the similarity of the actors in the diagram. In our case, we set α to the proportion of the actors divided by the use cases of the queried diagram. Given e.g. a diagram with 3 actors and 10 use cases, α is set to 0.3.

The similarity between two actors or between two use cases is given by the combination between all the matched elements with the highest score. Thus for example, given two sets $\{user, administrator, guest\}$ and $\{administrator, user\}$, the best possible combination is $\{(user,user),(administrator,administrator),(guest,null)\}$. The matching in this case would return a score of 2/3, i.e. 0.66. The similarity between two strings is given using the semantic measure defined in paragraph 3.2.2. In specific, given two strings S_1 and S_2, we first split each string into tokens, i.e. $tokens(S_1)=\{t_1,t_2\}$ and $tokens(S_2)=\{t_3,t_4\}$ and then find the best possible combination of tokens, i.e. the one with the maximum token scores. After that, the scores for all tokens are averaged to provide a similarity score for the two strings in the range [0, 1]. For example, given the strings "Get bookmark" and "Retrieve bookmarks", the best combination is ("get", "retrieve") and ("bookmark", "bookmarks"). The semantic similarity between "get" and "retrieve" according to the scheme of paragraph 3.2.2 is 0.677. The similarity of "bookmark" with "bookmarks" is 1.0. Thus the final similarity between the two strings is the average between these two scores, i.e. (0.677 + 1) / 2 = 0.8385.

5.2.3 Example

In this subsection we provide an example output of our system for two use case diagrams. Note that given a database of use case diagrams, one could use our system to find similar ones in order possibly to reuse some elements and get examples or useful ideas. In this example, we assume that the user would have searched for similar diagrams and then our system would return the matching information about the top diagram(s). For this example, we depict the matching between the diagram of Figure 5.1 and the diagram of Figure 5.2.

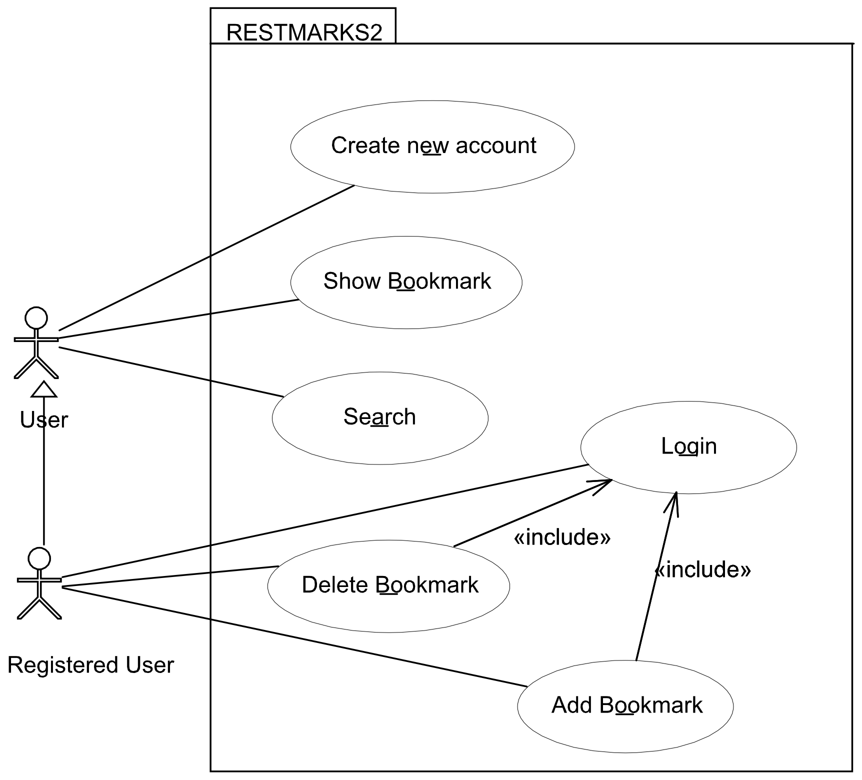

Figure 5.2 Example use case diagram for matching with the one of Figure 5.1

The matching score between the two diagrams is 0.692. Our system also returns the matching between the two diagrams, shown in Table I.

Table 5.1 Matching between the diagrams of Figure 5.1 and Figure 5.2

Diagram 1	**Diagram 2**	**Score**
User	User	1.00
Registered User	Registered User	1.00
Guest User	null	0.00
Delete Bookmark	Delete Bookmark	1.00
Show Bookmark	Show Bookmark	1.00
Add Bookmark	Add Bookmark	1.00
Create Account	Create new account	0.66
Search by Tag	Search	0.33

Login to Account	Login	0.33
List Bookmarks	null	0.00
Update Bookmark	null	0.00
Update Account	null	0.00
Add Tag	null	0.00

As shown in this Table, the matching between the actors indicates that the requirements engineer (of the second diagram) could consider adding a guest user. Additionally, the use cases for listing and updating bookmarks, for updating account information and for adding tags to bookmarks could be added to the requirements of the (second) system.

5.3 Detecting Similar Activity Diagrams

5.3.1 Parsing Activity Diagrams

Activity diagrams depict the flow of actions in dynamic scenarios of software projects. As such, we required a representation that would take the flows of the diagram into account. An example activity diagram of project Restmarks is shown in Figure 5.3.

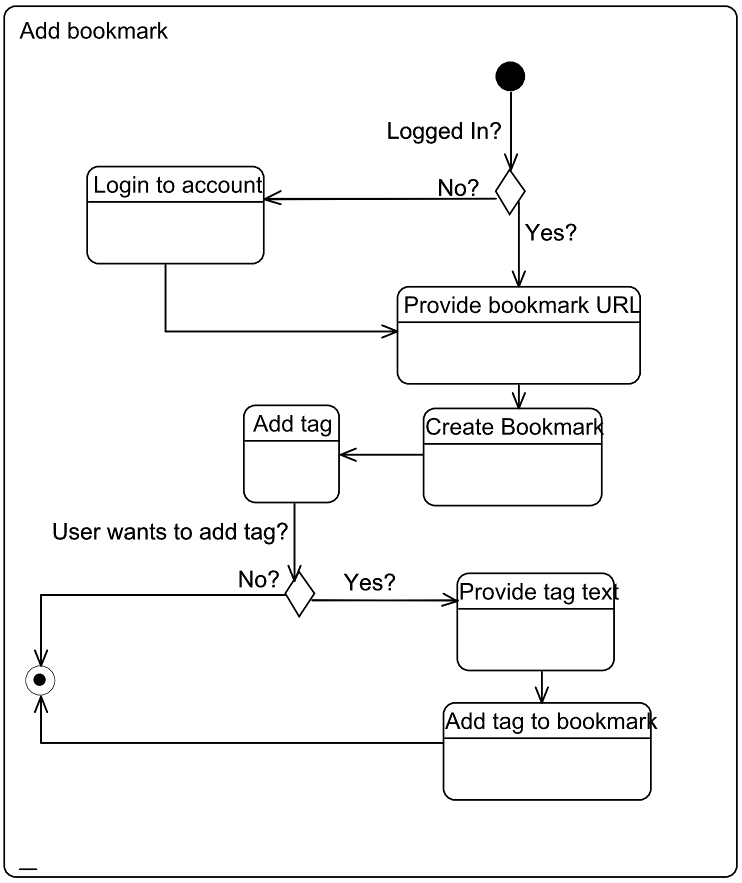

Figure 5.3 Example activity diagram for project Restmarks

As shown in this diagram, activity diagrams are mostly sequences of activities and conditions. In the case of conditions (or forks and joins), the flow of the diagram is split, thus one can view an activity diagram as a set of sequences.

In the context of our work, we analyze each activity diagram into a set of sequences. Each sequence denotes a set of activities required to traverse from the start node to the end node of the diagram. For instance, the diagram of Figure 5.3 spawns the following set of sequences:

- `StartNode > Logged In? > Login to account > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > EndNode`
- `StartNode > Logged In? > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > EndNode`
- `StartNode > Logged In? > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > Provide tag text > Add tag to bookmark > EndNode`
- `StartNode > Logged In? > Login to account > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > Provide tag text > Add tag to bookmark > EndNode`

Thus, given these sequences, the problem is reduced to finding the most similar ones.

5.3.2 Matching Activity Diagrams

Upon having parsed the activity diagrams and having extracted the sequences for each one, in this subsection we present a matching scheme for these sequences. Thus, the similarity between two diagrams is the similarity between the sets of their sequences. The similarity between two sets of sequences is given by the combination between all the matched sequences with the highest score. Thus for example, given two sets $\{[a,b,e],[a,b,d,e],[a,b,c,e]\}$ and $\{[a,b,e],[a,c,e]\}$, the best possible combination is $\{([a,b,e],[a,b,e]),([a,b,c,e],[a,c,e]),([a,b,d,e],null)\}$.

The similarity between two sequences is based on their Longest Common Subsequence (LCS) [45]. Given two sequences X and Y, their LCS is defined as the longest subsequence of which the elements are not necessarily consecutive that is common to both sequences. For example, given the sequences $X=[a,b,d,e,g]$ and $Y=[a,b,e,h]$, their LCS is $LCS(X,Y)=[a,b,e]$. The score between two sequences is defined using their LCS as:

$$sim(X,Y)=2\cdot\frac{|LCS(X,Y)|}{|X|+|Y|} \tag{5.2}$$

The above equation ensures that the score is normalized in the range $[0,1]$. Finally, in contrast with the matching of actors or use cases, we set a threshold t for the string similarity metric to form a binary decision denoting whether two strings are similar. If the similarity score of the two compared strings is larger than the threshold, then the two strings are considered similar. We set t to 0.5.

5.3.3 Example

In this subsection we provide an example output of our system for two UML activity diagrams. For this example, we depict the matching between the diagram of Figure 5.3 and the diagram of Figure 5.4.

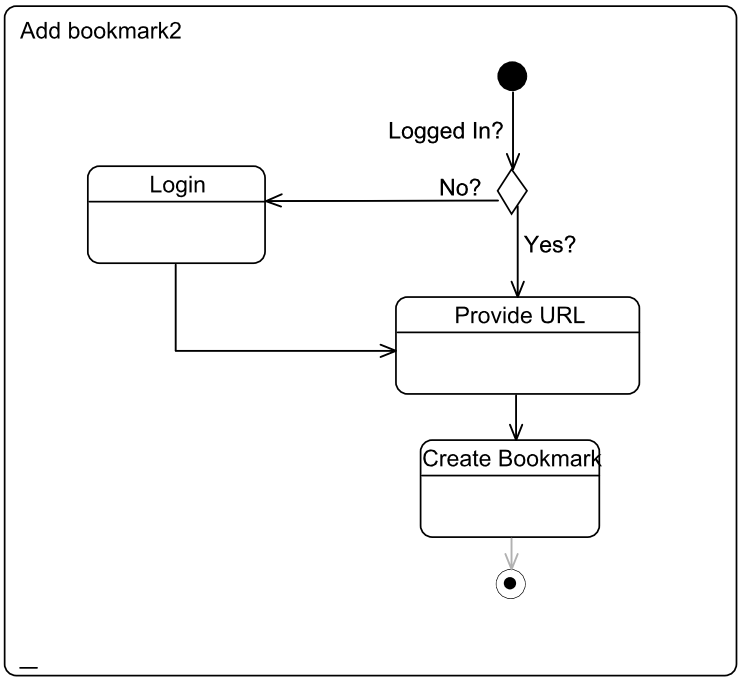

Figure 5.4 Example activity diagram for matching with the one of Figure 5.3

The matching score between the two diagrams is 0.387. Our system also returns the matching between the two diagrams, shown in Table 5.2.

Table 5.2 Matching between the diagrams of Figure 5.3 and Figure 5.4

Diagram 1	**Diagram 2**	**Score**
StartNode > Logged In? > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > EndNode	StartNode > Logged In? > Provide URL > Create Bookmark > EndNode	0.833
StartNode > Logged In? > Login to account > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > EndNode	StartNode > Logged In? > Login > Provide URL > Create Bookmark > EndNode	0.714
StartNode > Logged In? > Provide bookmark URL > Create Bookmark >	null	0.000

Add tag > User wants to add tag? > Provide tag text > Add tag to bookmark > EndNode		
StartNode > Logged In? > Login to account > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > Provide tag text > Add tag to bookmark > EndNode	null	0.000

As shown in this Table, the matching between the sequences indicates that the requirements engineer (of the second diagram) could consider adding a new flow that would include the possibility that the user would like to add a tag to his/her newly created bookmark.

5.4 Comparison with the Current State-of-the-Art

As noted in Sections 2.4 and 2.5, although current literature on UML diagrams is broad, there is a common methodology that involves extracting models from UML diagrams and mining these models using a variety of methods.

Various types of methods have been developed for computing the similarity of UML diagrams, including Information Retrieval techniques [25], [26], Graph based methods [27]–[34], and even ontology based, i.e. semantics enabled, methods [35]–[37]. However, these methods are usually too generic to conform to the special characteristics of UML diagrams. For instance, the lexical approach of Information Retrieval techniques [25], [26] is not effective for diagrams that incorporate structure or flow information. Graph based methods [27]–[34], on the other hand, can be effective for certain types of structured diagrams (e.g. class diagrams), however, as noted also by Kelter et al. [31], these methods do not exploit the semantics of the models, since they only employ arbitrary graph similarity metrics. Finally, in cases where domain specific information is limited, ontology based methods [35]–[37] cannot be applied.

An interesting alternative to the aforementioned approaches involves considering UML models as ordered trees. Differencing ordered trees that are derived from structured representations, such as XML documents, is a widely known problem, for which several algorithms have been developed [32], [33]. In the case of UML diagrams, the problem lies in defining and populating the underlying data model that holds the elements of the diagrams. Designing the data model can be crucial to the overall effectiveness of the similarity matching algorithm.

Upon having designed two difference algorithms, in Sections 5.2 and 5.3, for Use Case and Activity diagrams respectively, in this Section we compare them with a widely known approach in the area of difference algorithms for UML models, defined in [31]. The work in [31] is part of the FUJABA Project [47], an open source CASE tool that is meant to support developers in model-based software engineering and re-engineering tasks. In [31], the authors define a data model that is generic enough to support several types of UML diagrams (and even non-UML models, such as generic XML documents).

This approach was selected for a number of reasons. At first, as an ordered tree approach, it offers a solid middle ground between unstructured (e.g. text) and heavily structured (i.e. graph) methods. Given also the current state-of-the-practice, ordered tree approaches are extensively used for tree-like structures in several tools (including e.g. Eclipse matching on xml or ecore files). In addition, it employs a complete data model which includes all diagram (or model) information as our algorithms do. In specific, the underlying data model supports different types of models, including Use Case and Activity diagrams, and allows the application of different similarity measures for string or structural elements. The rest of this subsection presents the proposed data model of [31] and illustrates the related aspects to our methodology.

The proposed data model is shown in Figure 5.5.

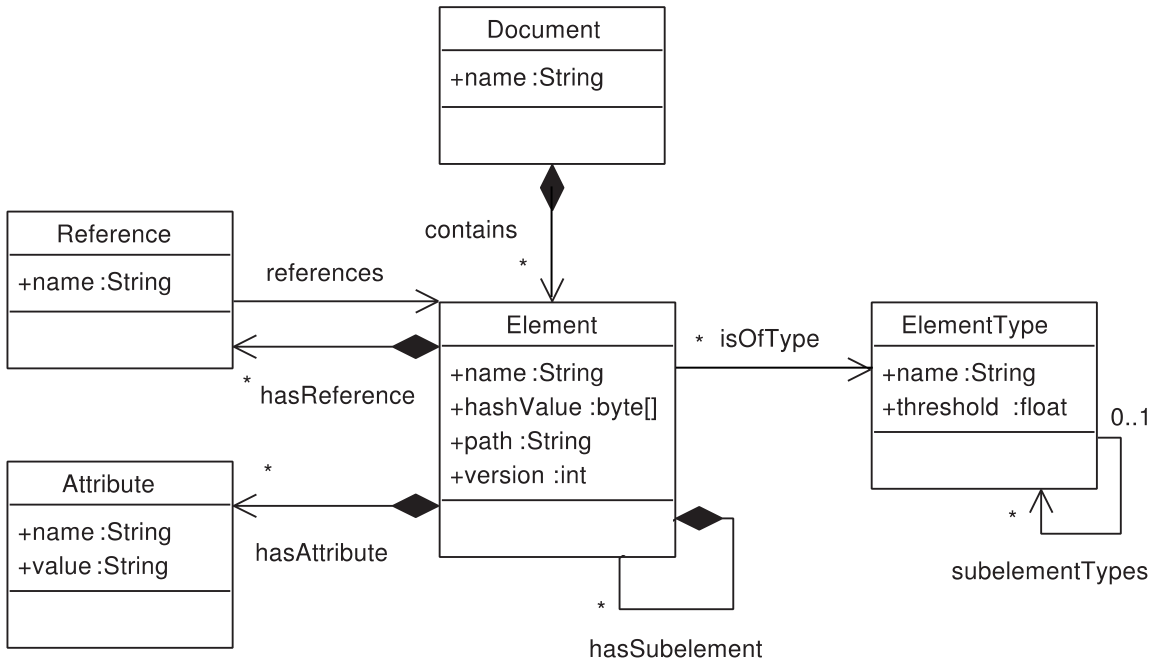

Figure 5.5 Data model of the difference algorithm, as shown in [31]

As shown in Figure 5.5, the data model consists of the following elements: a Document, a set of Elements, each one with its type (ElementType), a set of Attributes for each Element, and a set of References between elements. Kelter et al. [31] focus on UML Class diagrams, where examples of Elements include classes, parameters, etc. Note also that the types of the Elements have hierarchy, which is quite optimal for class diagrams since they are designed with a notion of hierarchy; e.g. packages include classes, classes include methods, etc.

The procedure of finding the similarity between two diagrams is two-step. The first step includes the instantiation of the model of Figure 5.5 for each diagram. After that, the two instantiated models are compared against each other using a simple top-down algorithm. The comparison between strings is performed using the LCS algorithm [45], which is an effective distance metric that however does not include any semantics.

Although the data model of Figure 5.5 is mainly oriented towards class diagrams, it is applicable also to several different types of UML diagrams, including Use Case and Activity diagrams. In the following subsections, we illustrate how the data model is applied to these types of diagrams. In specific, we assess our algorithms against three implementations of the data model and string metrics defined in the work of Kelter et al. [31].

For Use Case diagrams, the data model described by Kelter et al. [31] is actually the same as our data model (see Section 5.2). As a result, the assessment, illustrated in Section 5.4.1, is oriented towards the semantic string similarity metrics that we have developed.

For Activity Diagrams, the data model described by Kelter et al. [31] differs significantly from our data model (see Section 5.3). As a result, we implement a higher order comparison, to illustrate the different characteristics of the two data models, in Section 5.4.2.

The two approaches are thereafter referred to as the S-CASE approach and the FUJABA approach, which are the projects that they originated from.

5.4.1 Assessing the Matching of Use Case Diagrams

In the case of Use Case Diagrams the data model described in Figure 5.5 is quite simplified. In specific, the class/type instantiation of the model is shown in Table 5.3.

Table 5.3 Type instantiation of the data model shown in Figure 5.5 for Use Case diagrams

Meta-Class	Class Instantiation
Document	Use Case diagram
Element (with ElementType)	Use Case or Actor
Reference	includes, extends, associations and generalizations
Attribute	-

Use Case diagrams do not usually entail an order for the elements, since they are static representations of requirements. As a result, the data model is not fully exploited to result in an ordered tree, and is reduced to a set of elements (and a set of references). Therefore, the two data models, the one by Kelter et al. [31] and our own, described in this Section and Section 5.2 respectively, are equivalent. Upon applying the data model of Kelter et al. [31] using the LCS metric for the similarity between strings (and a threshold set to 0.5 as in our methodology), we compare the outputs of the two models.

We executed the algorithm for the two Use Case diagrams of Figure 5.1 and Figure 5.2. The result of the method is shown in Table 5.4.

Table 5.4 Matching between the diagrams of Figure 5.1 and Figure 5.2, using the FUJABA approach

Diagram 1	Diagram 2	Score
User	User	1.00

Registered User	Registered User	1.00
Guest User	null	0.00
Delete Bookmark	Delete Bookmark	1.00
Show Bookmark	Show Bookmark	1.00
Add Bookmark	Add Bookmark	1.00
Create Account	Create new account	0.81
Search by Tag	Search	0.63
Login to Account	Login	0.48
List Bookmarks	null	0.00
Update Bookmark	null	0.00
Update Account	null	0.00
Add Tag	null	0.00

As shown in this Table and in comparison with the matching shown in Table 5.1, the two methods have very similar results. This is expected since the two data models are almost identical, while the semantic characteristics of our algorithm are not shown in this scenario.

A more semantic-oriented scenario can be designed by modifying the names of the use cases for the two diagrams. In specific, for a more complex scenario, we modify the Use Case diagram of Figure 5.2, so that it now contains the use cases “Retrieve Bookmark” and “Remove Bookmark” in the place of “Show Bookmark” and “Delete Bookmark” respectively. The new results for the two methods are shown in Table 5.5.

Table 5.5 Matching between the diagrams of Figure 5.1 and Figure 5.2 (modified so that “Show Bookmark” and “Delete Bookmark” are replaced with “Retrieve Bookmark” and “Remove Bookmark” respectively), using the S-CASE approach and the FUJABA approach

	S-CASE		**FUJABA**	
Diagram 1	**Diagram 2**	**Score**	**Diagram 2**	**Score**
User	User	1.00	User	1.00
Registered User	Registered User	1.00	Registered User	1.00
Guest User	null	0.00	null	0.00

Delete Bookmark	Remove Bookmark	0.86	Retrieve Bookmark	0.75
Show Bookmark	Retrieve Bookmark	0.5	Remove Bookmark	0.71
Add Bookmark	Add Bookmark	1.00	Add Bookmark	1.00
Create Account	Create new account	0.66	Create new account	0.81
Search by Tag	Search	0.33	Search	0.63
Login to Account	Login	0.33	Login	0.48
List Bookmarks	null	0.00	null	0.00
Update Bookmark	null	0.00	null	0.00
Update Account	null	0.00	null	0.00
Add Tag	null	0.00	null	0.00

As shown in this Table, the semantic string similarity algorithm outperforms the simple LCS method. For instance, the semantic algorithm successfully matches the “Remove Bookmark” use case to the “Delete Bookmark” use case, and the matching is also given a relatively high score. On the other hand, the LCS matches “Remove Bookmark” to the “Show Bookmark” and the “Delete Bookmark” use case is matched to “Show Bookmark”. Although in terms of string distance similarity this matching is reasonable, in terms of semantics is certainly not effective.

5.4.2 Assessing the Matching of Activity Diagrams

In the case of Activity Diagrams the data model described in Figure 5.5 is instantiated as shown in Table 5.6.

Table 5.6 Type instantiation of the data model shown in Figure 5.5 for Activity diagrams

Meta-Class	Class Instantiation
Document	Activity diagram
Element (with ElementType)	Activity, Initial Node, Final Node, Decision Node
Reference	Edge
Attribute	-

Given that Activity diagrams do not impose hierarchy to the elements, the data model described by Kelter et al. [31] is not fully exploited as is the case for other types of diagrams, such as class diagrams. Activity diagrams represent the dynamic aspects of a software project, focusing for example on data flow, sequences of user or system actions (or both). As a result, the analysis performed on Activity diagrams is usually focused on the flow of actions, rather than the structure of the diagram elements, thus super-elements and sub-elements are not applicable in this model.

Furthermore, note that Activity diagrams (and Use Case diagrams) concern the requirements elicitation phase of the software project. Consequently, these diagrams usually do not include detailed attributes, as Class diagrams do. For instance, as noted by Kelter et al. [31], an example attribute of a class is its abstractness; similar attributes are not applicable to Activity diagrams.

As in Use Case diagrams, the FUJABA approach employs the LCS method for computing the similarity between strings. In this Section, however, the focus is to assess the differences between the data models, since the difference between the similarity techniques has been assessed in Section 5.4.1. Thus, we compare only the two data models, by manually selecting the matched strings for the two models.

Given the two Activity diagrams of Figure 5.3 and Figure 5.4, the result of our method is shown in Table 5.2, while the result of the FUJABA approach is shown in Table 5.7.

Table 5.7 Matching between the diagrams of Figure 5.3 and Figure 5.4, using the FUJABA approach

Diagram 1	Diagram 2	Score
Create Bookmark	Create Bookmark	1.00
Logged In?	Logged In?	1.00
StartNode	StartNode	0.00
EndNode	EndNode	1.00
Provide bookmark URL	Provide URL	0.71
Login to account	Login	0.48
Add tag	null	0.00
User wants to add tag?	null	0.00
Provide tag text	null	0.00
Add tag to bookmark	null	0.00

Comparing Table 5.2 to Table 5.7, one may draw useful conclusions about the advantages and the disadvantages of each model. At first, the representation shown in Table 5.2 is a better fit for the action flow of Activity diagrams. The abstraction employed by this model presents a higher order result to the user, indicating e.g. that he/she maybe should add a new flow of actions (i.e. a new functionality) to the original plan. Given the newly added flow of actions, the user is able to understand the purpose of each individual activity. The data model of Table 5.7 is highly useful for visualizing the output, since it isolates better the nodes (activities or conditions) that have to be added. This is expected since, as noted also by Kelter et al. [31], the visualization of the differences lies within the main scope of the design of this model. Thus, in structural (or static) diagrams, this data model can achieve several goals, including visualization or even diagram versioning. However, in dynamic types of diagrams, such as sequence or activity diagrams, our data model is more effective on entailing higher order semantics and presenting them to the requirements engineer.

5.5 Experimental Evaluation

Upon illustrating the differences between our methodology and the current state-of-the-art, in this Section we further assess the effectiveness of our approach using a dataset of UML diagrams. The dataset originates from the one used in Deliverable D3.3.2 of S-CASE and includes 65 Use Case diagrams and 72 Activity diagrams. These diagrams belong to several software projects with different semantic characteristics. In the context of our evaluation, we may classify each diagram in one of the following categories:

- Diagrams including health and mobility terms, which originate mostly from the requirements of health systems
- Diagrams with traffic and transportation terms, including projects that may relate to routing, traffic and transportation analysis, etc.
- Diagrams that are relevant to social networks and generally p2p communications, which refer mostly on social network and communication projects
- Diagrams that refer to the use of services, including mostly common processes for creating/modifying accounts, issuing requests for products, etc.
- Business process diagrams, which refer to the specifics of running a business (e.g. diagrams that explain the procedure of creating and internally reviewing reports etc.), and may originate from different projects
- The rest of the diagrams, including most of the diagrams created specifically for S-CASE

Note that the above categorization is mostly semantic; however structural equivalence between diagrams of the same category is also expected, since most use cases and action flows are common in these types of systems.

Upon constructing all the possible pairs of use case and activity diagrams, which are 2080 and 2556 pairs respectively, we mark each pair of diagrams as relevant or non-relevant according to their categories. In specific, any pair consisting of diagrams of the same category is considered relevant, while and all other pairs of diagrams are regarded as non-relevant.

In the following subsections, we present the results of our evaluation on this dataset, upon executing our method against the method of Kelter et al. [31]. For each method we find the average similarity value for all diagram pairs, and consider as relevant the pairs that surpass this value, while all other pairs are considered as irrelevant.

5.5.1 Evaluating the Matching of Use Case Diagrams

The execution of the two methods on Use Case diagrams provides an interesting assessment, given that the S-CASE approach and the FUJABA approach are structurally similar, as discussed in section 5.4. Hence, the results of the evaluation on Use Case diagrams are expected to determine the effectiveness of the semantic methodology of our approach. The results of executing the two approaches on the pairs of use case diagrams are shown in Table 5.8 and Table 5.9, for S-CASE and FUJABA respectively.

Table 5.8 Classification results of S-CASE for the Use Case diagrams of the dataset

Class Label	Precision	Recall	F-Measure
Irrelevant Diagram Pairs	0.806	0.633	0.709
Relevant Diagrams Pairs	0.348	0.562	0.430
Average	**0.688**	**0.614**	**0.637**

Table 5.9 Classification results of FUJABA for the Use Case diagrams of the dataset

Class Label	Precision	Recall	F-Measure
Irrelevant Diagram Pairs	0.786	0.561	0.655
Relevant Diagrams Pairs	0.308	0.561	0.397
Average	**0.662**	**0.561**	**0.588**

As shown in these tables, the S-CASE approach outperforms the FUJABA approach when it comes down to finding relevant diagram pairs. In specific, both systems have similar recall for relevant pairs; however our approach provides more accurate results, since its precision is clearly higher. Furthermore, non-relevant pairs are successfully isolated by our method, given that the values of precision and recall for this class label are also higher than the values of the FUJABA approach. The results are also visualized in Figure 5.6, where it is clear that the S-CASE approach outperforms the FUJABA approach for the metrics of precision, recall, and F-measure.

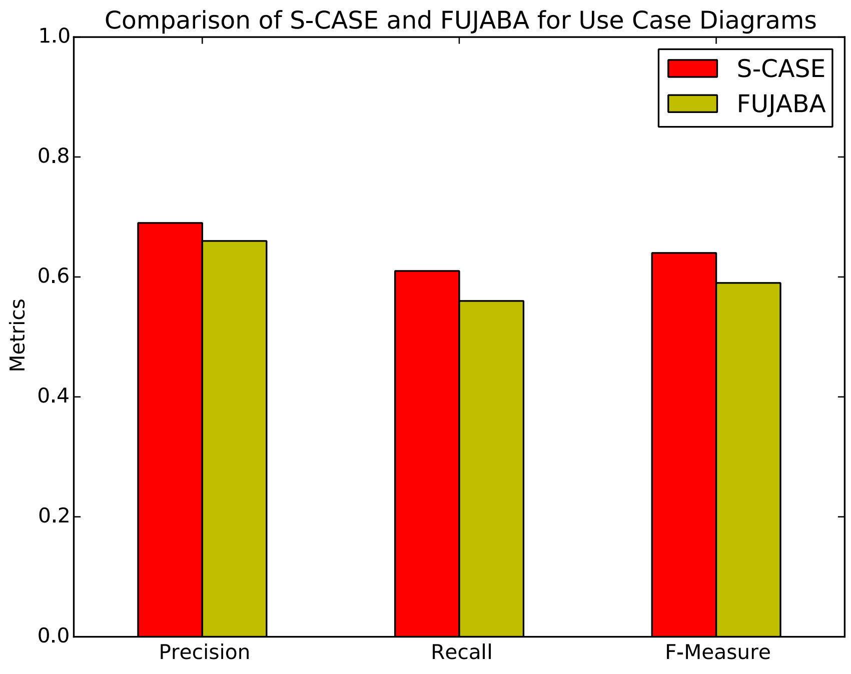

Figure 5.6 Evaluation metrics for the Use Case diagrams of the dataset

5.5.2 Evaluating the Matching of Activity Diagrams

Concerning Activity diagrams, the S-CASE approach and the FUJABA approach both have their pros and cons, as discussed in section 5.4. In this section, our evaluation is focused on the structural aspects of the diagrams. The results of our evaluation on the Activity diagrams of the dataset are summarized in Table 5.10 and Table 5.11, for the S-CASE approach and the FUJABA approach respectively.

Table 5.10 Classification results of S-CASE for the Activity diagrams of the dataset

Class Label	Precision	Recall	F-Measure
Irrelevant Diagram Pairs	0.698	0.619	0.656
Relevant Diagrams Pairs	0.346	0.430	0.384
Average	**0.586**	**0.559**	**0.569**

Table 5.11 Classification results of FUJABA for the Activity diagrams of the dataset

Class Label	Precision	Recall	F-Measure
Irrelevant Diagram Pairs	0.690	0.515	0.590
Relevant Diagrams Pairs	0.329	0.507	0.399
Average	**0.575**	**0.513**	**0.529**

As shown in these tables, S-CASE approach outperforms the FUJABA approach in terms of average precision, recall and F-measure. However, the results for each class label are diverse. In specific, the FUJABA approach seems to effectively detect several similar activity diagrams, given that the recall of relevant diagram pairs is quite high. However, the precision for this class label is lower than that of S-CASE. This is actually expected, since the structural model of FUJABA is based on elements with connections while the model of S-CASE is based on action flows. Thus, FUJABA issues higher similarity scores for certain diagrams even if they are not equivalent in terms of the flow of actions. This results in higher false positive rates, which is clear not only from the low precision of relevant diagram pairs, but also the relatively low recall of the irrelevant diagram pairs.

The S-CASE approach, on the other hand, has higher metric values on the class label of irrelevant diagram pairs, indicating that its false positives are reduced. Thus, the S-CASE approach has a clearly higher F-measure value for irrelevant diagram pairs, while the F-measure for the two approaches is similar.

The results for the averaged values of the metrics for the two approaches are also visualized in Figure 5.7, where it is clear that S-CASE outperforms FUJABA.

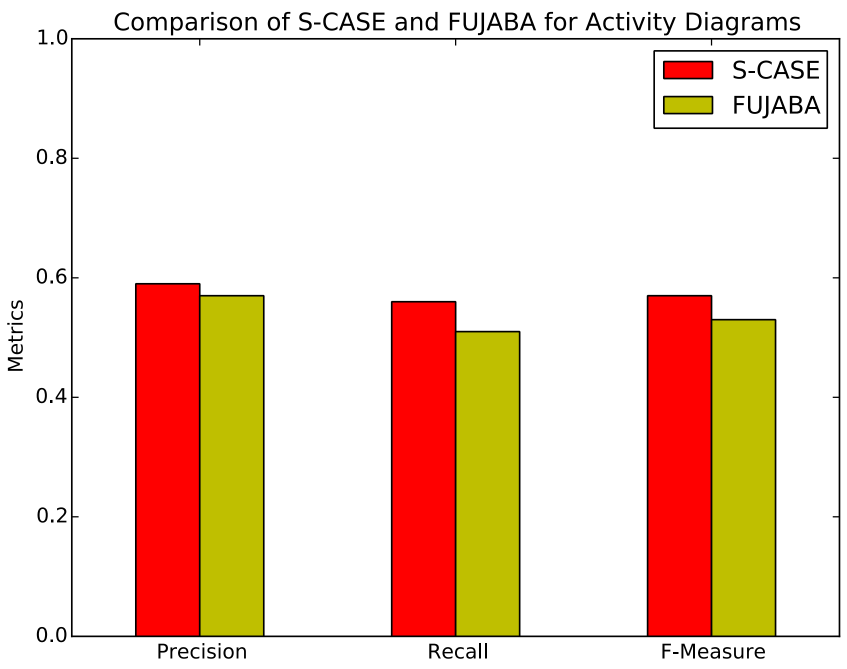

Figure 5.7 Evaluation metrics for the Activity diagrams of the dataset

6 Conclusions

The work discussed in this deliverable summarizes our progress on Task 2.4 of WP2 of S-CASE, which comprises applying mining techniques in three fields of study: functional requirements, source code, and UML diagrams. Although several research efforts have focused on these areas, most of these efforts do not conform to the special characteristics of our scenario. In specific, requirements elicitation recommendation systems are oriented towards high-level requirement representations and/or are dependent on domain specific knowledge. Source code mining techniques are not adapted to RESTful systems, while the area of UML model mining techniques can be greatly improved by the use of semantics and structural information.

Concerning functional requirements elicitation, our system employs association rule mining to extract useful rules from functional requirements, and uses the rules to create recommendations based on the project under development. As we have shown in Section 4, our system can provide a set of recommendations out of which approximately 60% of them will be rational. Thus, given that the requirements engineer (or a stakeholder) has compiled a set of requirements, he/she could effectively check whether he/she has omitted any important ones.

In the area of source code component-reuse systems, we have proposed a methodology that involves using the model of RESTful resources and employing CSE technology to recommend example algorithms that can be used to aid the implementation of algorithmic resources. As shown in Section 5, our methodology is effective for finding potential algorithms that conform to the RESTful resource model.

Finally, our work on UML diagram mining involves using structural representations and semantics to find similar diagrams. The use of distance metrics between sets and semantics for use case diagram matching ensures that the similarity between diagrams is computed in a meaningful manner. Additionally, since activity diagrams are represented as sequences of action flows, the dynamic features of the diagrams are taken into account in order to provide effective recommendations of similar diagrams.

References

[1] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9th edition, 2010.

[2] Dean Leffingwell. Calculating your return on investment from more effective requirements management. *American Programmer*, 10(4):13–16, 1997.

[3] Alexander Felfernig, Monika Schubert, Monika Mandl, Francesco Ricci, and Walid Maalej. Recommendation and decision technologies for requirements engineering. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 11–15, New York, NY, USA, 2010. ACM.

[4] Gunther Ruhe and Moshood Omolade Saliu. The art and science of software release planning. *IEEE Softw.*, 22(6):47–53, November 2005.

[5] William Frakes, Ruben Prieto-Diaz, and Christopher Fox. Dare: Domain analysis and reuse environment. *Ann. Softw. Eng.*, 5:125–141, January 1998.

[6] Manish Kumar, Nirav Ajmeri, and Smita Ghaisas. Towards knowledge assisted agile requirements evolution. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 16–20, New York, NY, USA, 2010. ACM.

[7] Smita Ghaisas and Nirav Ajmeri. Knowledge-assisted ontology-based requirements evolution. In *Managing Requirements Knowledge*, pages 143–167. Springer Berlin Heidelberg, 2013.

[8] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. An approach to constructing feature models based on requirements clustering. In Proceedings of the 13th IEEE International Conference on Requirements Engineering, RE '05, pages 31–40, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummler. An exploratory study of information retrieval techniques in domain analysis. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 67–76, Washington, DC, USA, 2008. IEEE Computer Society.

[10] Horatiu Dumitru, Marek Gibiec, Negar Hariri, Jane Cleland-Huang, Bamshad Mobasher, Carlos Castro-Herrera, and Mehdi Mirakhorli. Ondemand feature recommendations derived from mining public product descriptions. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 181–190, New York, NY, USA, 2011. ACM.

[11] Jose Romero-Mariona, Hadar Ziv, and Debra J. Richardson. Srrs: A recommendation system for security requirements. In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, RSSE '08, pages 50–52, New York, NY, USA, 2008. ACM.

[12] Soo Ling Lim and Anthony Finkelstein. Stakerare: Using social networks and collaborative filtering for large-scale requirements elicitation. *IEEE Trans. Softw. Eng.*, 38(3):707–735, May 2012.

[13] Carlos Castro-Herrera, Chuan Duan, Jane Cleland-Huang, and Bamshad Mobasher. Using data mining and recommender systems to facilitate large-scale, open, and inclusive requirements elicitation processes. In *Proceedings of the 2008 16th IEEE International Requirements Engineering Conference*, RE ’08, pages 165–168, Washington, DC, USA, 2008. IEEE Computer Society.

[14] Bamshad Mobasher and Jane Cleland-Huang. Recommender systems in requirements engineering. *The AI magazine*, 32(3):81–89, 2011.

[15] Alexander Felfernig, Monika Schubert, Monika Mandl, and P. Ghirardini. Diagnosing inconsistent requirements preferences in distributed software projects. In *Software Engineering 2010 - Workshopband*, volume 160 of LNI, pages 495–502. GI, 2010.

[16] Jane Cleland-Huang, Horatiu Dumitru, Chuan Duan, and Carlos Castro- Herrera. Automated support for managing feature requests in open forums. *Commun. ACM*, 52(10):68–74, October 2009.

[17] Soo Ling Lim, Daniele Quercia, and Anthony Finkelstein. Stakenet: Using social networks to analyse the stakeholders of large-scale software projects. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE ’10, pages 295– 304, New York, NY, USA, 2010. ACM.

[18] Leonard Richardson and Sam Ruby. *Restful Web Services*. O’Reilly, first edition, 2007.

[19] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE Softw.*, 27(4):80–86, July 2010.

[20] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: Mining for Sample Code. *SIGPLAN Not.*, 41(10):413–430, October 2006.

[21] Suresh Thummalapenta and Tao Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE ’07, pages 204–213, New York, NY, USA, 2007. ACM.

[22] Tao Xie and Jian Pei. MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR ’06, pages 54–57, New York, NY, USA, 2006. ACM.

[23] Oliver Hummel, Werner Janjic, and Colin Atkinson. Code Conjurer: Pulling Reusable Software out of Thin Air. *IEEE Softw.*, 25(5):45–52, September 2008.

[24] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. CodeGenie: A Tool for Test-driven Source Code Search. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA ’07, pages 917–918, New York, NY, USA, 2007. ACM.

[25] Thomas A. Alspaugh, Annie I. Antón, Tiffany Barnes, and Bradford W. Mott. An integrated scenario management strategy. In *Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, RE ’99, pages 142–149, Washington, DC, USA, 1999. IEEE Computer Society.

[26] Maurits C. Blok and Jacob L. Cybulski. Reusing uml specifications in a constrained application domain. In *Proceedings of the Fifth Asia Pacific Software Engineering Conference*, APSEC ’98, page 196, Washington, DC, USA, 1998. IEEE Computer Society.

[27] Han G. Woo and William N. Robinson. Reuse of scenario specifications using an automated relational learner: A lightweight approach. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, RE ’02, pages 173–180, Washington, DC, USA, 2002. IEEE Computer Society.

[28] William N. Robinson and Han G. Woo. Finding reusable uml sequence diagrams automatically. *IEEE Softw.*, 21(5):60–67, September 2004.

[29] Wei-Jin Park and Doo-Hwan Bae. A two-stage framework for uml specification matching. *Inf. Softw. Technol.*, 53(3):230–244, March 2011.

[30] Hamza Onoruoiza Salami and Moataz Ahmed. Class diagram retrieval using genetic algorithm. In *Proceedings of the 2013 12th International Conference on Machine Learning and Applications - Volume 02*, ICMLA ’13, pages 96–101, Washington, DC, USA, 2013. IEEE Computer Society.

[31] Udo Kelter, Jürgen Wehren, and Jörg Niere. A Generic Difference Algorithm for UML Models. *Software Engineering* 64.105-116 (2005): 4-9.

[32] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *ACM SIGMOD Record*. Vol. 25. No. 2. ACM, 1996.

[33] Yuan Wang, David J. DeWitt, and Jin-Yi Cai. X-Diff: An effective change detection algorithm for XML documents. In *Proceedings of the 2003 19th International Conference on Data Engineering*. IEEE, 2003.

[34] Daniel Bildhauer, Tassilo Horn, and Jurgen Ebert. Similarity-driven software reuse. *In Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM ’09, pages 31–36, Washington, DC, USA, 2009. IEEE Computer Society.

[35] Paulo Gomes, Pedro Gandola, and Joel Cordeiro. Helping software engineers reusing uml class diagrams. In *Proceedings of the 7th International Conference on Case-Based Reasoning: Case-Based Reasoning Research and Development*, ICCBR ’07, pages 449–462, Berlin, Heidelberg, 2007. Springer-Verlag.

[36] Karina Robles, Anabel Fraga, Jorge Morato, and Juan Llorens. Towards an ontology-based retrieval of uml class diagrams. *Inf. Softw. Technol.*, 54(1):72–86, January 2012.

[37] Belén Bonilla-Morales, Sérgio Crespo, and Clifton Clunie. Reuse of use cases diagrams: An approach based on ontologies and semantic web technologies. *Int. J. Comput. Sci.*, 9(1):24–29, 2012.

[38] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’93, pages 207–216, New York, NY, USA, 1993. ACM.

[39] George A. Miller. Wordnet: A lexical database for English. *Commun. ACM*, 38(11):39–41, November 1995.

[40] Mark Finlayson. Java libraries for accessing the princeton wordnet: Comparison and evaluation. In Heili Orav, Christiane Fellbaum, and Piek Vossen, editors, *Proceedings of the Seventh Global Wordnet Conference*, pages 78–85, Tartu, Estonia, 2014.

[41] Ted Pedersen, Siddharth Patwardhan, and Jason Michelizzi. Wordnet:: similarity: Measuring the relatedness of concepts. In *Demonstration Papers at HLT-NAACL 2004*, HLT-NAACL–Demonstrations '04, pages 38–41, Stroudsburg, PA, USA, 2004. Association for Computational Linguistics.

[42] Dekang Lin. An information-theoretic definition of similarity. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pages 296–304, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[43] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[44] RESTAPPS, S-CASE Consortium, 2014.

[45] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*, pages 390–396. The MIT Press, 3rd edition, 2009.

[46] Papyrus, Eclipse UML tool, 2014, available online: http://eclipse.org/papyrus/

[47] Ulrich Nickel, Jörg Niere, and Albert Zündorf. 2000. The FUJABA environment. In *Proceedings of the 22nd international conference on Software engineering* (ICSE '00). ACM, New York, NY, USA, 742-745.