Recovering Traceability Links between Code and Textual Specification through Automated Domain Model Extraction

Jiří Vinárek¹, Petr Hnětynka¹, Viliam Šimko², and Petr Kroha¹

¹ Charles University in Prague, Faculty of Mathematics and Physics, Department of Distributed and Dependable Systems, Prague, Czech Republic
² Karlsruhe Institute of Technology, Institute for Program Structures and Data Organisation, Karlsruhe, Germany

Abstract. Requirements traceability is an extremely important aspect of software development. Maintaining efficiently traceability links between high-level requirements specification and low-level implementation is hindered by many problems. In this paper we propose a method for automated recovery of links between parts of the specification and code. The described method is based on a method allowing extraction of a prototype domain model from plain text requirements specification. The proposed method is evaluated on a real-life example.

Keywords: specification, requirements, traceability

1 Introduction

Requirements traceability is an extremely important aspect of software development [2]. Traceability itself has been defined in [8] as “the ability to describe and follow the life of requirements, in both a forwards and backwards direction”.

Maintaining efficiently traceability links between high-level requirements specification and low-level implementation is hindered by many problems (as also stated in [2]). These problems include high manual effort of making the links up-to-date, insufficient tool support, etc. Keeping the links up-to-date is hard due to the evolving implementation as well as specification (as stated in [9] requirements specification cannot be understood as final and unchangeable, especially when incremental development is applied). On the other hand, as the specification commonly serves as a bridge between developers and stakeholders without technical background, it is vital to keep them synchronized with correct traceability links. Also, it is important as the specification quite often serves as a base for decisions about software taken by the system stakeholders.

In this paper we propose a viable method for automated recovery of links between parts of the specification and code that is suitable especially for projects in a later stage of development. We do not assume that our method recovers all links, yet it may be useful as a starting point in this tedious process. Currently,
we focus only on use-case specifications written in natural language and Java implementation code.

The method proposed in this paper is based on our tool [12] which is able to extract a prototype domain model from plain text employing a statistical classifiers. We present evaluation results on a real-life example project [11].

The paper is structured as follows. Section 2 presents the method we use as a basis. It also contains the real-life project used for evaluation. In Section 3, the core method is described and it is evaluated in Section 4. Section 5 discusses related work and Section 6 concludes the paper.

2 Background

2.1 Domain Model Extraction Tool

As a basis of our traceability method, we utilize the Domain Model Extraction Tool described in [12]. The tool extracts potential domain model entities from text written in natural language (English). Input of the tool is a regular HTML document and output is an EMF\(^1\) model containing the derived domain entities linked to parts of the input text.

The tool itself runs a deep linguistic analysis on the input text and then, using a set of statistical classifiers (Maximum Entropy models), it derives the prototype domain model. The linguistic pipeline employed is based on the Stanford CoreNLP framework\(^2\). The pipeline generates linguistic features such as identified sentences, dependency trees of each sentence and coreferences, etc. Most of the linguistic features are preserved and stored in the generated EMF model. The tool already contains a default set of classification models trained on several real-life systems (a simple book library system, CoCoME). More details about the tool are available at [12].

2.2 CoCoME

To evaluate the described method, we have used the CoCoME (Common Component Modeling Example) [11] example. The goal of CoCoME was to create a common example for evaluation of component-based frameworks.

Primary reason for CoCoME usage has been the fact that it offers a real-life system with both the requirements specification (the use-cases) and implementation freely available\(^3\) (which is typically not very common).

3 Method description

Our method for recovery of traceability links is rather straightforward. As mentioned above, it is based on the Domain Model Extraction Tool. The whole

\(^1\)http://eclipse.org/emf/

\(^2\)http://nlp.stanford.edu/software/corenlp.shtml

\(^3\)http://cocome.org/
pipeline of our method is depicted in Figure 1 and described in following paragraphs.

![Method pipeline](image)

**Fig. 1. Method pipeline**

*Extraction of a domain model:* As a first step, we use the Domain Model Extraction Tool to predict domain entities out of text (an HTML document containing textual requirements specification). The tool usually predicts more entities than would be predicted by manual inspection. These false positives usually do not affect the final outcome of our method as they are filtered out in the linking phase.

*Extraction of the Implementation model:* The Implementation model of the project is reverse-engineered from the source files with the use of the MoDisco framework. MoDisco is able to obtain a model from multiple sources (Java, JSP, XML, etc.) but currently only Java code is relevant for our method.

*Linking phase:* Model linker generates a similarity matrix, where the rows represent domain entities and columns represent classes/interfaces found in the Implementation model. Each cell in the matrix contains a fractional number between 0 and 1 which represents the Jaro-Winkler string similarity measure between the corresponding entity and class/interface. Next, we further filter-out cells from the matrix that are lower than a given threshold value. The surviving entity-class pairs are taken as a result of our method (see Figure 3). Finally, as the domain model entities “remember” from which words in the input document they were generated, these words are transformed into links pointing to the particular sources files (i.e., visualizing the traceability links via one of the widely used techniques [10]). Classes from the predicted model that have no classes/interfaces from the implementation model assigned are rejected. Apart from the Jaro-Winkler distance we also tried several other string-similarity measures (Levenshtein distance, Jaro distance, Dice’s coefficient) but Jaro-Winkler gave us the best results as it gives higher score to the words with a similar prefix.

4 Evaluation

We evaluated our method on CoCoME by setting the following goals:

G1 Find the best performing threshold value for filtering the similarity matrix.

G2 Compare the results obtained by our method in a fully automated scenario against a manually prepared baseline.

**auto**: In this scenario, the domain entities were predicted by the tool in an automated fashion.

**manual**: Here, the list of entities was prepared manually, i.e. by skipping the domain model extraction phase. This scenario acts as our *first baseline*.  

**manual-ent**: Similar to the previous scenario, only the manually defined entities were filtered to avoid terms that normally do not belong to the domain model, such as actors. This scenario is our *second baseline*.

Our dataset consisted of 8 use-cases. Training set was composed of 5 use-cases with annotations added to mark involved actors. Gold set contained 7 predicted and 21 implementation classes.

*Results for G1:* To find the optimal threshold value, we executed the model linker multiple times for values in an interval (0.5, 0.95) and computed the accuracy. Results are shown in the Figure 2 as diagrams: **Precision**, **Recall**, and **F1-measure**.

*Results for G2:* The diagram denoted as **auto(F1,Recall,Precision)** in Figure 2 focuses only on the **auto** scenario and shows all the measures together. Model linker with threshold set to value 0.84 returned 13 domain entities and 38 implementation classes from which 6 predicted and 18 implementation were recognized/classified correctly. When counting predicted-implementation pairs the model linker achieved **Precision** $= 47\%$ and **Recall** $= 86\%$, thus **$F_1 = 61\%$**.

![Fig. 2. Evaluating accuracy for different threshold values (X-axis).](image-url)
### Fig. 3. Results after filtering entity-class pairs using the threshold value 0.84

<table>
<thead>
<tr>
<th>Entity from text</th>
<th>Class from code</th>
<th>Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank</td>
<td>BankImpl</td>
<td>0.90</td>
</tr>
<tr>
<td>CardReader</td>
<td>CardReader</td>
<td>1.00</td>
</tr>
<tr>
<td>CashBox</td>
<td>CashBox</td>
<td>1.00</td>
</tr>
<tr>
<td>CashBox</td>
<td>CashBoxClosedEvent</td>
<td>0.88</td>
</tr>
<tr>
<td>CashDesk</td>
<td>CashDesk</td>
<td>1.00</td>
</tr>
<tr>
<td>Change</td>
<td>ChangeAmountCalculatedEvent</td>
<td>0.94</td>
</tr>
<tr>
<td>LightDisplay</td>
<td>LightDisplayController</td>
<td>0.91</td>
</tr>
<tr>
<td>Printer</td>
<td>PrinterController</td>
<td>0.94</td>
</tr>
<tr>
<td>Printer</td>
<td>PrinterControllerEventHandler</td>
<td>0.86</td>
</tr>
<tr>
<td>Printer</td>
<td>PrinterControllerEventHandlerImpl</td>
<td>0.84</td>
</tr>
<tr>
<td>Repair</td>
<td>RepairController</td>
<td>0.88</td>
</tr>
<tr>
<td>Store</td>
<td>Store</td>
<td>0.90</td>
</tr>
<tr>
<td>Stock</td>
<td>StockItem</td>
<td>0.95</td>
</tr>
<tr>
<td>Stock</td>
<td>StockItemTO</td>
<td>0.94</td>
</tr>
<tr>
<td>Store</td>
<td>StoreImpl</td>
<td>0.91</td>
</tr>
<tr>
<td>Store</td>
<td>Store(2)</td>
<td>1.00</td>
</tr>
<tr>
<td>Store</td>
<td>StoreDesig</td>
<td>0.90</td>
</tr>
<tr>
<td>Store</td>
<td>StoreTO</td>
<td>0.94</td>
</tr>
<tr>
<td>Store</td>
<td>StoreWithEnterpriseTO</td>
<td>0.85</td>
</tr>
<tr>
<td>Trading</td>
<td>TradingEnterprise</td>
<td>0.88</td>
</tr>
</tbody>
</table>

5 Related work

Probabilistic and vector space information retrieval techniques for traceability links are explained in [1]. These approaches apply a text normalization procedures to both source code and software documents. Normalized documents are indexed and traceability links are estimated according to their similarity score. Contrary to that, our method goes in a opposite direction—it tries to synthesize a domain model from the given documents and match them with source code.

Probabilistic and vector space methods are also discussed in [4]. In addition the paper proposes best practices for writing and structuring software artefacts (documentation, specification etc.) to improve automated traceability.

A probabilistic approach to bridge the gap between high-level description of the system and its implementation is described in [3]. The paper introduces a method and its implementation as an Eclipse plugin. The cognitive assignment technique has 2 phases—the cognitive map derivation and concept assignment. In the first phase, the system processes relevant project documents (specification, bug reports, etc.) authored by an expert engineer. In the second phase, a non-expert engineer uses queries to look for relevant pieces of code. The query together with the cognitive maps are transformed into a Bayesian network; it is used to classify the source code and relevant results are returned to the user. Compared with our method this method is more suited to interactive exploration of the software project and less applicable to automatic link derivation.

An extensive survey and categorization of traceability discovery techniques can be found in [7]. The categorization defines four common types of techniques—dynamic, static, textual and other. Based on it, our method would be classified as textual as it makes use of NLP tools.
The TraceLab project\(^5\) [6] aims at providing an experimental workbench for designing, constructing, and executing traceability experiments, and facilitating the rigorous evaluation of different traceability techniques. As a future work, we plan to integrate our method to TraceLab.

Automated detection and classification of the non-functional requirements from both structured and unstructured documents is discussed in [5]. It describes a classification algorithm and evaluates its effectiveness on two datasets—requirements specification developed as a student term project and a large dataset from an industrial project. The method as well as our method uses machine-learning techniques to identify candidate entities. However, the method targets the non-functional requirements only and processes only specifications.

6 Conclusion and future work

We have presented a method for recovering traceability links between a requirements specification and implementation. The method has been evaluated on a real-life example. Results of the evaluation are promising; compared to probabilistic and vector space information retrieval model methods (e.g. as in [1]), our method performs better with respect to precision/recall ratio.

Currently, we plan to evaluate the method on several different case studies and examples to confirm performance of the method and to tune it. Obtained values may be affected by the fact that a common specification does not contain high number of domain entities. This issue would be solved with evaluation of additional data sets. Evaluation on the bigger data set would also reveal how big portion of the training data must user manually annotate to get reasonable results. Also we plan to integrate the method to the TraceLab workbench to allow for easy experimentations and evaluation.

Acknowledgments

This work was partially supported by the Grant Agency of the Czech Republic project P103/11/1489.

References


\(^5\) http://www.coest.org/index.php/tracelab/