Bilingual Software Requirements Tracing using Vector Space Model

Olcay Taner Yıldız[1], Ahmet Okutan[1,2], Ercan Solak[1]
[1]*Department of Computer Engineering, Işık University, Kumbaba Mevkii, Sile, Istanbul, Turkey*
[2]*Mobipath Erenet Software Ltd, Istanbul, Turkey*
{*olcaytaner, ercan*}*@isikun.edu.tr, ahmet.okutan@gmail.com*

Keywords: Requirements tracing, NLP

Abstract: In the software engineering world, creating and maintaining relationships between byproducts generated during the software lifecycle is crucial. A typical relation is the one that exists between an item in the requirements document and a block in the subsequent system design, i.e. class in the source code. In many software engineering projects, the requirement documentation is prepared in the language of the developers, whereas developers prefer to use the English language in the software development process. In this paper, we use the vector space model to extract traceability links between the requirements written in one language (Turkish) and the implementations of classes in another language (English). The experiments show that, by using a generic translator such as Google translate, we can obtain promising results, which can also be improved by using comment info in the source code.

1 Introduction

For large software projects, maintaining relations among artefacts produced during the software lifecycle is crucial. Once a collection of relations are established and documented, the client and the developer can consult the collection in order to understand the reasons behind design decisions or in order to verify the completeness of the software product or its components. Moreover, having such a snapshot of relations enables developers to better assess the impact of changes to one of the artefacts.

The traceability is a measure of the prevalence and the quality of these relations among software artefacts. Relations can be represented as simple links in a traceability graph whose nodes are the artefacts. Links can also be labeled to denote the nature of the relation such as, dependency, use, variant etc.

Traceability is a dynamic property of a software project. During the development and maintenance, the product evolves over time as new artefacts are generated and existing ones are modified. Therefore, maintaining traceability links within a project requires that the developers create and update the relations among the artefacts alongside the artefacts themselves.

When the artefacts lend themselves to formal analysis, generating traceability links becomes relatively easy and can often be automated with little or no human intervention, (Egyed, 2003). A typical example of this is the traceability graph between the objects in the UML diagram of system design and the collection of objects in an actual instance of the software product. Indeed, many UML tools feature bidirectional traversal of the traceability graph at this level.

However, automatically generating traceability becomes a lot more difficult when one end of a traceability link is a document written in a natural language. A typical such example is the requirements document (RD). Even if a domain language and constrained style are used in the RD, rule-based automatic analysis is made hard by the inherent informality of the natural language. On the other hand, after an appropriate formulation of the problem, some of the natural language processing (NLP) and information extraction (IR) (Baeza-Yates and Ribeiro-Neto, 1999) methods can be adapted to the traceability context. Moreover, using NLP tools, the coverage of the traceability graph can be made larger to encompass all the artefacts in the projects, from RE meeting notes to test comments, which are textual artefacts typically written in natural language.

Even if an automatic generation of traceability links yield a partially correct graph, this might be used both as a preliminary step to the manual refinement and validation. In the reverse engineering context, even having a partial traceability graph linking requirements to classes in the implementation helps the program comprehension by providing a shortcut

between the lowest and the highest layers of the product.

In this paper, we use vector space IR models to extract traceability links between the requirements written in Turkish and the implementations of classes in Java. The code uses English in all its identifiers and comments. Thus, the first challenge is the use of an automated translation tool alongside IR methods. Secondly, the agglutinative nature of the Turkish grammar requires a nontrivial morphological analysis and transformation step while generating document vectors.

The rest of the paper is organized is follows. In the next section, we review the related IR-based traceability work in the literature. In Section 3, we give the details of our proposed method. In Section 4, we describe our experimental setup and the results. The last section discusses the results and suggests directions for future research.

2 Related Work

The application of vector space IR methods to traceability were first elaborated in (Antoniol et al., 2002). In their approach, a software artefact is treated as a document vector and a metric is used to rank the proximity of pairs of artefacts. In particular, this work applies probabilistic and vector space IR models to recovering traceability links between code and its documentation. In order to be able compare the artefacts in the same document space, they pass both code and documentation through a preprocessing stage where text and code are normalized by, for example, removing all the stop words.

In (Hayes et al., 2003), vector space IR models are applied to automatically generate links among requirements. The authors use key-phrase repository and thesaurus to enrich the documents so that the metric yields non-trivial similarities between semantically close yet lexically distant documents.

Several other approaches have been used to apply IR methods to traceability link extraction. (Marcus and Maletic, 2003) uses latent semantic indexing where the dependence among the document terms are taken into account by defining an orthogonal document space and similarities are calculated by first projecting the document into this space.

(Abadi et al., 2008) and (De Lucia et al., 2009) compare and evaluate several IR-based approaches to automated traceability recovery. (De Lucia et al., 2009) also report results when the automated tools are used in conduction with manual maintenance of traceability.

Recovering the traceability link among artefacts in different natural languages poses different challenges. Clearly a translation step is needed when normalizing documents into comparable vectors. (Hayes et al., 2011) reports on the results of their work when the artefacts are English and Italian.

3 Proposed Method

Often, software artefacts share some words and traceability links could be generated using these words. Tracing links between software requirements and software modules or classes can be used in different scenarios. First it helps software developers to analyze if requirements are satisfied and which software product will be affected in case a change request is received for some requirement. Second, it can reduce the amount of time needed for a new member of the software development or support team to adapt to the project.

One distinguishing feature of our work is that our model incorporates a second language in finding the traceability links between requirements and source code. Most of the time the RD is prepared in the language of the clients and users, as this is the natural implication of RD process. On the other hand, software developers tend to use English in the software development (coding) process. To incorporate the traceability across bilingual artefacts, a translation step is needed.

In our work, we implemented a software pipeline which accepts the RD and source code in different languages. We have the RD in some local (second) language (Turkish in our case) and the source code packages in English. We then use our software pipeline to generate the traceability links between the requirements and source code classes.

3.1 Preprocessing of Software artefacts

First, we break down the RD into distinct requirements sets. To trace the links between each set and source code (classes), we use five different term generation strategies. For each case except the last one, English terms were converted to the second language using Google translate. The cases are the following:

1. Plain: To generate the set of terms for software classes, package, class, method and attribute names of the classes are used.
2. NoAccessors: The terms in the Plain case are analyzed and the getter and setter methods of the classes are excluded. The intuition is that the

terms like 'get' or 'set' are not good keywords in distinguishing relevant classes. Moreover, keywords in the accessors (like the term 'Name' in a method named 'getName') are already included in the attribute names of the classes.

3. Comments: In addition to the terms in the Plain case, the terms in the software comments are considered.
4. TranslateAll: When translating the attributes, function names, and class names from English using Google Translate, we include all possible translations, not just the best candidate. The intuition is that other translations might include a semantically close terms that match.
5. AllEnglish: The original set of 15 requirements were translated to English but no translation is applied on the source code side. In order to extract the terms for the source code classes, like the Plain case, the package, class, method and attribute names are included.

We apply the following preprocessing steps to generate the set of terms for each software artefact.

- We prune all special characters like spaces, operators or punctuation marks.
- We split words which are composed of one or more sub words. For instance a word 'MapViewer' is separated to two words as 'Map' and 'Viewer'.
- We apply a morphological analysis and clean and stem all terms (For instance the term 'maps' becomes 'map', 'viewer' becomes 'view').

3.2 Vector Space Model

We use Vector Space Model (VSM) to extract traceability links between software requirements and source code. We represent the software artefacts as vectors of terms where the set of terms are generated after preprocessing steps defined above. We generate a term by document matrix of size $m \times n$ where m represents the number of generated terms and n represents the number of software artefacts. Each $e(i,j)$ entry of the term by document matrix (for the requirements or the source code classes) is a measure of the importance of term t_i for artefact a_j. In this work we calculate $e(i,j)$ by the term frequency-inverse document frequency (tf-idf) (Baeza-Yates and Ribeiro-Neto, 1999).

Assume t is a term, d is a document and D is the document space, then tf-idf is calculated as:

$$\text{tf-idf}(t,d,D) = \text{tf}(t,d) \times \text{idf}(t,D) \qquad (1)$$

where $tf(t,d)$ or simply tf shows the number of times that the term t occurs in document d. On the other hand, the inverse document frequency $idf(t,D)$ shows whether the term t is common or rare across the document space D. We calculate $idf(t,D)$ by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of the quotient.

$$\text{idf}(t,D) = \log \frac{|D|}{|\{d \in D : t \in d\}|} \qquad (2)$$

where $|D|$ shows the total number of documents in the corpus and $|\{d \in D : t \in d\}|$ shows number of documents where the term t appears (i.e., $\text{tf}(t,d) \neq 0$). Using tf-idf, we decrease the weight of terms that are observed very frequently in the software artefacts and increase the weight of terms that are observed rarely.

Cosine similarity is used to measure the similarity of two vectors. If a source code class vector is represented as c and a requirement vector is represented as r, then the cosine similarity of these vectors is calculated as:

$$\text{cosine similarity}(c,r) = \frac{\langle c,r \rangle}{||c|| \cdot ||r||} \qquad (3)$$

Assuming we have k classes and l requirements, we use the cosine similarity to calculate the similarity of the requirement and software class vectors and generate a matrix $\mathbf{M}$ of size $k \times l$, where the software classes are used as rows and the requirements are used as columns.

$$\mathbf{M} = \begin{bmatrix} e(\mathbf{c}_1,\mathbf{r}_1) & e(\mathbf{c}_1,\mathbf{r}_2) & \ldots & e(\mathbf{c}_1,\mathbf{r}_l) \\ e(\mathbf{c}_2,\mathbf{r}_1) & e(\mathbf{c}_2,\mathbf{r}_2) & \ldots & e(\mathbf{c}_2,\mathbf{r}_l) \\ \ldots & \ldots & \ldots & \ldots \\ e(\mathbf{c}_k,\mathbf{r}_1) & e(\mathbf{c}_k,\mathbf{r}_2) & \ldots & e(\mathbf{c}_k,\mathbf{r}_l) \end{bmatrix} \qquad (4)$$

Each entry $e(i,j)$ of this matrix shows how much a class c_i is related with a requirement r_j and its value ranges from 0 to 1.

In order to evaluate the performance of the model, we also have a software developer in the project to generate a reference matrix $\mathbf{M_r}$ that shows the actual (in his/her expert opinion) relations among artefacts. $\mathbf{M_r}$ is a binary matrix that shows the existence or the absence of relations. We filter the relevance scores (between 0 and 1) in $\mathbf{M}$ with different thresholds λ and compare the resulting matrix with $\mathbf{M_r}$.

3.3 Comparison Metrics

During the traceability analysis, we predict if a requirement r_i is relevant for a software class c_j. Depending on the outcome of our prediction, there are four possible cases:

- True positive (TP): We find a relation where there is one.
- False negative (FN): We don't find a relation where there is one.
- True negative (TN): We don't find a relation where there is none.
- False positive (FP): We find a relation where there is none.

The 2×2 confusion matrix is calculated using the counts of the cases above and is shown in Table 1.

Table 1: 2×2 confusion matrix

	Prediction	
Truth	+	-
+	TP	FN
-	FP	TN

It is always desirable to be able to find as many relationships as possible while avoiding false predictions FN and FP. Precision, Recall, and F-measure are based on the confusion matrix shown in Table 1 and are calculated as:

$$\text{Precision} = \frac{TP}{TP+FP}$$

$$\text{Recall} = \frac{TP}{TP+FN}$$

$$\text{F-measure} = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Merely, F-measure is the harmonic mean of precision and recall.

4 Experiment Results

We use a Java based Internet vehicle tracking project called Mobitrack which was developed by a private company and supported by Technology Development Foundation of Turkey. One project manager, an analyst, three developers and a tester worked in the project. All the documentation and source code packages of the project are shared with us during our experiments. Since the software requests are received from local customers, the RD of the project was written in Turkish. On the other hand, to avoid encoding problems, English was used in all class, method, variable names, and even software comments. The requirements in the RD are divided into 15 separate artefacts and the number of classes in the source code is 140. There are approximately 300 traceability links between the artefacts and classes.

We compare the following approaches, namely Plain, NoAccessors, Comments, TranslateAll, and AllEnglish using the vector space model. In order to compare these five approaches, we use precision-recall and F-measure metrics for different thresholds (λ). In the precision-recall curve, each point corresponds to a precision-recall pair for a specific λ, whereas in F-measure curve, we plot F-measure values for increasing levels of λ.

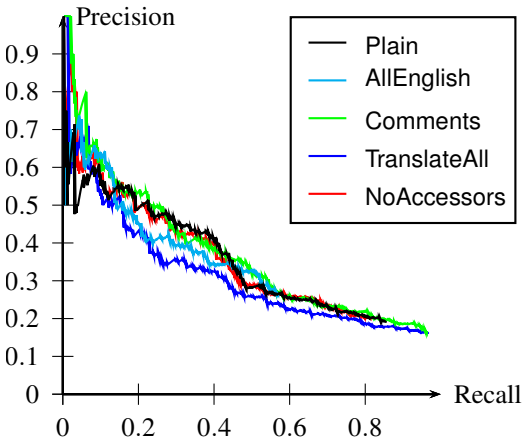

Figure 1: Precision-recall curve of five approaches.

Figure 1 shows the precision-recall curve of these five approaches. We see from the results that, in general, the performances of Plain, Comments, and NoAccessors are similar to each other, and they are also better than both AllEnglish and TranslateAll. On the other hand, for small recall values, AllEnglish has the highest precision.

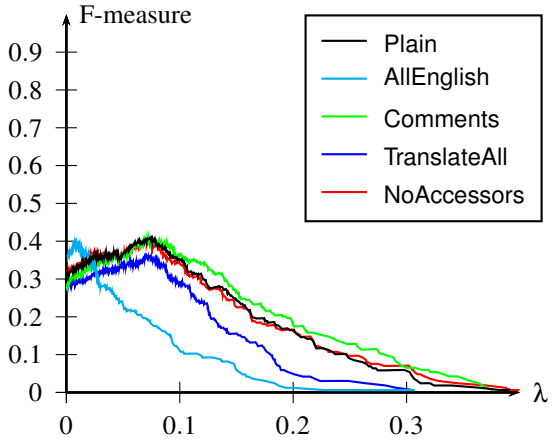

Figure 2: F-measure curve of five approaches

Figure 2 shows the F-measure curve of these five approaches. We see from the results that, AllEnglish

and other four methods have peaks for different thresholds. AllEnglish has the best performance for smaller thresholds, whereas all bilingual approaches (class info in English, requirements in Turkish) have the best performance for larger thresholds. In general, Comments performs better than NoAccessors and Plain which also perform better than TranslateAll in terms of F-measure.

Table 2: Number of shared words in artefacts (classes) and requirements

Approach	Shared Word Count
AllEnglish	1146
NoAccessors	3457
Plain	4237
Comments	13471
TranslateAll	19459

Although TranslateAll increases the number of words in the class artefacts significantly, it also spoils the class similarity with irrelevant words. In other words, TranslateAll not only increases the similarity between the classes and their associated requirements, but also other requirements which are not associated with those classes (see Table 2).

5 Threats To Validity

There are three types of validity threats to consider in empirical studies i.e. construct, internal and external validity (Perry et al., 2000). We feel more safe in terms of the construct validity threats, since our experiment steps are so clear and straight forward. An internal validity threat is observed if causal relationships between the dependent and independent variables are not established properly. In cases where naming conventions are not followed during source code development, it could be more difficult to find the traceability links among software artefacts. Furthermore, we think that it is difficult to generalize our findings, and further research with more data sets are needed to justify our observations.

6 Conclusions and Future Work

The last decade saw a significant amount of application of artificial intelligence techniques to standard computer engineering processes. For example, standard natural language processing techniques are applied more and more on software engineering processes. In this paper, we are interested in the problem of automatically determining traceability links between RD and the source code. Although there are important works on this subject in the literature, bilingual approaches, that is, the documents are in one language and the source code in another language, are not covered well.

In this work, we propose an approach that accepts the RD and source code in different languages and generates the traceability links between the requirements and the implementation. The proposed approach translates (with Google translate) requirements/classes into the second/first language and uses the vector space model to calculate the similarities between requirements and classes. With appropriately selected thresholds, the proposed approach can reach encouraging F-measure levels.

As future work, we are planning to incorporate more pairs of languages into our approach and investigate the impact of the language types on the performances. Clustering requirements into groups before similarity calculation may also improve the performance of our approach and therefore can be another future direction.

REFERENCES

Abadi, A., Nisenson, M., and Simionovici, Y. (2008). A traceability technique for specifications. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 103–112, Washington, DC, USA. IEEE Computer Society.

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983.

Baeza-Yates, R. A. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

De Lucia, A., Oliveto, R., and Tortora, G. (2009). Assessing IR-based traceability recovery tools through controlled experiments. *Empirical Software Engineering*, 14(1):57–92.

Egyed, A. (2003). A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2):116–132.

Hayes, J. H., Dekhtyar, A., and Osborne, J. (2003). Improving requirements tracing via information retrieval. In *Proceedings of the 11th IEEE International Conference on Requirements Engineer-*

ing, RE '03, pages 138–, Washington, DC, USA. IEEE Computer Society.

Hayes, J. H., Sultanov, H., Kong, W.-K., and Li, W. (2011). Software verification and validation research laboratory (svvrl) of the university of kentucky: Traceability challenge 2011: Language translation. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '11, pages 50–53, New York, NY, USA. ACM.

Marcus, A. and Maletic, J. I. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 125–135, Washington, DC, USA. IEEE Computer Society.

Perry, D. E., Porter, A. A., and Votta, L. G. (2000). Empirical studies of software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 345–355, New York, NY, USA. ACM.