Enhancing an Artefact Management System with Traceability Recovery Features

Andrea De Lucia, Fausto Fasano, Rocco Oliveto, Genoveffa Tortora
adelucia@unisa.it, ffasano@unisa.it, r.oliveto@tiscali.it, tortora@unisa.it

Dipartimento di Matematica e Informatica – Università di Salerno
Via Ponte don Melillo – 84084 Fisciano (SA) – Italy

Abstract

We present a traceability recovery method and tool based on Latent Semantic Indexing (LSI) in the context of an artefact management system. The tool highlights the candidate links not identified yet by the software engineer and the links identified but missed by the tool, probably due to inconsistencies in the usage of domain terms in the traced software artefacts.
We also present a case study of using the traceability recovery tool on software artefacts belonging to different categories of documents, including requirement, design, and testing documents, as well as code components.

1. Introduction

Software artefact traceability is widely recognized as an important factor for effectively managing the development and evolution of software systems. Traceability is also fundamental to help in program comprehension, maintenance, impact analysis, and reuse of existing software. However, despite of its importance, the support for traceability in contemporary software engineering environments and tools is not satisfactory. This inadequate traceability is one of the main factors that contributes to project over-runs and failures [15], [21].
Several research and commercial tools are available that support traceability between artefacts [7], [9], [10], [19], [20], [24], [25], [26], [28], [29]. However, the main drawback of these tools is the lack of automatic or semi-automatic traceability link generation and maintenance. This results in the need for a costly activity of manual detection and maintenance of the traceability links, that may have to be done frequently due to the iterative nature of software development [10]. Even when semi-automatic support is provided, this task is time consuming, error prone, and person-power intensive [12], [18].
Recently, researchers have addressed the problem of traceability link recovery between requirements [12], [18], between code and documentation [2], [22] and between maintenance requests and software documents [3]. These methods are based on the observation that most of the software documentation is text based or contains text descriptions. For this reason, they apply Information Retrieval (IR) techniques [13], [17] to the traceability link recovery problem.
The aim of our work is to support the software engineer in the identification of the traceability links between software artefacts within an artefact management system. Indeed, the problem of maintaining traceability links involves all the artefacts produced during a software development process [7], [10], [14], [28]. This is especially true for evolutionary processes, where artefacts can be added, updated, or deleted in each phase, thus requiring a continuous reorganization of the traceability links. In [14] we have presented ADAMS (ADvanced Artefact Management System) an artefact-based process support system for the management of human resources, projects, and software artefacts. In particular, ADAMS provides support for traceability and event-based notification of changes, thus increasing the context awareness during the evolution of software artefacts. In the current implementation of ADAMS, the software engineer is in charge of maintaining traceability links between software artefacts.
In this paper we show how a traceability link recovery tool based on the IR technique Latent Semantic Indexing (LSI) [13], [16] can be integrated in ADAMS. For a given artefact the tool identifies the list of artefacts whose similarity is greater than or equal to a given threshold (*candidate links*) and compares them with the list of artefacts traced by the software engineer. In this way, the tool is able to show the list of candidate links the software engineer has not identified yet and the list of links identified by the software engineer and missed by the tool (*warning links*). The latter links may indicate inconsistencies in the usage of domain terms in the traced software artefacts. We propose a process to iteratively identify these two types of links by suitably tuning the similarity threshold.
We also present a case study of using the traceability recovery tool on software artefacts belonging to requirement and design documents, as well as testing artefacts and code components. As further contribution of this paper we show how better results can be achieved using a variable similarity threshold, rather than a constant threshold, and splitting the artefacts in the repository in different categories.
The remaining of the paper is organized as follows. Section 2 discusses related work in the fields of traceability support tools and traceability recovery. Section 3 and 4

Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)
1063-6773/04 $20.00 © 2004 **IEEE**

present the artefact management system ADAMS and the traceability recovery method developed to enhance traceability management, respectively. Section 5 discusses the results of the case study, while Section 6 concludes.

2. Related Work

The subject of this paper covers two areas of interest: artefact traceability support tools and traceability recovery. Each of them will be addressed below.

2.1. Artefact Traceability Support Tools

Several research and commercial tools are available that support traceability between artefacts: DOORS [29], IBIS [20], TOOR [24], REMAP [25], Rational RequisitePro [26], and RDD.100 [19] are only a few examples. These tools provide effective support for recording, displaying, and checking the completeness of installed traces using operations such as drag and drop [29], or by clicking on a cell of a traceability link matrix [26].

Recently artefact traceability has been tackled within the Ophelia project [28] which aims at developing a platform supporting software engineering in a distributed environment. In Ophelia the artefacts of the software engineering process are represented by CORBA objects. A graph is created to maintain relationships among these elements and can be used to navigate between them.

OSCAR [7] is the artefact management subsystem of the GENESIS environment [5]. It has been designed to non-invasively interoperate with workflow management systems, development tools, and existing repository systems. Artefacts in OSCAR have a type hierarchy, similar to the object-oriented style. Every artefact possesses a collection of standard meta-data and is represented by an XML document containing both meta-data and artefact data.

Some tools [9], [10] also combine the traceability layer with event-based notifications to make users aware of artefact modifications. For example, Chen and Chou [9] have proposed a method for consistency management in the Aper process environment. The method is based on maintaining traceability relations between artefacts and using triggers to identify artefacts affected by changes to a related artefact. Cleland-Huang *et al.* [10] have developed EBT (Event Based Traceability), an approach based on a publish-subscribe mechanism between artefacts. When a change occurs on a given artefact having the publish role, notifications are sent to all the subscriber (dependent) artefacts.

The tools described in this section have a drawback: the need for a manual detection and maintenance of the traceability links while the system changes and evolves, even when a semi-automatic support tool is adopted, is a time consuming, error prone, and person-power intensive task [10], [12], [18]. In general, such tools require the user to assign keywords to all the documents prior to tracing and to perform interactive searches for potential linking requirements or design elements [1]. In addition, most of them do not provide support for easily retracing new versions of documents. As a result, they return many potential or candidate links that are not correct and fail to return correct links.

In [14] we have presented the artefact management system ADAMS. In particular, ADAMS provides support for artefact versioning, as most of the tools presented in this section, and hierarchical composition of artefacts, like OSCAR [7]. Like Aper [9] and EBT [10], ADAMS supports traceability in an active way, by propagating events through the traceability layer whenever new artefacts are added, updated, or deleted. In this paper we show how to improve the artefact traceability in ADAMS by providing the software engineer with information retrieval based traceability recovery features.

2.2. Artefact Traceability Recovery

Several traceability recovery methods have been proposed in the literature. Some of them deal with recovering traceability links between design and implementation. Murphy *et al.* [23] exploit software reflexion models to match a design expressed in the Booch notation against its C++ implementation. Regular expressions are used to exploit naming conventions and map source code model entities onto high-level model entities. Similarly, Antoniol *et al.* [4] present a method to trace C++ classes to an OO design. Both the source code classes and the OO design are translated into an Abstract Object Language (AOL) intermediate representation and compared using a maximum match algorithm. Weidl and Gall [31] followed the idea of adopting a more tolerant string matching, where procedural applications are rearchitectured into OO systems.

The approach adopted in [30] is based on guidelines for changing requirements and design documents based on a conceptual trace model. A semi-automatic recovery support is provided by using name tracing. In [6] consistency rules between UML diagrams, automated change identification and classification between two versions of a UML model, as well as impact analysis rules have been formally defined by means of OCL constraints on an adaptation of the UML meta-model. Sefika *et al.* [27] have developed a hybrid approach that integrates logic-based static and dynamic visualization and helps determining design-implementation congruence at various levels of abstraction.

Other approaches consider text documents written in natural language, such as requirements documents. Zisman *et al.* [32] automate the generation of traceability relations between textual requirement artefacts and object models using heuristic rules. These rules match syntactically related terms in the textual parts of the requirements artefacts with related elements in an object model (e.g.

classes, attributes, operations) and create traceability relations of different types when a match is found.

Recently, several authors have applied Information Retrieval (IR) methods [17] to the problem of recovering traceability links between requirements [12], [18], between code and documentation [2], [22] and between maintenance requests and software documents [3].

Dag *et al.* [12] perform automated similarity analysis of textual requirements using IR techniques. They propose to continuously analyze the flow of incoming requirements to increase the efficiency of the requirements engineering process. Huffman Hayes *et al.* [18] use different information retrieval algorithms based on the vector space model [17] to improve traceability recovery between requirements.

Antoniol *et al.* [2] use information retrieval methods based on probabilistic and vector space models [17]. They applies the two methods on two case studies to trace C++ source code onto manual pages and Java code to functional requirements, respectively. In a different paper [3] the authors also use the vector space model to trace maintenance requests on software documents impacted by them. Marcus and Maletic [22] performed the same case studies as in [2], but used a different IR method, namely Latent Semantic Indexing (LSI) [13]. The advantage of LSI with respect to classical probabilistic or vector space models is that it does not require a preliminary morphological analysis (stemming) of the document words. This allows the method to be applied without large amounts of pre-processing, which drastically reduces the costs of traceability link recovery. In addition, using a method that avoids stemming is particularly useful for languages, such as Italian, that presents a complex grammar, verbs with many conjugated variants, words with different meanings in different contexts, and irregular forms for plurals, adverbs, and adjectives [2].

For the reason above, we also use LSI for traceability link recovery. However, we aim at integrating the traceability recovery tool in an artefact management system and therefore we deal with any type of software artefacts, including requirement and design artefacts, test case specifications, and source code modules.

3. ADAMS

ADAMS (ADvanced Artefact Management System) is an artefact-based Process Support System (PSS). It enables the definition of a process in terms of the artefacts to be produced and the relations among them, supporting a more agile software process management than activity-based PSSs, in particular concerning the deviations from the process model [14].

ADAMS poses a greater emphasis to the artefact life cycle by associating software engineers to the different operations that can be performed on an artefact. The support for cooperation is offered by ADAMS through typical features of a configuration management system. ADAMS enables groups of people to work on the same artefact, depending on the required roles. Software engineers can cooperate according to a lock-based policy or concurrently, if branch versions of artefacts are allowed.

The system has been enriched with features to deal with some of the most common problems faced by cooperative environments, in particular context awareness and communication among software engineers. A first context-awareness level is given by the possibility to see at any time the people who are working on an artefact. Context awareness is also supported through event notifications: software engineers working on an artefact are notified when another branch is created by another worker. This provides a solution to the isolation problem for resources working on the same artefact in different workspaces: in fact context awareness allows to identify possible conflicts before they occur, since the system is able to notify interested resources as soon as an artefact is checked-out and potentially before substantial modifications have been applied to it.

Software engineers can subscribe other events they would like to be notified about. Events mainly concern the operations performed on artefacts and projects. For example, an event could be the modification of the status of an artefact or the creation of a newer version for it. A number of events are automatically notified without any need for subscription. Examples include notifying a software engineer he/she has been allocated to a project or an artefact.

ADAMS enables software engineers to create and store traceability links between artefacts in terms of dependences. A dependence consists of a relation between two artefacts, together with some additional information to specify the type of dependence, as starting conditions, production constraints and output rules. Besides being useful for impact analysis during software evolution, traceability links in ADAMS are also useful to manage the software process and notify software engineers that the production of a given artefact can start, or that an artefact has to be changed, because of some changes in the artefacts it depends on: indeed, events concerning the production of new versions of an artefact are propagated through the traceability layer of ADAMS to the artefacts depending directly or indirectly on it (and consequently to the software engineers responsible for them).

Besides being used for event propagation, the traceability links can be visualized by a software engineer (see Figure 1) and browsed to look at the state of previously developed artefacts, to download latest artefact versions, or to subscribe events on them and receive notifications concerning their development. Moreover, software engineers that make use of previously developed artefacts to produce new artefacts may send feedbacks to the former artefacts if problems are discovered (see Figure 1). Feedbacks are then notified to artefact mangers to make decisions about.

Feedbacks and event-based traceability are the two mechanisms used by ADAMS to support process management. This approach is much more flexible than activity-based workflow management systems [5], in particular with respect to the deviations from the process model.

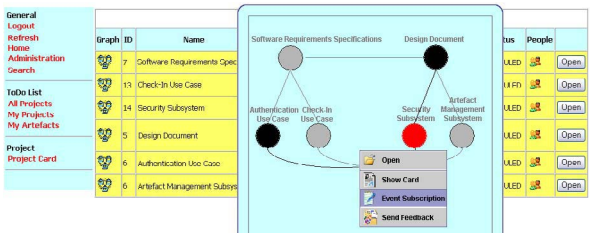

Figure 1. Traceability in ADAMS

4. Traceability recovery in ADAMS

Even if ADAMS provides an intuitive interface to specify traceability links, in the first release the charge of their identification was delegated to the software engineer. The latter also has the responsibility to manage traceability links whenever new artefacts are added to the project, existing artefacts are removed or new versions are checked-in. As the project grows-up, this task tends to be hard to manage, so automatic or semi-automatic tools are needed. We have developed a traceability link recovery tool based on an IR technique, namely LSI [13]. In particular, the tool proposes missing links and highlights probably wrong or obsolete ones.

In Section 2 we have overviewed several papers discussing the benefits of using IR techniques to help during traceability link recovery. All these studies have shown that these tools cannot completely replace the work of the software engineer, as the number of missed links and links incorrectly retrieved by IR tools is not insignificant.

For a given document d_i, an IR based traceability link recovery compares the document d_i (used as a query) against the other documents in the document space and ranks these documents according to their similarity with d_i. Moreover, these tools use some (fixed or variable) threshold to present the software engineer only the subset $retrieved_i$ of top documents in the ranked list that are deemed similar to d_i. The set of retrieved documents does not in general coincide with the set $correct_i$ of documents in the document space that are in fact similar to d_i. Indeed, the tool will fail to retrieve some of the correct documents, while on the other hand it will also retrieve documents that are not correct.

In general, the performances of IR tools and more specifically of traceability link recovery tools are measured using two IR metrics, namely, *recall* and *precision*:

$$recall_i = \frac{|correct_i \cap retrieved_i|}{|correct_i|} \quad precision_i = \frac{|correct_i \cap retrieved_i|}{|retrieved_i|}$$

Both measures will have values between [0, 1]. If the recall is 1, it means that all correct links were recovered, though there could be recovered links that are not correct. If the precision is 1, it means that all recovered links were correct, though there could be correct links that were not recovered. In general, retrieving a lower number of documents for each query would result in higher precision, while a higher number of retrieved documents would increase the recall. The values of recall and precision depends on the threshold used to cut the ranked list: in general, the higher the threshold the lower the recall (the higher the precision) and vice versa.

4.1. Traceability Recovery Process

All the studies on traceability recovery discussed in Section 2 aim at maximizing the number of recovered correct traceability links (i.e., the recall). As a consequence, the threshold is kept low, thus resulting in the recovery of many incorrect links.

Aware of the unfeasibility to completely substitute the software engineer in the task of maintaining traceability links during software evolution, in ADAMS we aim at providing a support during this process. Accordingly, we expect that for a given artefact d_i, the software engineer provides the tool with an initial (possibly empty) set of manually traced links. In the following, we denote with $link_i$ the set of artefacts traced by the software engineer on d_i and with $\overline{link_i}$ the complementary set of $link_i$, i.e., the set of all artefacts in the repository except the ones in $link_i$. As said before, given an artefact d_i, and a similarity threshold ε, the tool will return the set $retrieved_i(\varepsilon)$, i.e., the set of artefacts whose similarity to d_i is greater than or equals to ε. In the following we denote with $\overline{retrieved_i(\varepsilon)}$ the complementary set of $retrieved_i(\varepsilon)$.

The intersection of these four sets results in other four sets of interest:

- $TracingAgreement_i(\varepsilon) = link_i \cap retrieved_i(\varepsilon)$
- $NonTracingAgreement_i(\varepsilon) = \overline{link_i} \cap \overline{retrieved_i(\varepsilon)}$
- $LostLinks_i(\varepsilon) = \overline{link_i} \cap retrieved_i(\varepsilon)$
- $WarningLinks_i(\varepsilon) = link_i \cap \overline{retrieved_i(\varepsilon)}$

The first two sets contain the artefacts retrieved and excluded, respectively, by both the software engineer and the tool. The set $LostLinks_i(\varepsilon)$ contains the artefacts not traced by the software engineer and retrieved by the tool, while $WarningLinks_i(\varepsilon)$ contains the artefacts traced by the software engineer and missed by the tool.

The final goal of the traceability recovery process should be to maximize the first two sets and consequently minimize the other two sets, i.e., to have a complete agreement between the software engineer and the tool. It is worth noting that in case the final set of links traced by the software engineer coincides with the set of correct links, a

complete agreement means achieving 100% of precision and recall for the traceability recovery tool. Therefore, this agreement is in general impossible to achieve, because of the limitations of both the humans developing artefacts and the IR tool. For these reasons, we pursue an incremental process aiming at reaching such an agreement.

Given the initial set of traceability links defined by the software engineer, an artefact d_i added or updated in ADAMS, and a selected threshold ε, the traceability recovery tool will provide the software engineer with the sets on which they disagree: $LostLinks_i(\varepsilon)$ and $WarningLinks_i(\varepsilon)$. In particular, the software engineer has to investigate the lost links to discover new traceability links, thus enriching the set $link_i$. On the other hand, the warning links have to be investigated for two reasons: in case the tool is actually right the traceability link has to be removed, while in case the software engineer is right, the indications of the tool might reveal some inconsistencies in the usage of terms within the traced artefacts.

Figure 2 shows the way artefacts in the repository can move from a set to another, depending on two different operations made by a software engineer on artefact d_i, namely changes to d_i and inserting/deleting traceability links between d_i and other artefacts. In particular, the result of inserting a traceability link between d_i and an artefact d_j consists of moving d_j from $LostLinks_i(\varepsilon)$ to $TracingAgreement_i(\varepsilon)$ or from $NonTracingAgreement_i(\varepsilon)$ to $WarningLinks_i(\varepsilon)$, respectively. On the other hand, removing a traceability link between d_i and d_j results in the opposite moves. As a result of a change to the artefact d_i the consistency of term usage with respect to the artefact d_j can increase or decrease. Increasing the consistency result in moving d_j from $WarningLinks_i(\varepsilon)$ to $TracingAgreement_i(\varepsilon)$ or from $NonTracingAgreement_i(\varepsilon)$ to $LostLinks_i(\varepsilon)$, respectively, in case the similarity of the two documents becomes greater than or equal to the threshold ε. Decreasing the consistency can result in the opposite moves.

It is worth noting that the number of warning links increases with the chosen similarity threshold and it is limited by the number of links traced by the software engineer, so in general for a given artefact this is not a very large number. On the other hand, the number of links missed by the software engineer with respect to the IR tool increases when the threshold decreases and it is limited by the number of artefacts in the repository; this means that this set can be very large and contain a very high number of false positives when the threshold is low. For this reason, we propose to adopt an incremental traceability link recovery process, that starts with a high threshold. In this way, the set of warning links can be initially ignored, while the set of missed links will contain very few false positives. At each iteration, the software engineer will classify the missed links either as actual links (thus including them in $link_i$) or as false positives. In both cases the tool will not propose them in the next iterations, when a lower threshold is used. When the threshold is enough low the software engineer should consider to analyze the set of warning links, because this might be an indication of a possible inconsistence. Another consideration to make is the fact that the choice of the initial threshold greatly depends on how accurately the software engineer has initially defined the set $link_i$: if $link_i$ is empty (no definition at all of traceability links), then it is advisable to have a finer grained control on the process and use a very high initial threshold and small decrements of it in the following iterations. If the software engineer was very accurate in manually defining traceability links then he/she can use a lower initial threshold and larger decrements of it, that results in fewer iterations and a more coarse grained control over the process.

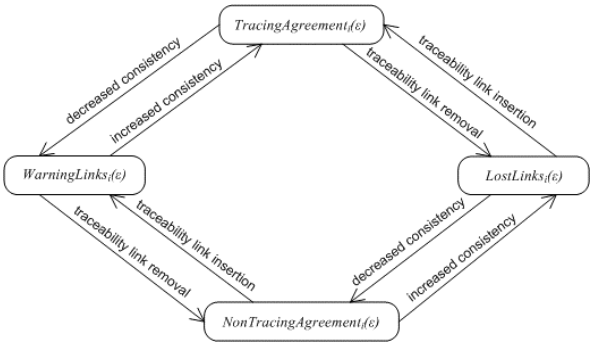

Figure 2. Traceability sets transitions

4.2. Latent Semantic Indexing

The IR technique used in ADAMS for traceability link recovery is Latent Semantic Indexing (LSI) [13]. LSI is an important extension of the Vector Space Model, in which the dependencies between terms and between documents, in addition to the associations between terms and documents, are explicitly taken into account. This is done by simultaneously modeling all the associations of terms and documents. LSI assumes that there is some underlying or "latent structure" in word usage that is partially obscured by variability in word choice, and use statistical techniques to estimate this latent structure. A description of terms, documents and user queries based on the underlying, "latent semantic", structure is used for representing and retrieving information. In this way LSI partially overcomes some of the deficiencies of assuming independence of words, and provides a way of dealing with synonymy automatically without the need for a manually constructed thesaurus.

The heart of LSI is Singular Value Decomposition (SVD), a technique closely related to eigenvector decomposition and factor analysis [11]. This technique is used to derive a particular latent semantic structure model from the term-by-document matrix. Any rectangular matrix, for example

a *txd* matrix of terms and documents, X, can be decomposed into the product of three other matrices:

$$X = T_0 \cdot S_0 \cdot D_0$$

such that T_0 and D_0 have orthonormal columns and S_0 is diagonal. This is called *singular value decomposition of X*. T_0 and D_0 are the matrices of *left* and *right singular* vectors called *terms matrix* and *documents matrix* respectively and S_0 is the diagonal matrix of *singular values* called *concepts matrix*.

SVD allows a simple strategy for optimal approximate fit using smaller matrices. If the singular values in S_0 are ordered by size, the first k largest values may be kept and the remaining smaller ones set to zero. The product of the resulting matrices is a matrix X' which is only approximately equal to X, and is of rank k. Since zeros were introduced into S_0, the representation can be simplified by deleting the zero rows and columns of S_0 to obtain a new diagonal matrix S, and deleting the corresponding columns of T_0 and D_0 to obtain T and D respectively. The result is a reduced model:

$$X \approx X' = T \cdot S \cdot D$$

which is the rank-k model with the best possible least square fit to X.

The choice of k is critical: ideally, we want a value of k that is large enough to fit all the real structure in the data, but small enough so that we do not also fit the sampling error or unimportant details. The proper way to make such choice is an open issue in the factor analytic literature. In our experiments, we used an operation criterion, i.e. a value of k which yields good retrieval performances. In particular we chose a k that have value between the 10% to 50% of the dimension of the artefact space.

One can also interpret the analysis performed by SVD geometrically. The result of SVD is a vector representing the location of each term and document in the k-dimensional LSI representation. The location of a term vector reflects the correlation in term usage across the document. In this space the cosine between vectors corresponds to their estimated similarity. Since both term and document vectors are represented in the same space, similarities between any combination of terms and documents can be easily obtained. Retrieval is then performed using the database of singular values and vectors obtained from the truncated SVD. The terms in a query are used to identify a point in this space, and all documents are then ranked by their similarity to the query. The similarity between a pair of artefacts is computed as the cosine of the angle between the corresponding vectors.

LSI does not use a predefined vocabulary, or a predefined grammar, therefore no morphological analysis or transformation is required. However some text transformations are needed to prepare the source code and documentation to form the corpus of LSI. First, the white spaces in the text are normalized and most non-textual tokens from the text are eliminated (i.e. operators, special symbol, etc). Then the identifiers in the source code are split into the compounding names based on well-know coding standards. All these operations are automatically applied to documents. It is worth nothing that the bulk of LSI processing time is anyway spent in computing the truncated SVD of the large sparse term-by-document matrix.

5. Case study

We have experimented the LSI based traceability link recovery method on software artefacts produced during different phases of an on-going development project conducted by final year students at the University of Salerno, Italy. The project aims at developing a software system implementing all the operations required to manage a medical ambulatory. Table 1 shows the type and the number of artefacts analyzed.

Artefact type	# artefacts
use cases	30
interaction diagrams	20
test cases	63
code classes	37
Total number	150

Table 1. Analyzed artefacts

5.1. Method used to assess the results

In section 4 we have mentioned some metrics used for assessing the results of traceability link recovery tools, called *precision* and *recall*. These metrics have been defined for a single artefact used as a query, while to assess the performances of the tool on the entire system we use the following aggregate metrics:

$$recall = \frac{\sum_i |correct_i \cap retrieved_i|}{\sum_i |correct_i|} \qquad precision = \frac{\sum_i |correct_i \cap retrieved_i|}{\sum_i |retrieved_i|}$$

Given a similarity measure between two artefacts, establishing if these have to be considered similar can be based on different approaches. A first method consists of imposing a threshold on the number of recovered links (*cut point*) [2], [22], regardless of the actual value of the similarity measure. In this way, we select the top μ ranked artefacts for each query, where $\mu \in \{1, 2, \ldots, n\}$. An extension of this method consists of specifying the percentage of the documents of the ranked list that can be considered similar to the query (*cut percentage*). In this way the cut point depends on the size of the ranked list.

A different approach consists of using a threshold ε on the cosine similarity measure. Among all the pairs of artefacts, only those having a similarity measure greater than or equal to ε will be retrieved. We have also compared other three methods to compute the cosine thresholds:

1. *Constant threshold*: this is the standard method used in the literature [22]. The cosine threshold is constant and has values between [-1, 1]; a good and widely used threshold is ε = 0.7, that corresponds to a 45º angle between the corresponding vectors.
2. *Variable threshold*: this is an extension of the previous approach. The constant threshold is projected onto a particular interval, where the lower bound is the minimum similarity measure (instead of -1) and the upper bound is the maximum similarity measure (instead of +1). The variable threshold has values between 0% and 100% and on the basis of this value this method determines a cosine threshold that has values between [min similarity, max similarity]. This approach has not been used in previous researches.
3. *Scale threshold*: a threshold ε is computed as the percentage of the best similarity value between two artefacts, i.e., $\varepsilon = c \cdot MaxSimilarity$, where $0 \le c \le 1$ [2]. Of course, this method can be applied when *MaxSimilarity* is a positive value. In this case, the higher the value of the parameter *c*, the smaller the set of documents returned by a query.

It is worth noting that if *MaxSimilarity* is 1, the scale threshold and the constant threshold methods are equivalent. The scale threshold method is useful when the maximum similarity measure is low, while the variable threshold method is useful when the distance between the maximum and minimum similarity is low.

5.2. Comparing different IR methods

In the first experiment we indexed all the artefacts within the same collection. Figure 3 shows the results. The 100% recall is reached with ε = 0.11 for the constant threshold (with about 12% precision) and ε = 10% for the variable threshold (with about 12% precision). For the cut point method 132 artefacts are necessary to reach the 100% recall (with about 11.5% precision). Generally, the constant threshold and variable threshold performs better than the cut point. It is worth noting that in this experiment we have not used the scale threshold, because for each query the maximum similarity measure was very high, thus giving similar results as the constant threshold. The best results were achieved with ε = 0.28 for the constant threshold, with ε = 31% for the variable threshold, and with 46 artefacts for the cut point method. For example, for the variable threshold we achieved a good compromise between recall (about 80%) and precision (about 24%).

We observed that the artefacts belonging to the same category have similar structure and this increases their similarity measure, even if they are not relevant to each other (*false alarm*) [16]. If a use case description is used as query, the related artefacts belonging to other categories will have a similarity measure lower than irrelevant use case descriptions. For this reason, we performed a second experiment, where the artefacts are indexed in four four different collections, one for each category of artefacts. Queries were then performed against each artefact subspace, thus achieving different ranked lists.

Figure 4 shows the results: 100% recall is reached with *c* = 0.23 for the scale threshold and ε = 17% for the variable threshold. For the cut percentage method 96% of the artefacts in each collections is necessary to reach the 100% recall. In this case we did not use the constant threshold, because it does not take into account the differences in the maximum similarity values achieved in the four different collections of artefacts. For the same reason we used a cut percentage rather than a fixed cut point. It is worth noting that the results achieved in this case are better than the results achieved with a single collection of artefacts (compare Figures 3 and 4). In particular, for the variable threshold about 35% precision with more than 80% recall is achieved for ε = 66%.

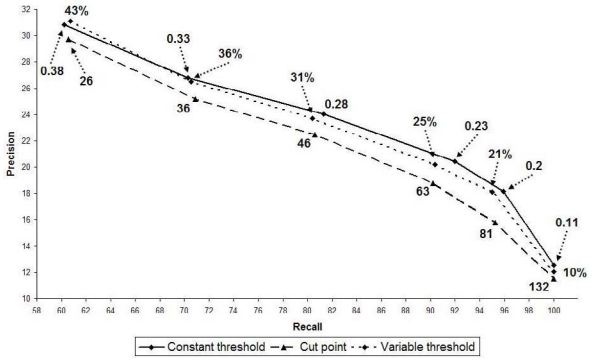

Figure 3. Precision/recall without categorization

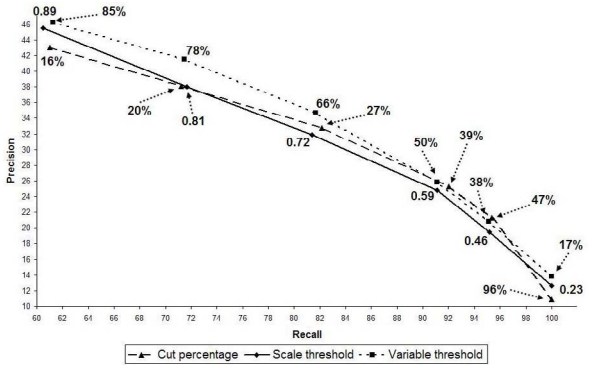

Figure 4. Precision/recall with categorization

To have more evidence that the categorization improves the results, Figure 5 compare the variable threshold results of the two previous experiments. It is worth noting that we compare the variable threshold performances because the variable threshold method, in general, gives the best results in both the experiments. We observed that when the recall is 80% with the categorization we have an

improvement of the precision of about 40% (35% against 25%) and when the recall is 70% the improvement is more than 60% (42% against 26%). Another advantage of using categorization is the fact that different thresholds can be used for different categories of artefacts, thus further improving the results. In conclusion, the two experiments suggest that the best method used to define the cosine threshold is the variable threshold with categorization of artefacts.

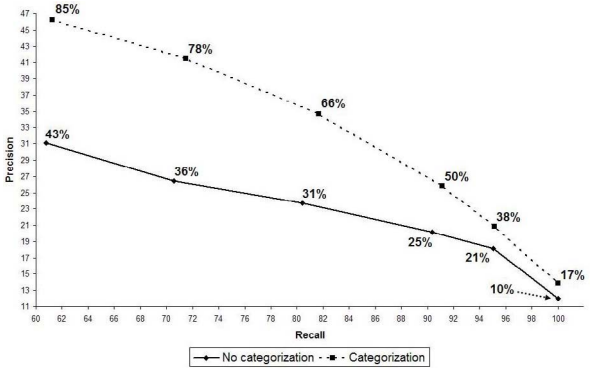

Figure 5. Variable threshold results

5.3. Analytical results

The precision/recall graphs shown in Figures 3-5 are useful to compare the different IR methods. To evaluate the effectiveness of the traceability link recovery tool we need a deeper analysis of the results. For this reason we analysed the results of recovering traceability links between all different pairs of artefact categories (16 pairs), using the variable threshold method. For sake of space we do not show all the results, but only the results achieved by tracing code classes on the other categories of artefacts.

Figure 6 shows the precision/recall graph, while Table 2 summarizes the characteristics of the experiment: the first column represents the artefact type (category); the second column shows the number of artefacts in each category (size of the artefact subspace), while the third column represents the number of classes used as queries; finally the fourth column shows the average number in the correct links between code classes and artefacts of the different categories. As we can see, only for the use case and the code class subspaces we used all the available code classes as queries; for the other two subspaces we could not use all the classes, because the set of available artefacts was incomplete and therefore some classes could not be traced on any interaction diagram and test case.

In the following we will discuss in particular the results achieved by tracing code classes on use cases and test case specifications, respectively. We do not show the results achieved by recovering traceability links between code classes and interaction diagram descriptions, because they are very similar to the results achieved by recovering traceability links between code classes and use case descriptions (see Figure 6). Moreover, we do not show the results achieved by recovering traceability links between code classes because they are not very good (see Figure 6); in our opinion IR methods are not adequate for this purpose or should at least be combined with other techniques [8].

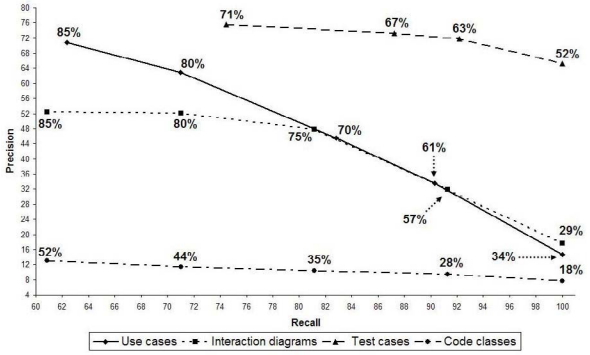

Figure 6. Precision/recall for code tracing

Artefact type	# artefacts	# queries (classes)	# average correct links
Use cases	30	37	2.66
Interaction diagrams	20	30	2.30
Test cases	63	13	14.10
Code classes	37	36	2.76

Table 2. Code class tracing statistics

Tables 3 and 4 summarizes the results of tracing code classes onto use cases and test case specifications, respectively. The first column represents the threshold value; the second column represents the average number of total artefacts retrieved by the tool, while the third and fourth columns represent the average number of irrelevant retrieved artefacts (false positives) and the average number of correct links missed by the tool, respectively. The fifth and sixth columns show the aggregate recall and precision values.

The higher threshold used in both cases is 95%. When we relax the selection criterion and decrease the threshold we get more false positives (the precision decreases) and fewer missed links (the recall increases). As we can see from the two tables and from Figure 6, tracing code classes onto test case specifications gives better results than tracing code classes onto use cases: for the use cases a 100% recall is achieved with a threshold of 33% and a precision of about 14%, while for the test case specification we get 100% recall with a threshold of 51% and a precision of about 65%.

For the code classes to use cases tracing a good compromise is achieved when the threshold is 70% (see Table 3):

in this case on average of the 2.66 links that are correct (see Table 2), only 0.54 links are missed by the tool, while of the 4.48 links retrieved by the tool, only 2.37 links are false positives. On the other hand, the better results for the code classes to test case specifications tracing are achieved when the threshold is 65% (see Table 4): in this case on average of the 14.10 links that are correct (see Table 2), only 1.50 links are missed by the tool, while of the 17.10 links retrieved by the tool, only 4.50 are false positive.

Threshold (%)	Retrieved	False Positives	Missed Links	Recall (%)	Precision (%)
95	1.23	0.26	1.69	36.56	79.07
90	1.48	0.31	1.49	44.09	78.85
85	2.23	0.65	1.09	59.13	70.51
80	2.80	0.97	0.83	68.82	65.30
75	3.49	1.43	0.60	77.42	59.02
70	4.48	2.37	0.54	79.57	47.13
65	5.57	3.31	0.40	84.95	40.51
60	7.00	4.63	0.29	89.25	33.88
55	8.34	5.94	0.26	90.32	28.77
50	10.20	7.74	0.20	92.47	24.09
45	12.54	10.03	0.14	94.62	20.05
35	16.80	14.17	0.03	98.92	15.65
33	18.03	15.37	0.00	100.00	14.74

Table 3. Code class to use case tracing results

Threshold (%)	Retrieved	False positives	Missed Links	Recall (%)	Precision (%)
95	3.30	0.40	11.20	20.57	87.88
90	4.30	0.50	10.30	26.95	88.37
85	6.10	0.90	8.90	36.88	85.24
80	8.60	1.50	6.70	52.48	83.15
75	10.70	2.00	5.40	61.70	81.31
70	13.90	3.40	3.60	74.47	75.54
65	17.10	4.50	1.50	89.36	73.68
60	19.20	5.90	0.80	94.33	69.27
55	20.10	6.60	0.60	95.74	67.16
51	21.60	7.50	0.00	100.00	65.28

Table 4. Code class to test case tracing results

It is worth noting that the data in Tables 3 and 4 are shown from the point of view of the tool. Indeed, the set of links missed by the tool corresponds to the set $WarningLinks_i(\varepsilon)$ defined in Section 4 from the point of view of the software engineer, in case he/she has identified and traced all correct links ($link_i = correct_i$). Also, in this case the set of false positives corresponds to the set $LostLinks_i(\varepsilon)$ from the point of view of the software engineer. Therefore, assuming that the set of links identified and traced by the software engineer is a subset of the correct links, the values in the column Missed Links in Tables 3 and 4 are an upper bound for the number of warning links. On the other hand, concerning the cardinality of the set $LostLinks_i(\varepsilon)$, an upper bound is given by the total number of artefacts retrieved by the tool (in case $link_i$ is empty), while a lower bound is given by the number of false positives retrieved by the tool (in case $link_i = correct_i$). It is worth noting that the cardinality of this set is acceptable also for a low threshold value. For example, when tracing code classes to use cases, if the initial set of links traced by the software engineer is empty, with a 70% threshold on average he/she can discover up to 2.11 correct links (difference between average retrieved links and average correct links) by analyzing only 4.48 retrieved links (see Table 3). On the other hand, if the software engineer would be accurate in manually tracing all correct links, he/she can have a confirmation of about 80% (ratio between 2.11 average correct links retrieved by the tool and 2.66 average correct links) of the traced links. Similar considerations can be made when tracing code classes to test case specifications.

6. Conclusion

Software artefact traceability is fundamental to effectively managing the development and evolution of software systems, to help in program comprehension, maintenance, impact analysis, and reuse of existing software. However, despite of its importance, contemporary artefact management tools provides a limited support to automatic or semi-automatic traceability link recovery.

In this paper we have shown how to enhance an artefact management system called ADAMS, with a traceability recovery tool based on a IR technique, namely Latent Semantic Indexing (LSI) [13]. The tool provides the software engineer with the set of links not traced by the software engineer and retrieved by the tool and the set of links traced by the software engineer and not retrieved by the tool. The manual analysis of the links in these two sets as well as the possibility of incrementally tuning the threshold helps the software engineer to identify relevant links not traced yet, as well as links previously traced that are actually not relevant or cases of inconsistencies in the usage of terms between traced software artefacts.

We have also presented a case study of using the traceability recovery tool on different types of software artefacts. We have shown how better results can be achieved using a variable threshold and categorizing the artefacts in the repository in different subspaces. The latter findings are similar to the results achieved by other authors with documents of different nature [16].

Future work will be devoted to integrating the traceability recovery tool in ADAMS and further experimenting it in software projects.

References

[1] I. Alexander, "Towards Automatic Traceability in Industrial Practice", *Proc. of the 1st Intern. Workshop on Traceability*, Edinburgh, UK, 2002, pp 26-31.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation", *IEEE Transactions on Software Engineering,* vol. 28, no. 10, 2002, pp. 970-983.

[3] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Identifying the Starting Impact Set of a Maintenance Request", *Proc. of 4th European Conference on Software Maintenance and Reengineering*, Zurich, Switzerland, IEEE CS Press, 2000, pp. 227-230.

[4] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, "Design-Code Traceability for Object Oriented Systems" *Annals of Software Engineering*, vol. 9, 2000, pp. 35–58.

[5] L. Aversano, A. De Lucia, M. Gaeta, and P. Ritrovato, "GENESIS: a Flexible and Distributed Environment for Cooperative Software Engineering", *Proc. of 15th Intern. Conference on Software Engineering and Knowledge Engineering*, S. Francisco, CA, USA, 2003, pp. 497-502.

[6] L. C. Briand, Y. Labiche and L. O'Sullivan, "Impact Analysis and Change Management of UML Models" *Proc. of IEEE Intern. Conference on Software Maintenance*, Amsterdam, The Netherlands, IEEE CS Press, 2003, pp. 256-265.

[7] C. Boldyreff, D. Nutter, and S. Rank, "Active Artefact Management for Distributed Software Engineering", *Proc. of 26th IEEE Annual Intern. Computer Software and Applications Conference*, Oxford, UK, IEEE CS Press, 2002, pp. 1081-1086.

[8] B. Caprile and P. Tonella, "Nomen Est Omen: Analyzing the Language of Function Identifiers", *Proc. of 6th IEEE Working Conference on Reverse Engineering*, Atlanta, GA, USA, IEEE CS Press, 1999, pp. 112-122.

[9] J.Y.J. Chen and S.-C. Chou, "Consistency Management in a Process Environment", *The Journal of Systems and Software*, vol. 47, 1999, pp. 105-110

[10] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event-Based Traceability for Managing Evolutionary Change", *IEEE Transactions on Software Engineering*, vol. 29, no. 9, 2003, pp. 796-810.

[11] J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, vol. 1: Theory, Chapter 5: "Real rectangular matrices", Brikhauser, Boston, 1985.

[12] J. Dag, B. Regnell, P. Carlshamre, M. Andersson, J. Karlsson, "A Feasibility Study of Automated Natural Language Requirements Analysis in Market-driven Development", *Requirements Engineering*, vol. 7, no. 1, 2002, pp. 20-33.

[13] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, no. 41, 1990, pp. 391-407.

[14] A. De Lucia, F. Fasano, R. Francese, and G. Tortora, "ADAMS: an Artefact-based Process Support System", *Proc. of 16th Intern. Conference of Software Engineering and Knowledge Engineering*, Banff, Alberta, Canada, 2004, pp. 31-36.

[15] R. Domges and K. Pohl, "Adapting Traceability Environments to Project Specific Needs", *Communications of the ACM*, vol. 41, no. 12, 1998, pp. 55-62.

[16] S. T. Dumais, "LSI meets TREC: A status report", In D. Harman (Ed.) *The First Text REtrieval Conference* (TREC-1), NIST special publication 500-207, pp. 137-152.

[17] D. Harman, "Ranking Algorithms", in *Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, Englewood Cliffs, NJ, 1992, pp. 363–392.

[18] J. Huffman Hayes, A. Dekhtyar, and J. Osborne, "Improving Requirements Tracing via Information Retrieval", *Proc. of 11th IEEE Intern. Requirements Engineering Conference*, Monterey, CA, USA, IEEE CS Press, 2003, pp. 138-147.

[19] Holagent Corporation product RDD-100, http://www.holagent.com/new/products/modules.html

[20] J. Konclin and M. Bergen, "Gibis: A Hypertext Tool for Exploratory Policy Discussion," *ACM Transactions Office Information Systems*, vol. 6, no. 4, 1988, pp. 303–331.

[21] D. Leffingwell, "Calculating Your Return on Investment from More Effective Requirements Management", *Rational Software Corporation*, 1997. Available online from http://www.rational.com/products/whitepapers.

[22] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing", *Proc. of 25th Intern. Conference on Software Engineering*, Portland, Oregon, USA, 2003, pp. 125-135.

[23] G.C. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation" *IEEE Transactions on Software Engineering*, vol. 27, no. 4, 2001, pp. 364-380.

[24] F.A.C. Pinhero and J.A. Goguen, "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*, vol. 13, no. 2, 1996, pp. 52–64.

[25] B. Ramesh and V. Dhar, "Supporting Systems Development Using Knowledge Captured During Requirements Engineering" *IEEE Transactions on Software Engineering*, vol. 9, no. 2, 1992, pp. 498–510.

[26] Rational RequisitePro, http://www.rational.com/products/reqpro/index.jsp.

[27] M. Sefika, A. Sane, and R.H. Campbell, "Monitoring Compliance of a Software System with Its High-Level Design Models", *Proc. of 16th Intern. Conference on Software Engineering*, Berlin, Germany, 1996, pp. 387–396.

[28] M. Smith, D. Weiss, P.Wilcox, and R.Dewer, "The Ophelia traceability layer", in *Cooperative Methods and Tools for Distributed Software Processes*, A. Cimitile, A. De Lucia, and H. Gall (editors), Franco Angeli, 2003, pp. 150-161.

[29] Telelogic product DOORS, http://www.telelogic.com.

[30] A. von Knethen and M. Grund, " QuaTrace: A Tool Environment for (Semi-) Automatic Impact Analysis Based on Traces" *Proc. of IEEE Intern. Conference on Software Maintenance*, Amsterdam, The Netherlands, IEEE CS Press, 2003, pp. 246-255.

[31] J. Weidl and H. Gall, "Binding Object Models to Source Code" *Proc. of 22nd IEEE Annual Intern. Computer Software and Applications Conference*, Vienna, Austria, IEEE CS Press, 1998, pp. 26–31.

[32] A. Zisman, G. Spanoudakis, E. Perez-Miñana, and P. Krause, "Tracing Software Requirements Artifacts", *Proc. of Intern. Conference on Software Engineering Research and Practice*, Las Vegas, Nevada, USA, 2003, pp. 448-455.