Recovering Traceability Links in Software Artifact Management Systems using Information Retrieval Methods

ANDREA DE LUCIA, FAUSTO FASANO, ROCCO OLIVETO,
and GENOVEFFA TORTORA
University of Salerno

The main drawback of existing software artifact management systems is the lack of automatic or semi-automatic traceability link generation and maintenance. We have improved an artifact management system with a traceability recovery tool based on Latent Semantic Indexing (LSI), an information retrieval technique. We have assessed LSI to identify strengths and limitations of using information retrieval techniques for traceability recovery and devised the need for an incremental approach. The method and the tool have been evaluated during the development of seventeen software projects involving about 150 students. We observed that although tools based on information retrieval provide a useful support for the identification of traceability links during software development, they are still far to support a complete semi-automatic recovery of all links. The results of our experience have also shown that such tools can help to identify quality problems in the textual description of traced artifacts.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management—*Software configuration management*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Singular value decomposition*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*indexing methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Documentation, Management

Additional Key Words and Phrases: Software artifact management, traceability management, impact analysis, latent semantic indexing

ACM Reference Format:

De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G. 2007. Recovering traceability links in software artifact management systems using information retrieval methods. ACM Trans. Softw. Eng. Methodol. 16, 4, Article 13 (September 2007), 50 pages. DOI = 10.1145/1276933.1276934 http://doi.acm.org/10.1145/1276933.1276934

Authors' address: Dipartimento di Matematica e Informatica, Università di Salerno, 84084 Fisciano (SA), Italy; email: {adelucia, ffasano, roliveto, tortora}@unisa.it.

1. INTRODUCTION

Software artifact traceability is the ability to describe and follow the life of an artifact (requirements, code, tests, models, reports, plans, etc.) developed during the software lifecycle in both forward and backward directions (e.g., from requirements to the modules of the software architecture and the code components implementing them and vice-versa) [Gotel and Finkelstein 1994]. Traceability can provide important insights into system development and evolution assisting in both top-down and bottom-up program comprehension, impact analysis, and reuse of existing software, thus giving essential support in understanding the relationships existing within and across software requirements, design, and implementation [Palmer 2000].

Regardless 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 overruns and failures [Domges and Pohl 1998; Leffingwell 1997]. Although several research and commercial tools are available that support traceability between artifacts [Cleland-Huang et al. 2003; Conklin and Begeman 1988; Holagent Corporation 2006; Pinheiro and Goguen 1996; Ramesh and Dhar 1992; Rational Software 2006; Telelogic 2006], the main drawback of these tools is the lack of automatic or semi-automatic traceability link generation and maintenance [Alexander 2002]. There are several tools that require the user to assign keywords to all the documents prior to tracing and most of them do not provide support for easily retracing new versions of documents. 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 [Cleland-Huang et al. 2003].

The need to provide the software engineer with methods and tools supporting traceability recovery has been widely recognized in the last years [Antoniol et al. 2002; Egyed and Grünbacher 2002; Marcus and Maletic 2003; Murphy et al. 2001; Richardson and Green 2004; von Knethen and Grund 2003; Zisman et al. 2003]. In particular, several researchers have recently applied Information Retrieval (IR) techniques [Baeza-Yates and Ribeiro-Neto 1999; Deerwester et al. 1990; Harman 1992] to the problem of recovering traceability links between artifacts of different types [Antoniol et al. 2000a; 2002; Cleland-Huang et al. 2005; Dag et al. 2002; De Lucia et al. 2004b; 2004c; Huffman Hayes et al. 2003; 2006; Marcus and Maletic 2003; Settimi et al. 2004]. IR-based methods recover traceability links on the basis of the similarity between the text contained in the software artifacts. The rationale behind them is the fact that most of the software documentation is text based or contains textual descriptions and that programmers use meaningful domain terms to define source code identifiers. Although the results reported in these papers seem promising, so far no such an IR tool has been actually integrated within an artifact management system and experimented with real users during software development and maintenance. The authors of previous papers get to the general conclusion that IR methods can help the software engineer based on a performance analysis conducted on software repositories of completed projects.

In particular, for each experiment they compare the results of IR methods against a traceability matrix intended to contain the correct links between the artifacts of the repository. They also devise the need to define a similarity threshold and consider as candidate traceability links only the pairs of artifacts with similarity above such a threshold. The lower the threshold, the higher the probability to retrieve all correct links. However, from these studies it appears that IR-based traceability recovery methods and tools are in general far from helping the software engineer in the identification of all correct links: when attempting to recover all correct links with such a tool, the software engineer would also need to analyze and discard a much higher number of false positives that would also be recovered by the tool. Indeed, when discussing their results previous authors also try to identify an "optimal" threshold enabling the retrieval of as many correct links as possible, while keeping low the effort required to analyze and discard false positives. However, this ideal threshold is not easy to identify, as it can change together with the type of artifacts and projects.

In this article, we extend the preliminary results reported in De Lucia et al. [2004c; 2005a] and critically investigate to what extent IR-based methods can be used to support the software engineer in traceability recovery during software development and maintenance. We have integrated an IR-based traceability link recovery tool in ADAMS (ADvanced Artifact Management System), an artifact-based process support system for the management of human resources, projects, and software artifacts [De Lucia et al. 2004a]. The traceability recovery tool is based on Latent Semantic Indexing (LSI) [Deerwester et al. 1990], an advanced IR method. As noticed by Marcus and Maletic [2003], the advantage of LSI is that it is able to achieve the same performances as the classical probabilistic or vector space models [Baeza-Yates and Ribeiro-Neto 1999; Harman 1992] without requiring a preliminary morphological analysis (stemming) of the document words. This allows the method to be applied without large amounts of text pre-processing. In addition, using a method that avoids stemming is particularly useful for languages, such as Italian (the language of the documents we used in our experiments), that present a complex grammar, verbs with many conjugated variants, words with different meanings in different contexts, and irregular forms for plurals, adverbs, and adjectives [Antoniol et al. 2002]. Although in our implementation we have used LSI as an information retrieval technique, we are not claiming that this is in general the best method: we have not compared LSI with other IR-based techniques presented in the literature as this was out of the scope of this article and it is not difficult to replace the traceability recovery module implemented in ADAMS with another one that implements a better method.

The specific contributions of our article are:

—an assessment of LSI as a traceability recovery technique where we show how: (i) using such a technique to recover all traceability links is not feasible in general, as the number of false positives grows up too rapidly when the similarity of artifact pairs decreases below an "optimal" threshold; (ii) the "optimal" similarity threshold changes depending on the type of artifacts and projects; and (iii) such a threshold can be approximated case by case within an incremental traceability recovery process, where the similarity threshold is tuned by incrementally decreasing it;

—the definition and implementation of a tool that helps the software engineer to discover emerging traceability links during software evolution, as well as to monitor previously traced links; the latter is particularly useful when the similarity of previously traced artifacts is too low, as this might indicate the fact that the link should not exist (anymore) or that the traced artifacts have some quality problems, in terms of poor textual description;

—an evaluation of the usefulness of the proposed approach and tool during the development of seventeen projects involving about 150 students; the results of the experience show that IR-based traceability recovery tools help software engineers to improve their recovery performances, although they are still far to support a complete semi-automatic recovery of all traceability links.

The remainder of the article is organized as follows: Section 2 discusses related work, while Sections 3 and 4 present overviews of ADAMS and LSI, respectively. Section 5 assesses LSI as a traceability recovery technique through the analysis of a case study. Section 6 presents the architecture and functionality of the traceability recovery tool, while Section 7 discusses the results of the experience and the evaluation of the tool. Finally, Section 8 concludes and discusses lessons learned.

2. RELATED WORK

The subject of this article covers three areas of interest: traceability management, traceability recovery, and information retrieval for software engineering. Related work on each of these topics will be addressed below.

2.1 Traceability Management

Several research and commercial tools are available that support traceability between artifacts. DOORS [Telelogic 2006] and Rational RequisitePro [Rational Software 2006] are commercial tools that provide effective support for recording, displaying, and checking the completeness of traced artifacts using operations such as drag and drop [Telelogic 2006], or clicking on a cell of a traceability link matrix [Rational Software 2006]. RDD.100 (Requirements Driven Development) [Holagent Corporation 2006] uses an ERA (Entity/Relationship/Attribute) repository to capture and trace complicated sets of requirements, providing functionalities to graphically visualize how individual pieces of data relate to each other and to trace back to their source.

TOOR [Pinheiro and Goguen 1996] is a research tool in which traceability is not modelled in terms of simple links, but through user-definable relations that are meaningful for the kind of connection being made. This lets developers distinguish among different links between the same objects. Moreover, TOOR can relate objects that are not directly linked using mathematical properties of relations such as transitivity. A hypertext system designed to facilitate the capture of early design deliberations (rationale), called gIBIS, has been presented in Conklin and Begeman [1988]. It implements a specific method of design deliberation, called Issue Based Information System (IBIS) [Rittel and Kunz 1970], based on the principle that the design process for complex problems is fundamentally a conversation among the stakeholders in which they bring their respective expertise and viewpoints to the resolution of design issues. The system makes use of colours (gIBIS stands for graphical IBIS) and a high speed relational database server to facilitate building and browsing typed IBIS networks made up of nodes (e.g., issues in the design problem) and links among them. REMAP (REpresentation and MAintenance of Process knowledge) [Ramesh and Dhar 1992] is another conceptual model based on IBIS, that relates process knowledge to the objects that are created during the requirements engineering process. REMAP offers a built-in set of types of traceability relations with predefined semantics. It was developed using an empirical study of problem-solving behavior of individuals and groups of information system professionals.

Recently, artifact traceability has been tackled within the Ophelia project [Smith et al. 2003] which pursued the development of a platform supporting software engineering in a distributed environment. In Ophelia, the artifacts of the software engineering process are represented by CORBA objects. A graph is created to maintain relationships between these elements and navigate among them. OSCAR [Boldyreff et al. 2002] is the artifact management subsystem of the GENESIS environment [Aversano et al. 2003]. It has been designed to noninvasively interoperate with workflow management systems, development tools, and existing repository systems. Each artifact in OSCAR possesses a collection of standard meta-data and is represented by an XML document containing both meta-data and artifact data that include linking relationships with other artifacts. OSCAR introduces the notion of “active” software artifacts that are aware of their own evolution. To support such an active behavior, every operation on an artifact generates events that may be propagated by an event monitor to artifacts deemed to be interested in such events by their relationships with the artifact generating the event. Other tools [Chen and Chou 1999; Cleland-Huang et al. 2003] also combine the traceability layer with event-based notifications to make users aware of artifact modifications. Chen and Chou [1999] have proposed a method for consistency management in the Aper process environment. The method is based on maintaining different types of traceability relations between artifacts, including composition relations, and uses triggering mechanisms to identify artifacts affected by changes to a related artifact. Cleland-Huang et al. [2003] have developed EBT (Event Based Traceability), an approach based on a publish-subscribe mechanism between artifacts. When a change occurs on a given artifact having the publisher role, notifications are sent to all the subscriber (dependent) artifacts.

Traceability has also been used in the PROSYT environment [Cugola 1998] to model software processes in terms of the artifacts to be produced and their interrelationships. This artifact-based approach results in process models composed of simpler and more general operations than the operations identified using an activity based approach. PROSYT is able to manage the inconsistencies between a process instance and the process model [Cugola et al. 1996] by tracing the deviating actions and supporting the users in reconciling the enacted process and the process model, if necessary.

The connection between traceability and inconsistency management has also been highlighted by Spanoudakis and Zisman [2001] who emphasized the benefits that software requirement traceability can provide to support the stakeholders in establishing the cause of an inconsistency. In general, inconsistency management is the process by which inconsistencies between software models are handled to support the goals of the involved stakeholders [Finkelstein et al. 1996] and includes activities for detecting, diagnosing, and handling them [Nuseibeh 1996]. In this field, the contribution of traceability has been to provide schemes for annotating model elements with the stakeholders who contributed to their construction [Spanoudakis and Zisman 2001]. Such schemes can be used to locate the stakeholders who have constructed the relevant model elements or whose perspectives are expressed in them. However, this would be only one step towards the final objective, which is to identify the conflicts between the model contributors through the identification of inconsistencies in the related software models.

An important issue of traceability management related to consistency management concerns the evolution of traceability links. Nistor et al. [2005] developed ArchEvol an environment that manages the evolution of architecture-to-implementation traceability links throughout the entire software life cycle. The proposed solution maintains the mapping between architecture and code and ensures that the right versions of the architectural components map onto the right versions of the code (and vice versa), when changes are made either to the architecture or to the code. Nguyen et al. [2005] have developed Molhado, an architectural configuration management system that automatically updates traceability links between architecture and code artifacts during software evolution. Molhado uses a single versioning mechanism for all software components and for the connections between them and can track changes at a very fine-grained level, allowing users to return to a consistent state of a single node or link. Maletic et al. [2005] propose an approach to support the evolution of traceability links between source code and UML (Unified Modelling Language) artifacts. The authors use an XML-based representation for both the source code and the UML artifacts and applies meta-differencing whenever an artifact is checked-in to identify specific changes and identify traceability links that might have been affected by the change.

2.2 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. [2001] 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. Weidl and Gall [1998] followed the idea of adopting a more tolerant string matching, where procedural applications are rearchitectured into OO systems. Antoniol et al. [2000b] also present a method to trace C++ classes onto 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.

The approach adopted by von Knethen and Grund [2003] 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 Briand et al. [2003] 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 (Object Constraint Language) constraints on an adaptation of the UML meta-model. Sefika et al. [1996] have developed a hybrid approach that integrates logic-based static and dynamic visualization and helps determining design-implementation congruence at various levels of abstraction.

Egyed and Grünbacher [2002] propose to recover traceability links between requirements and Java programs by monitoring the Java programs to record which program classes are used when scenarios are executed. Further analysis then automatically refine these links. Richardson and Green [2004] have proposed a novel technique for automatically deriving traceability relations between parts of a specification and parts of the synthesised program for processes in which one artifact is automatically derived from another.

Approaches to recover traceability links between source code artifacts have also been proposed that use data mining techniques on software configuration management repositories [Gall et al. 1998; 2003; Ying et al. 2004; Zimmermann et al. 2005]. Gall et al. [1998] were the first to use release data to detect logical coupling between modules. They use CVS history to detect fine-grained logical coupling between classes, files, and functions. Their methodology investigates the historical development of classes measuring the time when new classes are added to the system and when existing classes are changed and maintaining attributes related to changes of classes, such as the author or the date of a change. Such information is inspected to reveal common change behavior of different parts of the system during the evolution (referred to as logical coupling) [Gall et al. 2003]. Ying et al. [2004] use association rule mining on CVS version archives. Their approach is based on the mining of change patterns (files that were changed together frequently in the past) from the source code change history of the system. Mined change patterns are used to recommend possible relevant files as a developer performs a modification task. Similarly, Zimmerman et al. [2005] have developed an approach that also uses association rule mining on CVS data to recommend source code that is potentially relevant to a given fragment of source code. The rules determined by their approach can describe change associations between fine-grained program entities, such as functions or variables, as well as coarse-grained entities, such as classes or files. Coarse-grained rules have a higher support count and usually return more results. However, they are less precise in the location and, thus, only of limited use for guiding programmers.

Other approaches consider textual documents written in natural language, such as requirements documents. Zisman et al. [2003] automate the generation of traceability relations between textual requirement artifacts and object models using heuristic rules. These rules match syntactically related terms in the textual parts of the requirement artifacts 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 [Baeza-Yates and Ribeiro-Neto 1999; Deerwester et al. 1990; Harman 1992] to the problem of recovering traceability links between software artifacts. Dag et al. [2002] 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.

Antoniol et al. [2002] use information retrieval methods based on probabilistic and vector space models [Baeza-Yates and Ribeiro-Neto 1999; Harman 1992]. They apply and compare the two methods on two case studies to trace C++ source code onto manual pages and Java code to functional requirements, respectively: the results show that the two methods have similar performances when a preliminary morphological analysis (stemming) of the terms contained in the software artifacts is performed. In Antoniol et al. [2000a] the vector space model is used to trace maintenance requests on software documents impacted by them.

Antoniol et al. [2000c] discuss how a traceability recovery tool based on the probabilistic model can improve the retrieval performances by learning from user feedbacks. Such feedbacks are provided as input to the tool in terms of a subset of correct traceability links (training set) and the results show that significant improvements are achieved both with and without using a preliminary stemming. Di Penta et al. [2002] also use this approach to recover traceability links between source code and free text documents in software systems developed with extensive use of COTS, middleware, and automatically generated code.

Marcus and Maletic [2003] use Latent Semantic Indexing (LSI) [Deerwester et al. 1990], an extension of the vector space model: they perform the same case studies as in [Antoniol et al. 2002] and compare the performances of LSI with respect to the vector space and probabilistic models. They demonstrate that LSI can achieve very good performances without the need for stemming that on the other hand is required for the vector space and probabilistic models to achieve similar results. Maletic et al. [2003] also propose to use LSI to construct and maintain hyper textual conformance graphs among software artifacts. In a recent paper, Marcus et al. [2005] discuss in which cases visualizing traceability links is opportune, as well as what information concerning these links should be visualized and how. They also present a prototype tool based on the traceability recovery tool presented in Marcus and Maletic [2003] to support the software engineer during recovery, visualization, and maintenance of traceability links.

Huffman Hayes et al. [2003] use the vector space model to recover traceability links between requirements and compare the results achieved by applying different variants of the base model: in particular, the use of document keywords is shown as a mean to improve the results of the vector space model. In Huffman Hayes et al. [2006] they address the issues related to improving the overall quality of the requirements tracing process. In particular, they define requirements for a tracing tool based on analyst responsibilities in the tracing process and present a prototype tool, called RETRO (REquirements TRacing On-target), to address these requirements. RETRO implements both VSM and LSI for determining requirements similarity. Two case studies conducted on two NASA artifact repositories are also presented where the tool is used to recover links between requirement artifacts. Good results seem to be achieved when user feedbacks are used to change the weights in the term-by-document matrix of the vector space model.

Settimi et al. [2004] investigate the effectiveness of IR methods for tracing requirements to UML artifacts, code, and test cases. In particular, they compare the results achieved applying different variants of the vector space model to generate links between software artifacts. Cleland-Huang et al. [2005] propose an improvement of the dynamic requirements traceability performance. The authors introduce three different strategies for incorporating supporting information into a probabilistic retrieval algorithm, namely hierarchical modelling, logical clustering of artifacts, and semi-automated pruning of the probabilistic network.

LSI is also used by Lormans and van Deursen [2006] to reconstruct traceability links between requirements and design artifacts and between requirements and test case specifications. The authors propose a new strategy for selecting traceability links and experiment the proposed approach in three case studies. They also discuss the most important open research issues concerning the application of LSI to recover traceability links in industrial projects.

2.3 Information Retrieval Applied to Software Engineering

Besides traceability recovery, the application of information retrieval and other natural language processing techniques to software engineering has been an issue for several researches. Analysis of comments and mnemonics for identifiers in the source code can be useful to associate domain concepts with program fragments and vice-versa. The importance of analysing such informal information has been addressed by Biggerstaff [1989] and Merlo et al. [1993] who adopt approaches for design recovery based on semantic and neural networks, in addition to traditional syntactic based approaches. The need for using such methods derives from the fact that comments and source code identifiers have an information content with an extremely large degree of variance between systems and, often, between different segments of the same system that cannot be analysed by simply using formal parsers.

Zhao et al. [2004] present a static and noninteractive method for feature location. The starting point of their approach is to locate some initial specific functions for each feature through IR. Based on the initial specific functions, they recover all relevant functions through navigating a static representation of the code named Branch-Reserving Call Graph (BRCG). The BRCG is an expansion of the call graph with branching and sequential information. Due to the characteristics of the BRCG, they also acquire the pseudo execution traces for each feature. LSI has also been applied to the problem of mapping high level concepts expressed in natural language to the relevant source code components implementing them [Marcus et al. 2004] and combined with structural information to cluster together software components for program comprehension [Maletic and Marcus 2001].

Another software engineering field that has largely exploited information retrieval methods is software reuse. In particular, the adoption of IR has been mainly aimed at automatically constructing reusable software libraries by indexing software components. Maarek et al. [1991] introduce an IR method to automatically assemble software libraries based on a free text indexing scheme. The method uses attributes automatically extracted from natural language documentation to build a browsing hierarchy which accepts queries expressed in natural language. REUSE [Arnold and Stepoway 1988] is an information retrieval system which stores software objects as textual documents in view of retrieval for reuse. ALICE [Pighin 2001] is another example of a system that exploits information retrieval techniques for automatic cataloguing software components for reuse. The RSL system [Burton et al. 1987] extracts free-text single-term indices from comments in source code files looking for keywords such as "author," "date," and so on. Similarly, CATALOG [Frakes and Nejmeh 1987] stores and retrieves C components, each of which is individually characterised by a set of single-term indexing features automatically extracted from natural language headers of C programs.

Di Lucca et al. [2002] apply different information retrieval and machine learning approaches (including vector space model, probabilistic model, support vectors, classification trees and k-nearest neighbour classification) to the problem of automatically classifying incoming maintenance requests and routing them to specialized maintenance teams. For each of the experimented approach they use a training set of correctly classified maintenance request; new incoming maintenance requests are compared against the maintenance requests in the training set and classified according to some distance metric varying with the used approach.

3. ADAMS

ADAMS (ADvanced Artifact Management System) is an artifact-based process support system [De Lucia et al. 2004a]. It poses a great emphasis on the artifact life cycle by associating software engineers with the different operations that can be performed on an artifact. ADAMS also supports quality management by associating each artifact type with a standard template, according to the quality manual of the organization, as well as a standard checklist that can be used during the review of the artifact.

The support for cooperation is provided through typical configuration management features. ADAMS enables groups of people to work on the same artifact, depending on the required roles. Different software engineers can access the same artifact according to a lock-based policy or concurrently, if branch versions of the artifact 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 artifact. Context awareness is also supported through event notifications: software engineers working on an artifact are notified when a new branch is created by another worker. This provides a solution to the isolation problem for resources working on the same artifact in different workspaces [Sarma and van der Hoek 2002]: in fact, context awareness allows to identify potential conflicts before they occur, because the system is able to notify interested resources as soon as an artifact is checked-out and potentially before substantial modifications have been applied to it.

Artifacts in ADAMS can be hierarchically defined through composition links and managed through hierarchical versioning policies. In addition, ADAMS enables software engineers to create and store traceability links between artifacts of the same or different types. Versions of composition and traceability links are also maintained in ADAMS, besides artifact versions.

Traceability links in ADAMS are useful for impact analysis and change management during software evolution. The traceability links can be visualized by a software engineer and browsed to look at the state of previously developed artifacts, to download latest artifact versions, or to subscribe events on them and receive notifications concerning their development [De Lucia et al. 2005b]. An example of such events is the modification of the status of an artifact or the creation of a new version for it. A number of events are automatically notified without any need for subscription. An example is the notification to a software engineer that he/she has been allocated to an artifact.

Events concerning the production of new versions of an artifact are propagated through the traceability layer of ADAMS to the artifacts impacted directly or indirectly by the change (and consequently to the software engineers responsible for them) [De Lucia et al. 2005b]. This reduces the overload of subscribing several events for notification and prevents from forgetting indirect but essential subscriptions. Finally, a software engineer may send a feedback whenever he/she discovers an inconsistency on an artifact his/her work depends on. Feedbacks are then notified to the software engineer responsible for the artifact.

In the first release of ADAMS [De Lucia et al. 2004a] the charge of traceability link identification was delegated to the software engineer, who has the responsibility to manage traceability links whenever new artifacts are added to the project, existing artifacts are removed, or new versions are checked-in. As the number of project artifacts grows-up, this task tends to be hard to manage, so automatic or semi-automatic tools are needed. For this reason, we have integrated in ADAMS a traceability recovery tool based on Latent Semantic Indexing (LSI) [Deerwester et al. 1990; Dumais 1992], an advanced IR technique that extends the Vector Space Model (VSM) [Baeza-Yates and Ribeiro-Neto 1999; Harman 1992]. VSM and LSI are discussed in the next section.

4. LATENT SEMANTIC INDEXING

IR methods index the documents in a document space as well as the queries by extracting information about the occurrences of terms within them. This information is used to define similarity measures between queries and documents. In the case of traceability recovery, this similarity measure is used to identify that a traceability link might exist between two artifacts, one of which is used as query.

4.1 Vector Space Model

In the Vector Space Model (VSM), documents and queries are represented as vectors of terms that occur within documents in a collection [Baeza-Yates and Ribeiro-Neto 1999; Harman 1992]. Therefore, a document space in VSM is described by a m x n matrix, where m is the number of terms, and n is the number of documents in the collection. Often this matrix is referred to as the *term-by-document matrix*. A generic entry $a_{i,j}$ of this matrix denotes a measure of the weight of the ith term in the jth document. Different measures have been proposed for this weight [Salton and Buckley 1988]. In the simplest case, it is a boolean value, either 1 if the ith term occurs in the jth document, or 0 otherwise; in other cases, more complex measures are constructed based on the frequency of the terms in the documents. In particular, these measures apply both a local and global weightings to increase/decrease the importance of terms within or among documents. Specifically, we can write:

$$a_{i,j} = L(i, j) \cdot G(i)$$

where $L(i, j)$ is the local weight of the ith term in the jth document and $G(i)$ is the global weight of the ith term in the entire document space. In general, the local weight increases with the frequency of the ith term in the jth document, while the global weight decreases as much as the ith term is spread across the documents of the document space. Dumais [1991] has conducted a comparative study among different local and global weighting functions within experiments with Latent Semantic Indexing (LSI). The best results have been achieved by scaling the term frequency by a logarithmic factor for the local weight and using the entropy of the term within the document space for the global weight:

$$L(i, j) = \log(tf_{ij} + 1) \quad G(i) = 1 - \sum_{j=1}^{n} \frac{p_{ij} \log(p_{ij})}{\log(n)}$$

where tf_{ij} is the frequency of the ith term in the jth document and p_{ij} is defined as:

$$p_{ij} = \frac{tf_{ij}}{\sum_{k=1}^{n} tf_{ik}}$$

We also use these two functions in our implementation of LSI. An advantage of using the entropy of a term to define its global weight is the fact that it takes into account the distribution of the term within the document space.

From a geometric point of view, each document vector (columns of the term-by-document matrix) represents a point in the m-space of the terms. Therefore, the similarity between two documents in this space is typically measured by the cosine of the angle between the corresponding vectors, which increases as more terms are shared. In general, two documents are considered similar if their corresponding vectors point in the same (general) direction.

4.2 Singular Value Decomposition

A common criticism of VSM is that it does not take into account relations between terms. For instance, having "automobile" in one document and "car" in another document does not contribute to the similarity measure between these two documents. LSI was developed to overcome the synonymy and polysemy problems, which occur with the VSM model [Deerwester et al. 1990]. In LSI, the dependencies between terms and between documents, in addition to the associations between terms and documents, are explicitly taken into account. LSI assumes that there is some underlying or "latent structure" in word usage that is partially obscured by variability in word choice, and uses statistical techniques to estimate this latent structure. For example, both "car" and "automobile" are likely to co-occur in different documents with related terms, such as "motor," "wheel," etc. LSI exploits information about co-occurrence of terms (latent structure) to automatically discover synonymy between different terms.

LSI defines a term-by-document matrix A as well as VSM. Then it applies singular value decomposition (SVD) [Cullum and Willoughby, 1985] to decompose the term-by-document matrix into the product of three other matrices:

$$A = T_0 \cdot S_0 \cdot D_0,$$

where T_0 is the $m \times r$ matrix of the terms containing the left singular vectors (rows of the matrix), D_0 is the $r \times n$ matrix of the documents containing the right singular vectors (columns of the matrix), S_0 is an $r \times r$ diagonal matrix of singular values, and r is the rank of A. T_0 and D_0 have orthogonal columns, such that:

$$T_0^T \cdot T_0 = D_0 \cdot D_0^T = I_r.$$

SVD can be viewed as a technique for deriving a set of uncorrelated indexing factors or concepts [Deerwester et al. 1990], whose number is given by the rank r of the matrix A and whose relevance is given by the singular values in the matrix S_0. Concepts "represent extracted common meaning components of many different words and documents" [Deerwester et al. 1990]. In other words, concepts are a way to cluster related terms with respect to documents and related documents with respect to terms. Each term and document is represented by a vector in the r-space of concepts, using elements of the left or right singular vectors. The product $S_0 \cdot D_0$ ($T_0 \cdot S_0$, respectively) is a matrix whose columns (rows, respectively) are the document vectors (term vectors, respectively) in the r-space of the concepts. The cosine of the angle between two vectors in this space represents the similarity of the two documents (terms, respectively) with respect to the concepts they share. In this way, SVD captures the underlying structure in the association of terms and documents. Terms that occur in similar documents, for example, will be near each other in the r-space of concepts, even if they never co-occur in the same document. This also means that some documents that do not share any word, but share similar words may none the less be near in the r-space.

SVD allows a simple strategy for optimal approximate fit using smaller matrices [Deerwester et al. 1990]. 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. 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 rows of D_0 to obtain T and D respectively. The result is a reduced model:

$$A \approx A_k = T \cdot S \cdot D,$$

where the matrix A_k is only approximately equal to A and is of rank $k < r$. The truncated SVD captures most of the important underlying structure in the association of terms and documents, yet at the same time it removes the noise or variability in word usage that plagues word-based retrieval methods. Intuitively, since the number of dimensions k is much smaller than the number of unique terms m, minor differences in terminology will be ignored.

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 a choice is an open issue in the factor analysis literature [Deerwester et al. 1990; Dumais 1992]. In the application of LSI to information retrieval, good performances have been achieved using about 100 concepts on a document space of about 1,000 documents and a vocabulary of about 6,000 terms [Deerwester et al. 1990]. With much larger repositories (between 20,000 and 220,000 documents and between 40,000 and 80,000 terms), good results have been achieved using between 235 and 250 concepts [Dumais 1992].

5. ASSESSING LATENT SEMANTIC INDEXING AS A TRACEABILITY RECOVERY METHOD

In this section, we evaluate LSI in the context of traceability recovery. We discuss the results of a case study where LSI has been applied to software artifacts of different types [De Lucia et al. 2004c]. These results have been used to make choices in the design of the traceability recovery tool integrated in ADAMS.

IR-based traceability recovery methods compare a set of *source* artifacts (used as a query) against another (even overlapping) set of *target* artifacts and rank the similarity of all possible pairs (candidate traceability links). Moreover, they use some method to present the software engineer only the subset of top links in the ranked list (*retrieved* links). Some methods cut the ranked list regardless of the values of the similarity measure:

(1) *Constant Cut Point*. This method consists of imposing a threshold on the number of recovered links [Antoniol et al. 2002; Marcus and Maletic 2003]. In this way, the top μ links of the ranked list are selected.

(2) *Variable Cut Point*. This is an extension of the previous method that consists of specifying the percentage of the links of the ranked list that have to be retrieved (*cut percentage*). In this way the cut point depends on the size of the ranked list.

Other methods use a threshold ε on a similarity measure and only the pairs of artifacts having a similarity measure greater than or equal to ε will be retrieved:

Table I. Analyzed Artifacts

Artifact Category	Number of Artifacts
Use cases	30
Interaction diagrams	20
Test cases	63
Code classes	37
Total number	150

(1) *Constant Threshold*. This is the standard method used in the literature. A widely adopted threshold is $\varepsilon = 0.70$, that for the vector space model (and LSI) approximately corresponds to a 45° angle between the corresponding vectors [Marcus and Maletic 2003].

(2) *Scale Threshold*. A threshold ε is computed as the percentage of the best similarity value between two artifacts, that is, $\varepsilon = c \cdot \mathit{MaxSimilarity}$, where $0 \leq c \leq 1$ [Antoniol et al. 2002]. In this case, the higher the value of the parameter c, the smaller the set of links returned by a query.

(3) *Variable Threshold*. This is an extension of the constant threshold approach [De Lucia et al. 2004c]. The constant threshold is projected from the interval [0, 1] into the interval [min similarity, max similarity], where min similarity and max similarity are the minimum and maximum similarity values in the ranked list.

The set of retrieved links does not in general coincide with the set of *correct* links between the artifacts in the repository. Indeed, any IR method will fail to retrieve some of the correct links, while on the other hand it will also retrieve links that are not correct. This is the reason why information retrieval based traceability recovery methods are semi-automatic methods and require the interaction of the software engineer [Antoniol et al. 2002; Huffman Hayes et al. 2003; Marcus and Maletic 2003].

In general, the performances of IR methods are measured using the following two metrics:

$$\mathit{recall} = \frac{|\mathit{correct} \cap \mathit{retrieved}|}{|\mathit{correct}|} \qquad \mathit{precision} = \frac{|\mathit{correct} \cap \mathit{retrieved}|}{|\mathit{retrieved}|}$$

Both measures have values in the interval [0, 1]. If the recall is 1, it means that all correct links have been recovered, though there could be recovered links that are not correct. If the precision is 1, it means that all recovered links are correct, though there could be correct links that were not recovered. In general, retrieving a lower number of links results in higher precision, while a higher number of retrieved links increases the recall.

We have experimented LSI as a traceability link recovery method on the software artifacts produced during different phases of a development project conducted by final year students at the University of Salerno, Italy [De Lucia et al. 2004c]. The project aimed at developing a software system implementing all the operations required to manage a medical ambulatory. Table I shows the category and the number of artifacts analyzed. The term-by-document matrix contained 865 terms and 150 documents. The rank of the matrix was equal to the number of documents.

The results of LSI were compared against a traceability matrix provided by the developers and containing 1005 correct links, to address the following issues:

—selection of the method to cut the ranked list;

—organization of the ranked list;

—size of the LSI subspace;

—scalability of the approach.

The first three issues are tightly coupled and will be discussed in Section 5.1, while the scalability of the approach will be discussed in a separate section (Section 5.2).

5.1 Selection of the Ranked List Cut Method, Organization of the Ranked List, and Size of the LSI Subspace

In the first two experiments we wanted to assess the variability of the performances of LSI with respect to the method used to cut the ranked list, the way the ranked lists are organized, and the size of the LSI subspace.

In the first experiment, each artifact was traced onto all the other artifacts in the repository. As also pointed out in Antoniol et al. [2002], recall is undefined for queries that do not have relevant documents associated. However, these queries may retrieve false positives that have to be discarded by the software engineer, thus affecting the precision. To take into account such queries in both the computation of the precision and the recall, we used the following aggregate metrics, rather than the average precision and recall[1]:

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

where $correct_i$ and $retrieved_i$ represent the number of correct and retrieved links for the ith query, respectively.

We analyzed how the values of precision and recall varied with different ranked list cut methods and different values of the size k of the LSI subspace. We discovered that in general varying the cut method and the size of the LSI subspace k only produced marginal differences. Concerning the cut method, in general threshold based methods performed slightly better (see Figure 1). On the other hand, concerning the size of the LSI subspace k we have basically achieved the same performances for values ranging from 30% up to 100% of the rank of the term-by-document matrix (between 45 and 150 concepts), as shown in Figure 2 for the variable threshold method. The performances are still similar when the value of k is 30 (20% of concepts), while they decrease when the value of k is 15 (10% of the concepts). This is probably due to the fact that some important information are lost when going below a given threshold. The highest precision with 100% of recall (this is the case where no correct links are missed) was achieved when the size of the LSI subspace was 20% of the rank

[1] A deep discussion about when using aggregate or average precision and recall metrics is reported in Zimmerman et al. [2005].

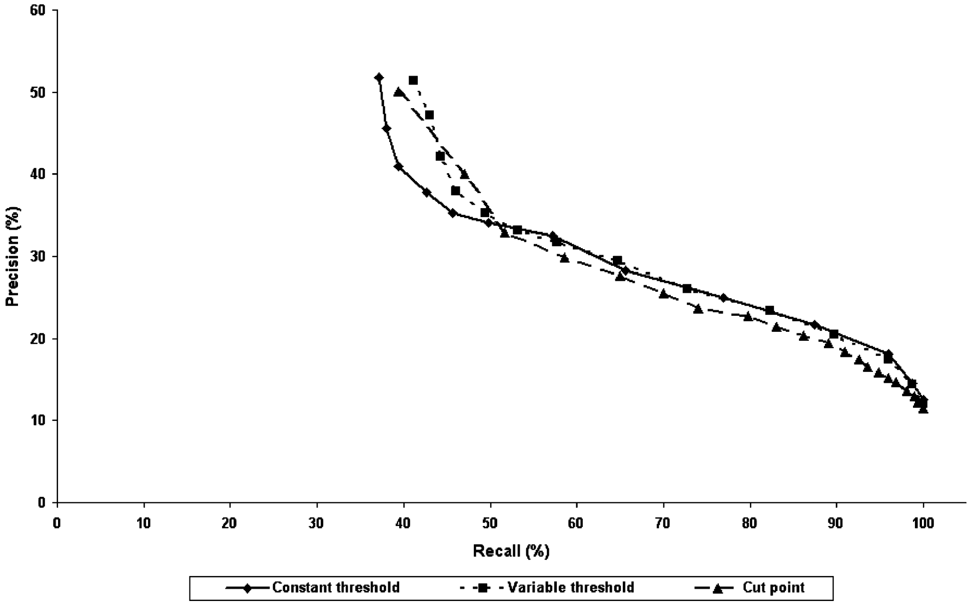


Fig. 1. Precision/recall without categorization.

of the term-by-document matrix (30 concepts), although the differences with other values of *k* are practically irrelevant at this point (see Figure 2).

In the second experiment, we analyzed whether the different structure and verbosity of different types of software artifacts influenced the retrieval performances. Indeed, in the first experiment we observed that artifacts of the same type have generally high similarity, even if they are not relevant to each other. For example, if a use case description is used as query (e.g., *insert_customer*), the related artifacts belonging to other categories will likely have a similarity lower than less relevant use case descriptions (e.g, *insert_ product*). For this reason, in the second experiment we separated the artifacts in different collections, one for each artifact type or category. Each query was then performed against each artifact subspace, thus achieving different ranked lists (we call *categorization* such kind of querying). As the size of the ranked lists is different, a variable cut point method should be preferred to a fixed cut point method. Also, note that unlike the scale and variable threshold methods, the constant threshold method does not achieve any benefit from categorization, as it does not take into account the differences in the minimum and maximum similarity values when applied to the different ranked lists.

While the variable threshold method performed only slightly better than the variable cut point and scale threshold methods (see Figure 3), the advantages of categorization in terms of precision/recall were evident for all three methods. Figure 4 compares the results achieved by the variable threshold method with and without categorization (using as size of the LSI subspace 40% of the rank of the term-by-document matrix). With categorization we achieved a precision of 35% at 80% of recall and a precision of 46% at 70% of recall, while without

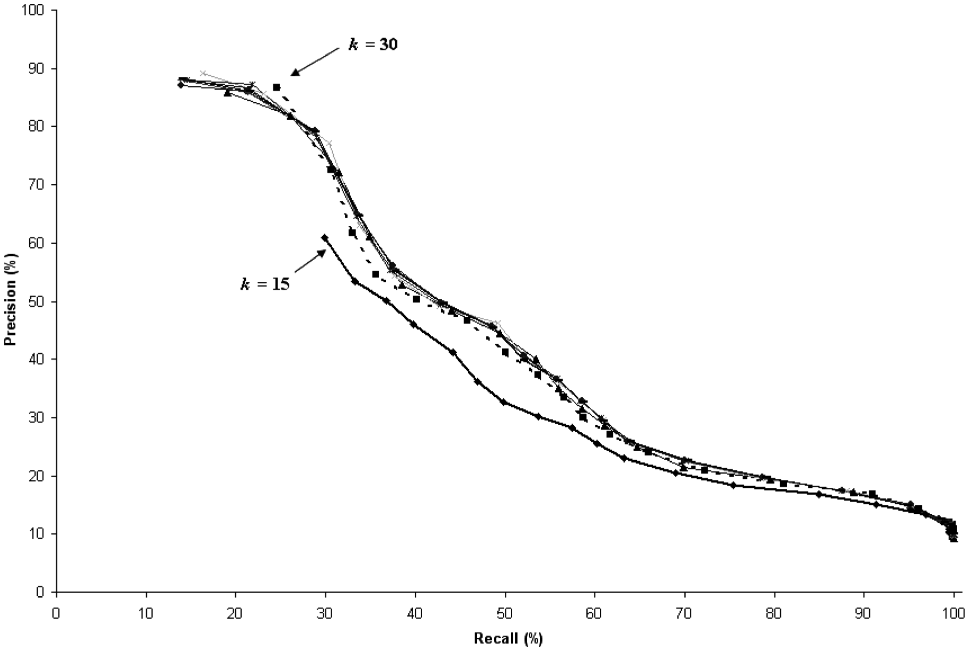


Fig. 2. Variable threshold performances with different sizes of the LSI subspace.

categorization we achieved a precision of 20% at 78% of recall and a precision of 22% at 70% of recall. As well as in the first experiment, when using categorization the size of the LSI subspace *k* only marginally affected the retrieval performances, whatever the cut method used. In particular, the behavior is very similar to what is shown in Figure 2. However, when using categorization, the higher precision with 100% of recall was achieved when the size of the LSI subspace *k* was 10% of the rank of the term-by-document matrix (15 concepts) although in general the curve for this value of *k* was below the others (similarly to Figure 2).

Due to the results of these two experiments, we decided to use categorization and the variable threshold method in the implementation of the traceability recovery tool in ADAMS. The choice of the variable threshold method is not only due to the (marginally) better performances achieved. We have chosen a similarity threshold method rather than a cut point method, as it gives the software engineer the idea of similarity between pairs of artifacts. Moreover, we preferred the variable threshold method to the scale threshold method, as it does not require the overhead to set the scale parameter (in some way this is done automatically). However, with the first two experiments, we also noticed that the advantage of the variable threshold method (i.e., the fact that the same threshold is projected on different variable thresholds depending on the minimum and maximum similarity values in the ranked list) has also a drawback when combined with categorization. In our experiments, each link in a ranked list had one of the artifacts of a given type as target artifact and all links had the same source artifact (the artifact used as query). As a

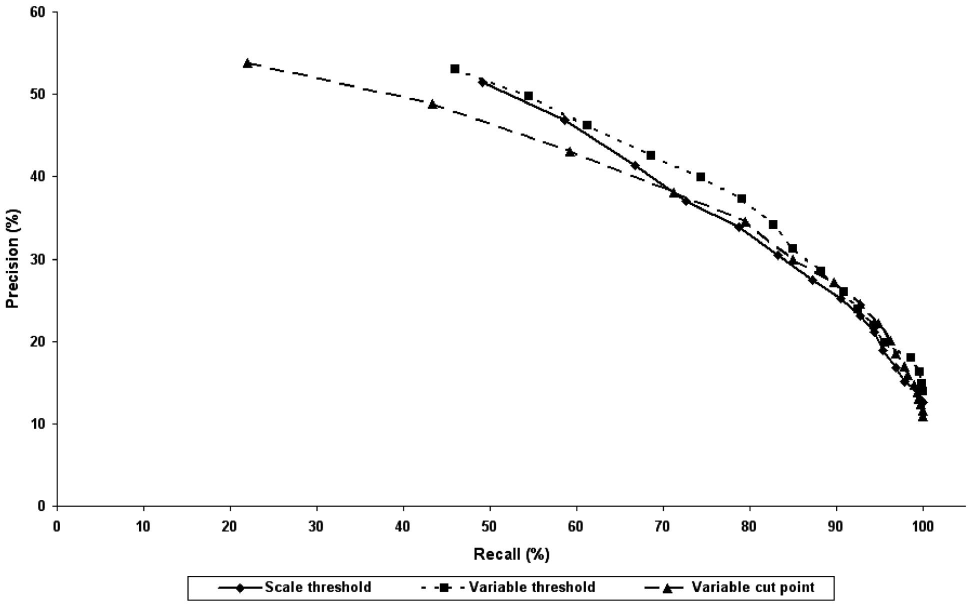


Fig. 3. Precision/recall with categorization.

consequence, the same link appeared in two different ranked lists with each of the two artifacts used once as source and once as target artifact, respectively. Although the similarity value between the two artifacts was the same in both ranked lists (independently of the direction of the query), the retrieval of the corresponding link depended on how the same threshold was projected into the minimum and maximum similarity intervals of the two ranked lists. This problem can be overcome if a single ranked list is constructed for all pairs of artifacts of two types in the repository. In this way the minimum and maximum similarity values of the ranked list in which a link is scored do not depend on the specific subsets of source and target artifacts the software engineer is interested in querying (and tracing). The traceability recovery tool might just filter the links between the selected source and target artifacts.

Concerning the size k of the LSI subspace, we cannot draw from these experiments a final conclusion. Indeed, we observed that the gap of the precision/recall curves decreases as much as the recall approaches 100%, whatever the value of k used. This means that rather than using the value that maximizes the precision when the recall is 100%, we should use the values that produce globally better precision/recall curves. We have noticed that for the software repository used in our experiments, we achieved basically the same performances with values of k ranging from 30% up to 100% of the rank of the term-by-document matrix.[2] However, we think that the reason for this behavior is due to the limited number of artifacts, compared to the results of experiments conducted

[2] These findings are also confirmed by the analysis of the Average Harmonic Mean of precision and recall that helps to understand the compromise between precision and recall [Baeza-Yates and Ribeiro-Neto 1999].

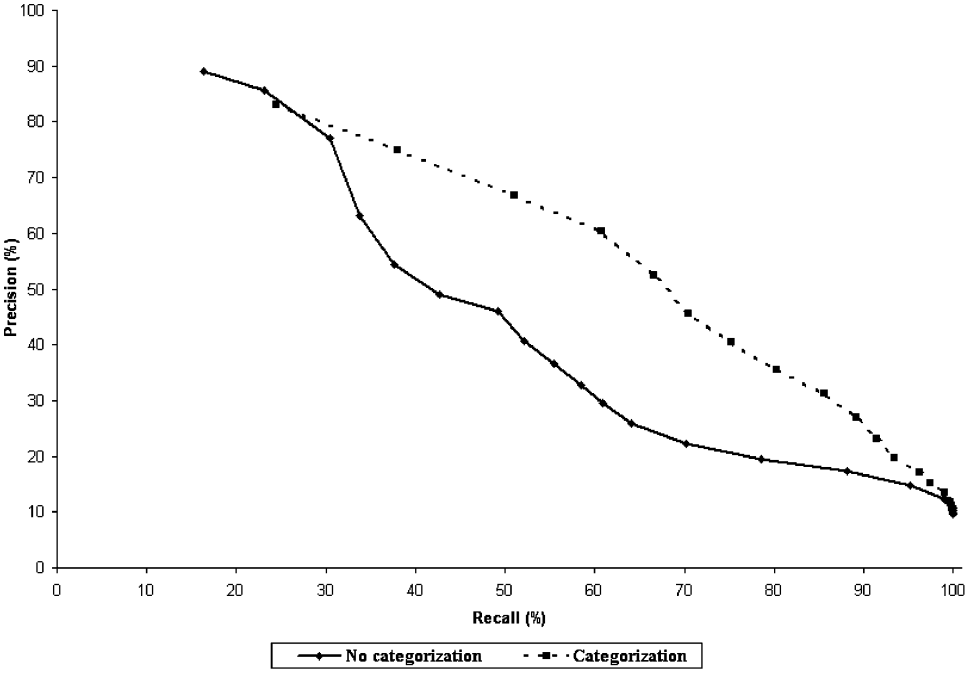


Fig. 4. Improvement of categorization with variable threshold method.

in the information retrieval field [Deerwester et al. 1990; Dumais 1992]. It is likely that using different percentages of the rank of the term-by-document matrix could give better results for larger repositories. For this reason, we decided to leave it as a configuration parameter in the traceability recovery tool.

5.2 Scalability of the Approach and Incremental Traceability Recovery

We performed a third experiment on the software artifacts of the project above based on the findings of Section 5.1. In particular, we analyzed the results of recovering traceability links between all different pairs of artifact types, by using the variable threshold method and categorization. Moreover, we used 40% of the rank of the term-by-document matrix as size of the LSI subspace (about 60 concepts), as it was a good compromise between reducing the number of concepts and achieving good retrieval performances.

Table II shows the number of possible links and correct links for all pairs of artifact types, while Figure 5 shows the precision/recall curves achieved with LSI. On average, the results are comparable with the results achieved by other authors [Antoniol et al. 2002; Cleland-Huang et al. 2005; Huffman Hayes et al. 2003; 2006; Marcus and Maletic 2003; Settimi et al. 2004], that is, at 80% of recall, precision ranges between 20% (tracing use cases onto code classes) and 50% (interaction diagrams onto test cases). The only exception is given by tracing code classes onto code classes: in our opinion, IR methods are not adequate for this purpose and should necessarily be combined with structural

Table II. Tracing Statistics

Source Category	Target Category	Possible Links	Correct Links
Use case	Use case	870	86
Use case	Interaction diagram	600	26
Use case	Test case	1890	63
Use case	Code class	1110	93
Interaction diagram	Interaction diagram	380	35
Interaction diagram	Test case	1260	83
Interaction diagram	Code class	740	69
Test case	Test case	3906	289
Test case	Code class	2331	204
Code class	Code class	1332	57
Total number of correct links			1005

or syntactic based techniques [Caprile and Tonella 1999; Maletic and Marcus 2001; Settimi et al. 2004].

To make a discussion about the scalability of LSI (and in general IR methods), in the following we show some of the tables with the details of the retrieval performances, in particular concerning tracing use cases onto code classes (among the worst results) and interaction diagrams onto test cases (among the best results). Full data of the results of this experiment can be found in the Appendix A of De Lucia et al. [2005c]. Table III shows how the performances of tracing use cases onto code classes (from column 2 to column 5) and interaction diagrams onto test cases (from column 6 to column 9) change with the threshold (column 1). In both cases, we show the number of correct links and false positives retrieved with a given threshold value, as well as the precision and recall values.

If the goal of traceability recovery is to achieve 100% of recall, we need to analyze 1013 links to trace only 93 correct links when tracing use cases onto code classes, while we need to analyze 452 links to trace 83 correct links when tracing interaction diagrams onto test cases. With such a low precision traceability link recovery becomes a tedious task, as the software engineer has to spend much more time to discard false positives than to trace correct links; this is better highlighted by the trends of correct links and false positives in Figure 6. It is important to note that these considerations are made using a medium size software system with a software repository of hundreds of artifacts. With much larger repositories, recovering all correct links using IR methods might be impractical.

A good compromise for using IR methods in practice would be getting on average a good recall (typically at least 70%), with an acceptable precision (typically at least 30%).[3] Of course, reducing the cost of discarding false positives has a drawback of missing a portion of correct links. While this can be a problem if the traceability links have to be recovered at the end of a project, it still might improve the performances of manual tracing and more traditional tools, when used during software development. Indeed, our goal is to evaluate

[3] Authors of other IR-based traceability recovery methods discussed in Section 2 also make similar considerations.

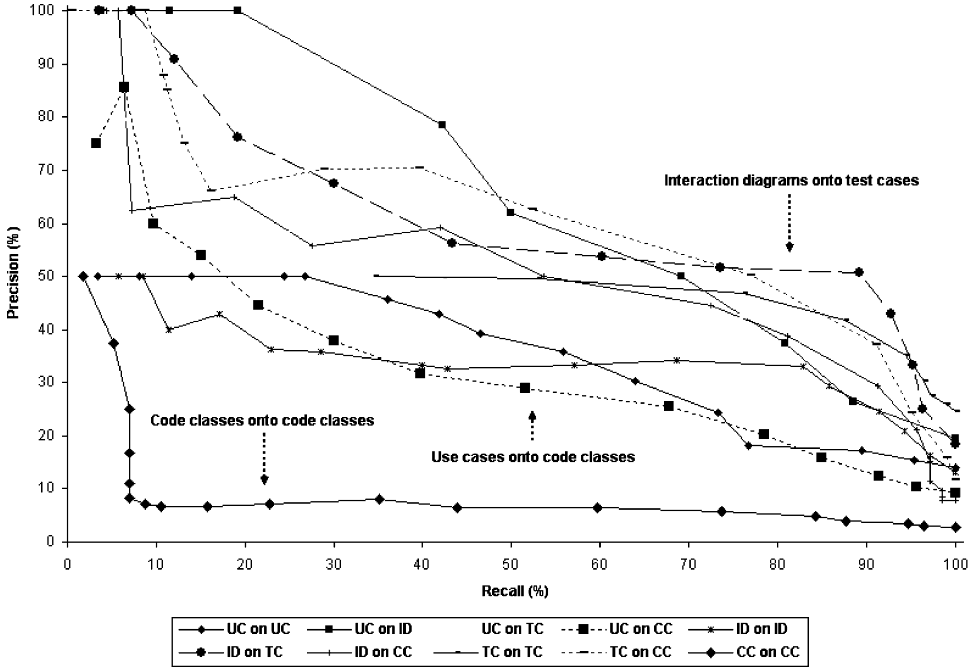


Fig. 5. Retrieval performances for pairs of artifact categories.

the benefits of IR-based traceability recovery, while being aware of their limitations.

Unfortunately, such a reasonable balance between precision and recall is not always achievable. For example, we get better results than average when tracing interaction diagrams onto test cases (a precision of 33% at 95% of recall with a threshold of 0.45) and worse results when tracing use cases onto code classes (a precision of 29% at 52% of recall with a threshold of 0.60), probably due to the higher distance between the two types of artifacts. However, in both cases, we can observe that the thresholds we are referring to are very close to the thresholds where the numbers of correct links and false positives start to diverge (see Figure 6).

Another problem that emerges from the comparison of the two cases in Table III is the fact that it is not so easy to identify the "optimal" threshold to use to achieve such a reasonable compromise between precision and recall. It is worth noting that this does not depend on the fact that we use a variable threshold as method to cut the ranked list; rather, with constant threshold the problem is even more evident, as this method does not take into account the variability in the distance and verbosity between different software artifact types. As a matter of fact, Table IV presents the results achieved tracing use cases onto code classes and interaction diagrams onto test cases using the constant threshold as method to cut the ranked list (see the Appendix A in De Lucia et al. [2005c] for the full data). While in the literature a good and widely used (constant) threshold is 0.70 Marcus and Maletic [2003], in our case study

Table III. Detailed Performances of LSI using Variable Threshold

	Use Cases onto Code Classes				Interaction Diagrams onto Test Cases			
	Retrieved Links				Retrieved Links			
Threshold	Correct Links	False Positives	Precision (%)	Recall (%)	Correct Positives	False Positives	Precision (%)	Recall (%)
0.95	3	1	75.00	3.23	3	0	100.00	3.61
0.90	6	1	85.71	6.45	6	0	100.00	7.23
0.85	9	6	60.00	9.68	10	1	90.91	12.05
0.80	14	12	53.85	15.05	16	5	76.19	19.28
0.75	20	25	44.44	21.51	25	12	67.57	30.12
0.70	28	46	37.84	30.11	36	28	56.25	43.37
0.65	37	80	31.62	39.78	50	43	53.76	60.24
0.60	48	118	28.92	51.61	61	57	51.69	73.49
0.55	63	184	25.51	67.74	74	72	50.68	89.16
0.50	73	287	20.28	78.49	77	102	43.02	92.77
0.45	79	422	15.77	84.95	79	158	33.33	95.18
0.40	85	600	12.41	91.40	80	240	25.00	96.39
0.35	89	770	10.36	95.70	83	369	18.36	100.00
0.30	93	920	9.18	100.00	–	–	–	–

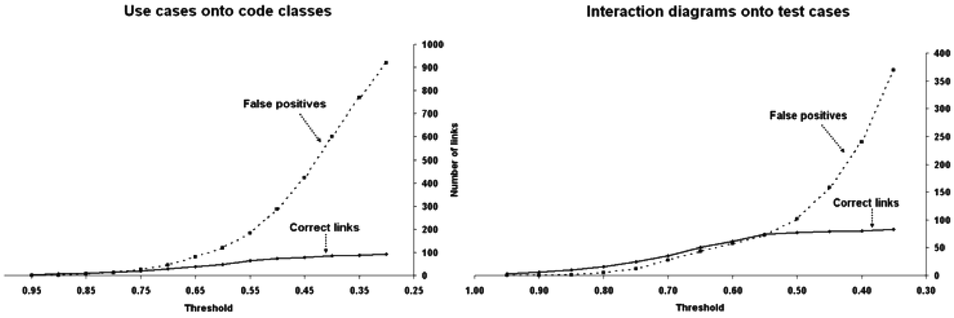


Fig. 6. Trend of number of correct links and false positives.

such a threshold does not allow to recover any link. A good compromise between precision and recall is achieved using 0.30 as threshold when tracing use cases onto code classes (a precision of 28% at 53% of recall) and 0.20 as threshold when tracing interaction diagrams onto test cases (a precision of 33% at 95% of recall).

All these results confirm the conjecture that identifying the "optimal" threshold a priori is not easy. The software engineer should make different trials to approximate it. During each trial, he/she should analyze the retrieved links, trace the correct links, discard the false positives, and possibly decrease the threshold in case the precision is still good (i.e., the effort to discard false positives is still acceptable). For this reason, we propose an incremental approach to the traceability recovery problem, so that the links proposed by the tool can be analyzed and classified step-by-step. The process should start with a high threshold that is decreased at each iteration. In this way, the software engineer is able to approximate the "optimal" threshold as soon as the effort to discard false positives becomes too high. Table V shows the results of applying

Table IV. Detailed Performances of LSI using Constant Threshold

	Use Cases onto Code Classes				Interaction Diagrams onto Test Cases			
	Retrieved Links				Retrieved Links			
Threshold	Correct Links	False Positives	Precision (%)	Recall (%)	Correct Links	False Positives	Precision (%)	Recall (%)
0.50	3	1	75.00	3.23	0	0	–	–
0.45	8	4	66.67	8.60	0	0	–	–
0.40	20	18	52.63	21.51	3	0	100.00	3.61
0.35	32	57	35.96	34.41	12	3	80.00	14.46
0.30	49	129	27.53	52.69	36	27	57.14	43.37
0.25	73	287	20.28	78.49	68	64	51.52	81.93
0.20	85	554	13.30	91.40	79	159	33.19	95.18
0.15	91	870	9.47	97.85	83	426	16.31	100.00
0.10	93	1105	7.76	100.00	–	–	–	–

Table V. Detailed Performances of LSI using the Incremental Approach

	Use Cases onto Code Classes			Interaction Diagrams onto Test Cases		
	Retrieved Links			Retrieved Links		
Threshold	Correct Links	False Positives	Partial Precision (%)	Correct Links	False Positives	Partial Precision (%)
0.95	3	1	75.00	3	0	100.00
0.90	3	0	100.00	3	0	100.00
0.85	3	5	37.50	4	1	80.00
0.80	5	6	45.45	6	4	60.00
0.75	6	13	31.58	9	7	56.25
0.70	8	21	27.59	11	16	40.74
0.65	9	34	20.93	14	15	48.28
0.60	11	38	22.45	11	14	44.00
0.55	15	66	18.52	13	15	46.43
0.50	10	103	8.85	3	30	9.09
0.45	6	135	4.26	2	56	3.45
0.40	6	178	3.26	1	82	1.20
0.35	4	170	2.30	3	129	2.27
0.30	4	150	2.60	–	–	–

the incremental approach to two traceability recovery tasks, namely use cases onto code classes tracing and interaction diagrams onto test cases tracing, respectively (see the Appendix A in De Lucia et al. [2005c] for full data): for each threshold the table shows the number of new correct links and false positives the software engineer has to classify at each iteration (supposing he/she is able to correctly classify all links), as well as the partial precision of the iteration. We are assuming that the traceability recovery tool maintains knowledge about the classification actions performed by the software engineer, thus showing at each iteration only the new traceability links retrieved.

From the results shown in Table V, it is possible to see that using an incremental traceability recovery approach the software engineer would be able to decide whether the effort to discard false positives is becoming too high and therefore stop the process. For example, in the case of tracing use cases onto code classes, it might be reasonable to proceed until the threshold value 0.55 or 0.50, since the false positives to discard are becoming much higher than

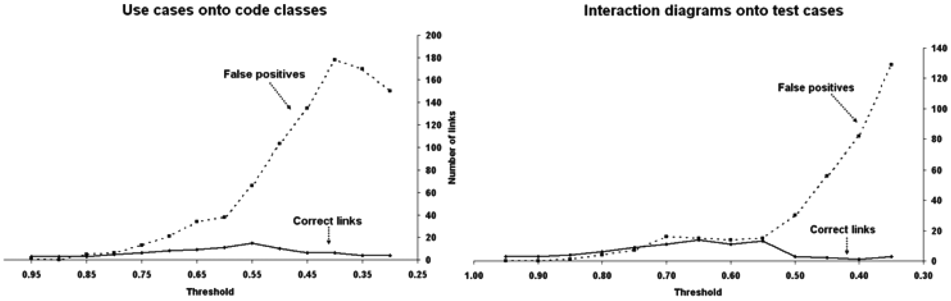


Fig. 7. Trend of number of correct links and false positives with incremental process.

with previous thresholds. In the case of tracing interaction diagrams onto test cases it might be reasonable to proceed until the threshold value 0.50 or 0.45. However, the incremental tuning of the threshold can only give the perception of how the classification effort is increasing: this perception might actually be different for different software engineers who may decide to stop the process at different thresholds.

Figure 7 shows the trend of correct links and false positives classified at each threshold value using the incremental approach. This trend is not very different than the trend shown in Figure 6 without an incremental classification of the links (one-shot approach). In particular, the points where the curves start to sensibly diverge in Figures 6 and 7 are very close. The advantage with the incremental approach is that in this case the software engineer is able to identify this point as soon as he/she realises that the effort required to discard false positives is becoming much higher than the effort required to trace correct links.

The same trends of correct links and false positives depicted in Figures 6 and 7 can also be observed using larger artifact repositories. Figures 8 and 9 show the trends of the numbers of correct links and false positives using both one-shot and incremental approaches when tracing 87 use cases onto 92 code classes of ADAMS and 88 manual pages onto 219 code classes of release 3.4 of LEDA (Library of Efficient Data structures and Algorithms).[4] The term-by-document matrix of the ADAMS system contains 309 documents (including use cases, Java Server Pages, servlets, and code classes) and 3763 terms. Concerning the size of the LSI subspace, we observed for ADAMS a behavior very similar to what is depicted in Figure 2 for the smaller student system. For this reason, we also used 40% of the documents as size of the LSI subspace (123 concepts) in the experiment with ADAMS. The data of using LSI on LEDA are taken from Marcus and Maletic [2003]: the term-by-document matrix contains 803 documents (including manual sections and source code files) and 3814 terms and the size of the LSI subspace was between 25% and 50% of the documents; in this case a fixed cut point was used to cut the ranked lists [Marcus and Maletic 2003].

[4] LEDA (available from http://www.algorithmic-solutions.com/endownloads.htm) is a well known library developed and distributed by Max Planck Institut für Informatik, Saarbrücken, Germany (and lately by Algorithmic Solutions Software GmbH).

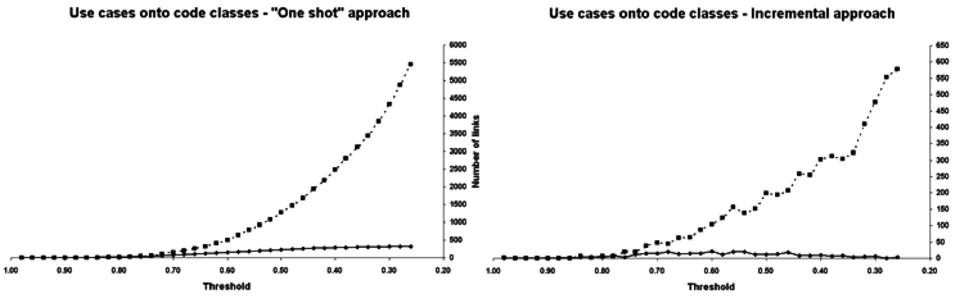


Fig. 8. Trend of number of correct links and false positives for ADAMS.

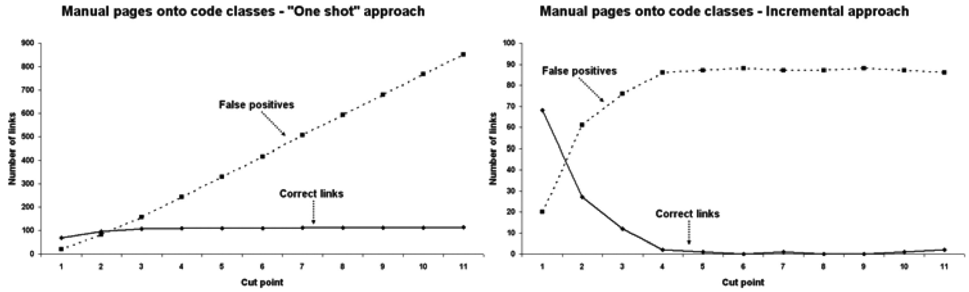


Fig. 9. Trend of number of correct links and false positives for LEDA.

It is important to note that the differences between the results of these two case studies are considerable. Indeed, the results of the ADAMS system are a bit worst but still comparable with the results achieved in the smaller case study discussed above (see use cases onto code classes tracing in Figures 6 and 7 and compare with Figure 8): a good threshold to stop the incremental process is between 0.60 and 0.50. Much better results are achieved tracing manual pages onto code classes of LEDA as shown in Figure 9. In this case, the first three iterations are enough to identify almost all correct links, while in the remaining iterations the retrieved links are almost all false positives. This is the reason why the curve of false positives has a linear trend in the one shot approach and a constant trend in the incremental approach when the cut point is greater than three (with a fixed cut point a constant number of links is retrieved after each increment of the cut point). A reason for this good (but rather unusual) trend is the fact that the pages of the LEDA manual are generated directly from the source code and include sections of code and comments. ADAMS and LEDA represent two very different cases of software systems where traceability recovery can be applied. Probably, ADAMS is a more realistic case than LEDA (that has nevertheless been used as a benchmark to compare IR-based traceability recovery methods [Antoniol et al. 2002; Marcus and Maletic 2003]), but both cases confirm that the considerations made for the student project discussed above are still valid for larger artifact repositories.

6. IMPLEMENTATION OF THE LSI-BASED TRACEABILITY RECOVERY TOOL

In this section, we describe the architecture and the functionalities of the LSI-based traceability recovery tool integrated in ADAMS. We exploited the results outlined by the case studies discussed in Section 5 while making design decisions concerning, in particular, the support for incremental traceability recovery and the use of the variable threshold method and categorization. Concerning the variable threshold method, rather than projecting the threshold in the interval [min similarity, max similarity], as discussed in Section 4, we decided to adopt the inverse (and equivalent) approach, that is, projecting the similarity values (cosine of the angle between two vectors) from the interval [min similarity, max similarity] to the interval [0, 1]. Although it is computationally more expensive, this approach hides the implementation details of variable threshold and provides the software engineer with relative rather than absolute similarity values that can be directly compared with the selected threshold and are less dependent on the variability in the verbosity of the artifacts in the repository.

We can also observe that using this approach the projected similarity values (rather than the threshold) can change depending on the minimum and maximum similarity values in the ranked list. As discussed in Section 4, this problem can be avoided if the tool always computes the similarity between all the artifacts of two types, rather than only the similarity between the pairs of artifacts the software engineer is interested in querying. Another observation that can be made is that during software development the minimum and maximum similarity values can still change due to fact that the artifact repository changes, although these values tend to become stable as more artifacts are added to the repository. However, this is not the main reason of variability of the relative (projected) similarity values, as changes to the repository also result in changes to the absolute similarity values of unchanged artifacts. For example, in the vector space model (and LSI) when adding an artifact to the repository, the term-by-document matrix changes: besides adding a new column for the artifact and new rows for the new terms introduced by the artifact, the global weights of the terms contained in both the new artifact and previous artifacts changes too. For this reason, the cosine of the angle between the vectors of unchanged artifacts might change too. Furthermore, when changing the term-by-document matrix, the singular value decomposition changes and this can result in changes in the document vectors in the LSI subspace. It is worth noting that, in our experience discussed in Section 7, we have observed a very marginal variability in the projected similarity values as a consequence of all these factors.

6.1 Architecture of the Traceability Recovery Tool

As well as ADAMS, the traceability recovery tool has been realized using web technologies, in particular Java Server Pages (JSP) and Servlets. The web server is Apache Tomcat 5.5, while the Database Management System is MySql 5.0. The tool has been integrated by developing three new modules, namely *Indexer*, *LSI*, and *Query*, in the artifact management subsystem of ADAMS:

—*Indexer*: When an artifact is checked-in this module builds a hash table containing the number of occurrences of the different terms in the artifact (when an artifact is deleted its table is deleted too). This module is also able to extract the text from a wide variety of file types. For this reason, the artifacts have not to be entered in ADAMS in a proprietary format. The extraction of the terms is preceded by a text normalization phase performed in three steps:

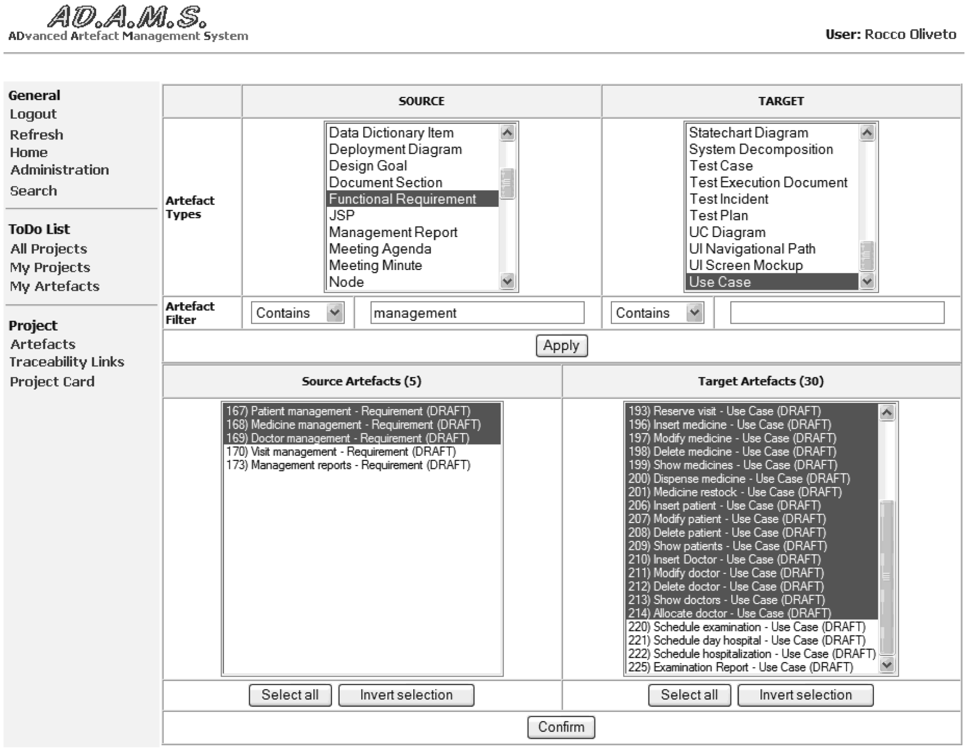
(1) White spaces in the text are normalized and most nontextual tokens from the text are eliminated (i.e., operators, special symbols, numbers, etc.);
(2) Identifiers that are composed of two or more words are split into separate words (i.e., TelephoneNumber and telephone_number are split into the words telephone and number);
(3) All the terms in the stop word list or with a length less than three are removed.

Finally, the indexer is implemented as a separate thread invoked at check-in time to avoid slowing-down the work of the software engineer.

—*LSI*: This module first computes the weights of the term-by-document matrix from the number of occurrences of the terms in the artifacts; then it computes the singular value decomposition of the term-by-document matrix[5] and the document vectors in the reduced k-space of concepts; finally, it builds the ranked lists of similarity values for all pairs of artifact types, by projecting the cosine of the angle between each pair of artifact vectors from the interval [min similarity, max similarity] into the interval [0, 1]. These ranked lists are stored in the ADAMS repository and used by the module *Query*. Concerning the size of the LSI subspace, we have used 40% of the concepts (rank of the original term-by-document matrix) as default percentage value, but we left this as a configuration parameter of ADAMS, that can be changed for repositories of different size. It is important to note that this module is the most expensive from a computation time point of view (it takes about 23 seconds on a repository of 1500 artifacts and about 5500 terms; further details can be found in the Appendix B of De Lucia et al. [2005c]). For this reason, it is invoked periodically (typically daily and at night) and only in case artifacts have been checked-in or deleted since last execution. The time interval is a configuration parameter of ADAMS.

—*Query*. This module enables the software engineer to choose the subsets of source and target artifacts on which he/she wants to work for traceability recovery. The software engineer can select the types and filter on the names of source and target artifacts, and finally select the artifacts he/she is interested in (see Figure 10). Once the source and target artifacts have been selected, the software engineer can choose a threshold and the module compares the links retrieved by using LSI (whose similarity values are greater or equal to the threshold) with the links traced by the software engineer. The module

[5] We used the SVD package developed in JAVA and kindly provided by Prof. Mike Berry of the University of Tennessee (http://www.cs.utk.edu/∼lsi).



	SOURCE	TARGET
Artefact Types	Data Dictionary Item Deployment Diagram Design Goal Document Section Functional Requirement JSP Management Report Meeting Agenda Meeting Minute Node	Statechart Diagram System Decomposition Test Case Test Execution Document Test Incident Test Plan UC Diagram UI Navigational Path UI Screen Mockup Use Case
Artefact Filter	Contains management	Contains

Source Artefacts (5)	Target Artefacts (30)
167) Patient management - Requirement (DRAFT) 168) Medicine management - Requirement (DRAFT) 169) Doctor management - Requirement (DRAFT) 170) Visit management - Requirement (DRAFT) 173) Management reports - Requirement (DRAFT)	193) Reserve visit - Use Case (DRAFT) 196) Insert medicine - Use Case (DRAFT) 197) Modify medicine - Use Case (DRAFT) 198) Delete medicine - Use Case (DRAFT) 199) Show medicines - Use Case (DRAFT) 200) Dispense medicine - Use Case (DRAFT) 201) Medicine restock - Use Case (DRAFT) 206) Insert patient - Use Case (DRAFT) 207) Modify patient - Use Case (DRAFT) 208) Delete patient - Use Case (DRAFT) 209) Show patients - Use Case (DRAFT) 210) Insert Doctor - Use Case (DRAFT) 211) Modify doctor - Use Case (DRAFT) 212) Delete doctor - Use Case (DRAFT) 213) Show doctors - Use Case (DRAFT) 214) Allocate doctor - Use Case (DRAFT) 220) Schedule examination - Use Case (DRAFT) 221) Schedule day hospital - Use Case (DRAFT) 222) Schedule hospitalization - Use Case (DRAFT) 225) Examination Report - Use Case (DRAFT)

Fig. 10. Selection of source and target artifacts for traceability recovery.

Query initially proposes 95% as a default threshold to cut the ranked list. This threshold can be decreased and tuned within an iterative process until the software engineer decides that it is not worth to proceed further, because the effort required to discard false positives is becoming too high. This issue is discussed in more details in the following subsections.

6.2 Tool Functionalities

As discussed in Section 5, IR methods cannot completely automate the traceability recovery process and therefore we expect that the software engineer does not completely rely on the traceability recovery tool, but during the software process he/she combines manual tracing with the tool support. In the following, we denote the set of links retrieved by the tool with the term, *retrieved*, and the set of links traced by the software engineer with the term *traced*. We also define the following sets:

—TracingAgreement = $traced \cap retrieved$

—NonTracingAgreement = $\overline{traced} \cap \overline{retrieved}$

—SuggestedLinks = $\overline{traced} \cap retrieved$

—WarningLinks = $traced \cap \overline{retrieved}$, where $\bar{A}$ denotes the complementary set of A. The first two sets contain the links retrieved and excluded, respectively, by both the software engineer and the tool. The set *SuggestedLinks* contains the links retrieved by the tool but not traced by the software engineer, while*WarningLinks* contains the links traced by the software engineer and missed by the tool. The software engineer has to investigate the suggested links to discover new traceability links, thus enriching the set of traced links. 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 quality problems in the text description of the traced artifacts.

During the incremental traceability recovery process the software engineer needs to classify false positive links suggested by the tool to avoid that these links are suggested again in following iterations. Once classified as false positives, these links are moved from the set *SuggestedLinks* to the set *FalsePositives*, so that the former set can be redefined as

$$\text{SuggestedLinks} = \overline{traced} \cap retrieved \cap \overline{FalsePositives}.$$

Similarly, the software engineer might decide that the quality of artifacts involved in a warning link is acceptable despite the fact that their similarity is low. Such a link can then be accepted as a false negative and move from the set *WarningLinks* to the set *FalseNegatives*. Having introduced the latter set, *WarningLinks* can be redefined as:

$$\text{WarningLinks} = traced \cap \overline{retrieved} \cap \overline{FalseNegatives}.$$

At any time, each potential link between two artifacts can belong to one of the sets defined above and move from a set to another as shown in Figure 11. Figure 12 shows a screenshot of the traceability recovery tool. The artifacts appearing as source or target of a suggested link (or false positive) in Figure 12 can be downloaded by clicking on the artifact name and analyzed. In this way, the software engineer can recognize whether a link in the set *SuggestedLinks* is a correct link or a false positive. In the first case he/she can trace the link (that will move to the set *TracingAgreement*), while in the second case the link can be discarded and classified as false positive (thus moving to the set *FalsePositives*). The software engineer can also revise his/her decision and trace a link retrieved by the tool that was previously classified as false positive; in this case the link moves to the set *TracingAgreement* (see Figure 11). In addition, a link in the set *TracingAgreement* can move to the set *SuggestedLinks* in case the software engineer removes the traceability link.

At each iteration of the incremental traceability recovery process, the tool shows the precision of the previous iteration in terms of the number of traced links with respect to the number of suggested links (see Figure 12). This is useful for the software engineer to decide if it is the case of stopping the threshold tuning. Moreover, the tool maintains for each pair of artifact types the lowest threshold used by software engineer to trace some suggested links. It is worth noting that the lowest threshold used by the software engineer during

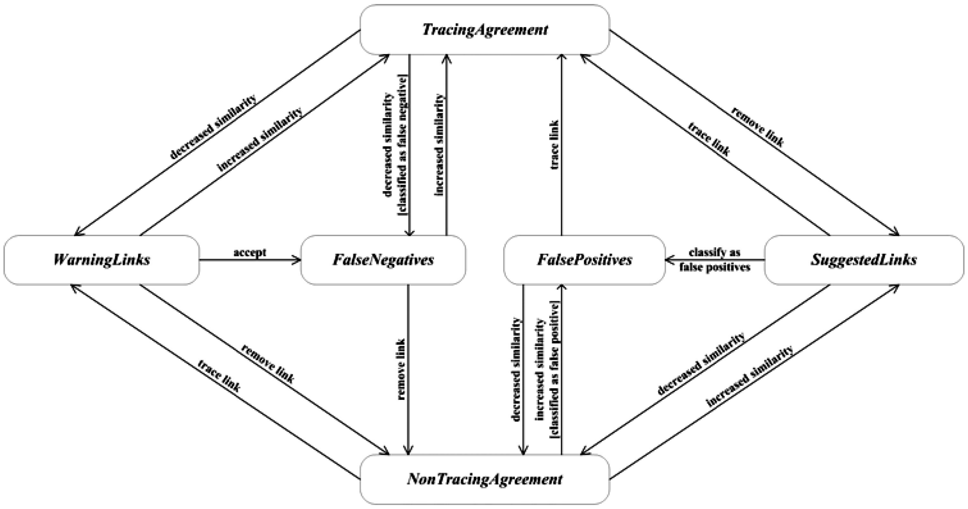
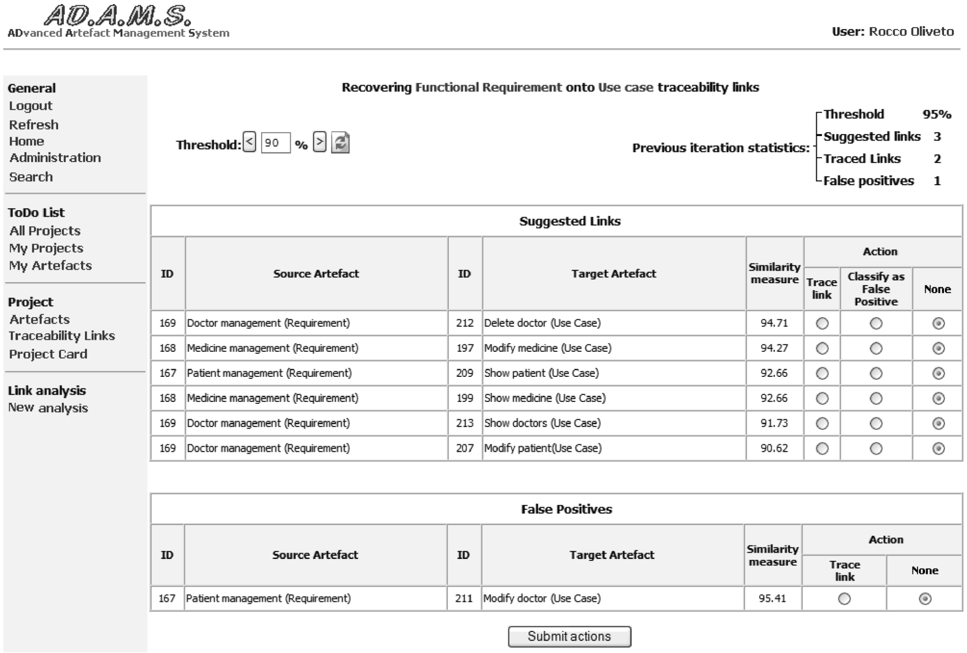


Fig. 11. Traceability link transition graph.



User: Rocco Oliveto

Recovering Functional Requirement onto Use case traceability links

Threshold: < 90 % >

Previous iteration statistics:

Threshold	95%
Suggested links	3
Traced Links	2
False positives	1

Suggested Links

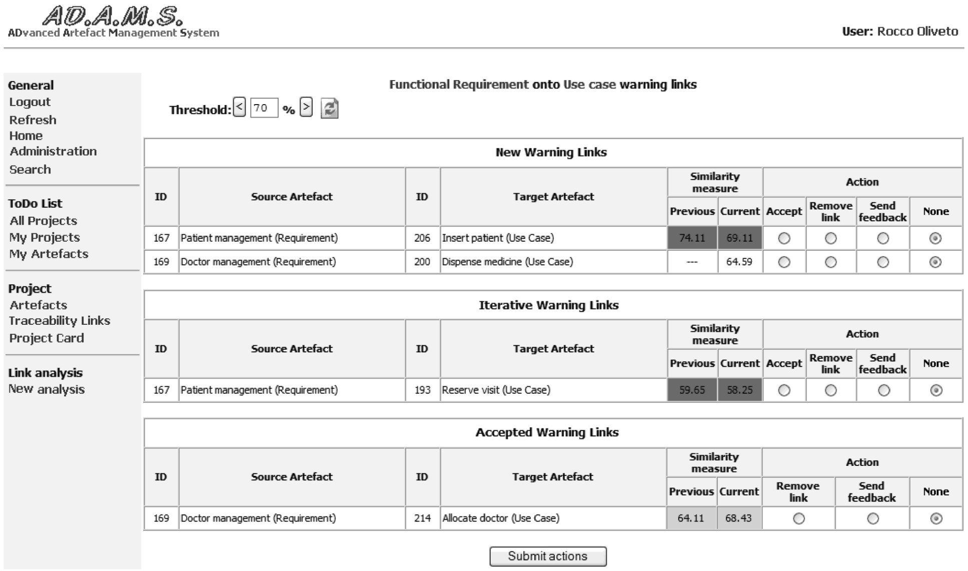
ID	Source Artefact	ID	Target Artefact	Similarity measure	Action		
					Trace link	Classify as False Positive	None
169	Doctor management (Requirement)	212	Delete doctor (Use Case)	94.71	○	○	●
168	Medicine management (Requirement)	197	Modify medicine (Use Case)	94.27	○	○	●
167	Patient management (Requirement)	209	Show patient (Use Case)	92.66	○	○	●
168	Medicine management (Requirement)	199	Show medicine (Use Case)	92.66	○	○	●
169	Doctor management (Requirement)	213	Show doctors (Use Case)	91.73	○	○	●
169	Doctor management (Requirement)	207	Modify patient(Use Case)	90.62	○	○	●

False Positives

ID	Source Artefact	ID	Target Artefact	Similarity measure	Action	
					Trace link	None
167	Patient management (Requirement)	211	Modify doctor (Use Case)	95.41	○	●

Fig. 12. Analysis of suggested links and false positives.

the incremental traceability recovery process is likely to approximate the "optimal" threshold as discussed in Section 5. Such a threshold rather than the default 95% threshold is proposed at the first iteration of the next traceability recovery session on the same pair of artifact types, thus reducing the time required for tuning.



User: Rocco Oliveto

Functional Requirement onto Use case warning links

Threshold: 70 %

New Warning Links

ID	Source Artefact	ID	Target Artefact	Similarity measure		Action			
				Previous	Current	Accept	Remove link	Send feedback	None
167	Patient management (Requirement)	206	Insert patient (Use Case)	74.11	69.11	○	○	○	●
169	Doctor management (Requirement)	200	Dispense medicine (Use Case)	---	64.59	○	○	○	●

Iterative Warning Links

ID	Source Artefact	ID	Target Artefact	Similarity measure		Action			
				Previous	Current	Accept	Remove link	Send feedback	None
167	Patient management (Requirement)	193	Reserve visit (Use Case)	59.65	58.25	○	○	○	●

Accepted Warning Links

ID	Source Artefact	ID	Target Artefact	Similarity measure		Action		
				Previous	Current	Remove link	Send feedback	None
169	Doctor management (Requirement)	214	Allocate doctor (Use Case)	64.11	68.43	○	○	●

Fig. 13. Analysis of warning links.

As said before, a link in the set *WarningLinks* might indicate that the software engineer erroneously traced two artifacts. In this case, he/she can remove the link previously inserted that will move to the set *NonTracingAgreement* (see Figure 11). If the two artifacts were correctly traced by the software engineer, the tool indicates that there might be some quality problems in terms of text description of one or both the artifacts. In this case, the tool can be used as a support to quality control and the information about warning links can be forwarded to the quality manager. After reviewing the traced artifacts, the quality manager can decide whether sending a feedback to the artifact developers to ask for changes or accepting the link (see Figure 13). In the latter case, it is like the warning link is classified as a false positive warning link (or false negative) and in future session it will appear in the *Accepted Warning Link* section in Figure 13. This set is represented by the set *FalseNegatives* in Figure 11. As well as links in the set *WarningLinks*, links in the set *FalseNegatives* can also be removed, thus moving to the set *NonTracingAgreement*.

The number of warning links also depends on the threshold used to cut the ranked list produced by LSI. However, unlike the number of suggested links, the number of warning links increases with the threshold and is limited by the number of links traced by the software engineer. The software engineer should define a "quality" threshold that is used to decide whether a traced link has to be considered as a warning link (the links with a similarity below such a threshold). By default, the quality threshold is equal to the lowest threshold used (and considered acceptable) by the software engineer during the traceability recovery process. However, the software engineer can still tune the quality threshold; the tool maintains the last threshold used and proposes it at the beginning of future sessions.

6.3 Support during Software Evolution

Besides the transitions made as a consequence of software engineer actions, links can make other transitions in the graph in Figure 11, depending on changes made to the artifacts that result in increased or decreased similarity. In particular, decreased similarity between two artifacts can result in the transition of a traceability link from *TracingAgreement* to *WarningLinks* (or to *FalseNegatives*), from *SuggestedLinks* to *NonTracingAgreement*, or from *FalsePositives* to *NonTracingAgreement*. The first case is the most important as it indicates that the similarity of previously traced artifacts (possibly traced following the suggestion of the tool) decreased below the “quality” threshold and then the traced artifacts are worth of investigation. These links will be highlighted as new warning links in future sessions of the warning link analysis and both the current similarity (below the quality threshold) and the previous similarity (above the quality threshold) are highlighted, as shown in Figure 13. Note that new warning links in Figure 13 are links traced by the software engineer that are newly classified as a warning link and they also include new links traced by the software engineer, for example as a consequence of the addition of artifacts to the repository. In this case, the previous similarity is not shown (see Figure 13). Moving from *TracingAgreement* to *FalseNegatives* is similar, but refers to a previously accepted warning link moved to the set *TracingAgreement* as a consequence of increased similarity.

ADAMS provides support to the software engineer for the analysis of the evolution of the warning links. Indeed, as a consequence of changes requested by the quality manager and made to artifacts involved in warning links, the similarity of such artifacts might change; if it increases above the quality threshold the link moves from the set *WarningLinks* (or *FalseNegatives*) to the set *TracingAgreement* in Figure 11 and will not be displayed further as a warning link, otherwise it will be displayed in the *Iterative Warning Link* section in Figure 13. Both the current and previous similarity measures are visualized for the iterative warning links as well as for the accepted warning links (false negatives), so that the software engineer can realise whether changes to the traced artifacts have resulted in improvements of the similarity values and decide for an action to take.

Finally, other transitions can result in the graph in Figure 11 as a consequence of increased similarity from *NonTracingAgreement* to *SuggestedLinks* or to *FalsePositives* (in case the link was also previously classified as false positive). Relevant changes to the similarity values of traceability links are also notified to the software engineer, besides being visualized during the traceability recovery sessions. By default, the event management subsystem of ADAMS weekly sends a notification to the software engineer containing a summary of the artifacts added, deleted, and modified together with a list of candidate links whose similarity value is (or has increased) above the “optimal” threshold (new links suggested for tracing). In addition, this summary also contains the lists of new warning links, as well as past warning links that have increased their similarity above the “quality” threshold.[6]

[6] Actually, how often ADAMS sends the report is defined by a configuration parameter. In this way,

It is important to note that the support for evolution of traceability links provided by ADAMS has some difference and similarity with respect to other approaches discussed in Section 2 [Maletic et al. 2005; Nguyen et al. 2005; Nistor et al. 2005]. Similarly to these approaches, ADAMS is able to identify the traceability links potentially affected by changes in the involved artifacts, but this is done in a different and complementary way.

7. EXPERIENCE AND EVALUATION

In this section, we present the results of a case study concerning the use of the traceability recovery tool of ADAMS during software development. In particular, ADAMS has been experimented from April 15th to July 20th 2005 as artifact management system within the projects conducted by students of the Software Engineering courses of the Computer Science program at the University of Salerno (Italy). This experience involved about 150 students allocated in seventeen software projects aiming at developing software systems with a distributed architecture (typically three tier) using mainly Java and web technologies and a relational DBMS (typically MySQL). Each project team included between six and eight undergraduate students with development roles and two master students with roles of project and quality management, respectively. The process model adopted for software development was incremental: the students performed a complete requirement analysis and high level design of the software system to be developed and then proceeded with an incremental development of the subsystems. The goal was to release at least one increment by July 20th, 2005 (deadline for closing the projects). The project manager was responsible for coordinating the project, defining the project schedule, organizing project meetings, collecting process metrics, and allocating human resources to tasks. The quality manager was responsible for defining process and product standards of the project, collecting product metrics, and organizing checklist-based artifact reviews for quality control.

The project and quality managers were also in charge of building the traceability matrix for the software artifacts developed in their project and to this aim they were also trained on the use of the traceability recovery tool of ADAMS. In particular, we showed them how trace links without and with the tool support, and in the latter case adopting both a “one-shot” and an incremental approach. Managers performed the traceability recovery task periodically during the different phases of the project and the links traced were validated during review meetings made by the whole team together with Ph.D. students and academic researchers. The managers were also required to submit a first traceability management report (as well as project management and quality management reports) by the end of May, a second report by the end of June and a summary report by the end of the project. It was not prescribed which type of artifacts had to be traced, but this was left to the project and quality managers that had to balance the effort required for this task with the effort required for the other tasks (80 hours was the upper bound for the effort to be devoted by each team member to the project). As a result, mainly traceability links between

the software engineer can decide when he/she wants to receive the notifications.

artifacts of the Requirement Analysis Document (RAD), namely functional requirements, scenarios, use cases, and sequence diagrams, were traced in all projects, as these artifacts were available through the different project phases. In particular, the managers traced functional requirements onto scenarios, scenarios onto use cases, and use cases onto sequence diagrams. To minimize the effort, they did not insert all possible traceability links, as ADAMS exploits indirect traceability links for event notifications. In some cases (only in the projects that were not late on their schedule), the managers also traced a subset of code classes onto the use cases. Basically, due to the short time available, in the latest phases of the projects the managers preferred to pay more attention to activities such as document review, code inspection, and testing, rather than traceability management.

Table VI summarizes the statistics of the artifacts used for traceability in the three periods (15th April–31st May, 1st June–30th June, 1st July–20th July). Besides the total number of (RAD and code) artifacts considered in each period, for the second and third period, the table also shows the number of added and modified artifacts with respect to the previous period.

7.1 Results of the Experience

Within each period, the project manager and the quality manager used the traceability recovery tool to trace new suggested links and to monitor both the link evolution and the artifact quality through the analysis of the warning links. Table VII shows for each project and each period the minimum and maximum numbers of iterations performed in the different sessions of tool usage. Table VII also shows for each project and each period the minimum and maximum values for the lowest thresholds used in the iterations of the tool usage sessions. It is worth noting that the incremental approach was preferred to a "one-shot" approach in all the projects, especially in the first period when the threshold was not tuned yet. In general, the number of iterations decreased in the following periods and often the lowest threshold discovered in the first period was used as a basis to perform only one iteration. However, the data in Table VII also show cases where the software engineer adopted an incremental approach made of several iterations also in the second and third periods to better tune the threshold. These generally coincide with cases where several artifacts were added in the following periods. In this case, a high number of suggested links had to be analyzed in the second and third period, so several iterations were needed to keep under control the number of false positives to discard. As further consideration, in these cases the lowest threshold used for a pair of artifact types was not always the same as the lowest threshold identified in the previous period.

Table VII also shows the minimum and maximum quality thresholds used in all the periods: as for the lowest threshold, different quality thresholds were used for the different pairs of artifact types, although the threshold used for each pair of artifact types was almost the same for all the periods. It is worth noting that in general, the quality threshold is also very close to the "optimal" threshold and usually it is above the lowest threshold used to recover traceability

Table VI. Artifact Statistics

		1st Period	2nd Period			3rd Period		
	Artifact Types	Num. of Artifacts	Num. of Artifacts	Added Artifacts	Modified Artifacts	Num. of Artifacts	Added Artifacts	Modified Artifacts
1	RAD artifacts	58	58	0	22	67	9	28
	Code artifacts	–	–	–	–	–	–	–
2	RAD artifacts	57	76	19	37	89	13	61
	Code artifacts	–	–	–	–	–	–	–
3	RAD artifacts	38	38	0	35	46	8	37
	Code artifacts	8	8	0	8	8	0	8
4	RAD artifacts	20	59	39	18	63	4	55
	Code artifacts	–	8	8	–	9	1	3
5	RAD artifacts	–	71	71	–	92	21	55
	Code artifacts	–	–	–	–	18	18	–
6	RAD artifacts	32	87	55	17	91	4	54
	Code artifacts	–	–	–	–	–	–	–
7	RAD artifacts	65	79	14	44	85	6	48
	Code artifacts	–	–	–	–	18	18	–
8	RAD artifacts	69	–	–	–	–	–	–
	Code artifacts	7	–	–	–	–	–	–
9	RAD artifacts	149	159	10	87	159	0	103
	Code artifacts	–	–	–	–	–	–	–
10	RAD artifacts	60	72	12	19	85	13	31
	Code artifacts	–	–	–	–	22	22	–
11	RAD artifacts	26	39	13	13	39	0	24
	Code artifacts	–	–	–	–	–	–	–
12	RAD artifacts	62	64	2	36	95	31	57
	Code artifacts	8	11	3	3	11	0	5
13	RAD artifacts	36	36	0	24	43	7	24
	Code artifacts	–	–	–	–	–	–	–
14	RAD artifacts	21	68	47	13	79	11	42
	Code artifacts	–	12	12	–	26	14	5
15	RAD artifacts	–	65	65	–	81	16	13
	Code artifacts	–	–	–	–	–	–	–
16	RAD artifacts	67	85	18	42	–	–	–
	Code artifacts	–	–	–	–	–	–	–
17	RAD artifacts	–	66	66	–	84	18	42
	Code artifacts	–	–	–	–	44	44	–

links. However, in projects where most links were manually traced the project managers performed only few iterations and the traceability recovery sessions terminated with a very high lowest threshold. As most manually traced links were below such a threshold, the quality manager tuned the quality threshold (and decreased it below the lowest threshold used for traceability recovery) to avoid that most warning links were links traced between artifacts that passed the review. For example, in project 8 (where the traceability recovery tool was used only in the first period), most links were traced without the help of the traceability recovery tool (see Table VIII): in Table VII, the maximum lowest threshold used was 85% and required two iterations to be identified (in the case of tracing use cases onto sequence diagrams). However, most pairs of artifacts manually traced by the managers and considered of good quality after the

Table VII. Tracing Statistics

	1st Period				2nd Period				3rd Period				Quality Threshold	
	Iterations		Lowest Threshold		Iterations		Lowest Threshold		Iterations		Lowest Threshold			
Id	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
1	3	3	65	65	1	1	65	65	1	1	65	65	65	70
2	3	4	55	70	2	2	50	65	1	2	45	60	55	65
3	5	6	40	50	1	1	40	50	1	1	40	50	50	60
4	5	5	40	40	2	8	30	55	1	3	30	50	45	65
5	–	–	–	–	4	4	50	55	1	7	45	55	45	50
6	2	4	50	60	1	4	50	70	1	2	50	65	55	60
7	3	4	70	70	1	1	70	70	1	8	45	70	45	70
8	2	5	40	85	–	–	–	–	–	–	–	–	40	70
9	3	6	55	70	1	4	60	70	2	2	55	65	55	70
10	3	5	55	70	2	2	55	65	1	5	50	65	50	65
11	3	4	40	70	1	4	50	60	2	3	45	50	55	60
12	3	5	40	60	1	3	40	60	1	3	45	60	45	65
13	4	5	50	65	1	1	50	65	1	2	50	60	55	60
14	5	5	40	40	2	8	30	55	2	3	30	50	45	65
15	–	–	–	–	3	4	60	70	2	3	60	65	75	80
16	3	6	45	60	2	2	40	55	–	–	–	–	50	60
17	–	–	–	–	3	7	40	60	2	5	40	55	50	60

artifact review had a similarity value below 85% and the selected quality threshold was 70%, as shown in Table VII.

Table VIII shows the results achieved in each project during the analysis of the links suggested by the tool; in particular, for each period, the table shows the number of links manually traced by the software engineer and the number of suggested links traced or classified as false positives. For the second and third periods, the column of suggested links is split in two subcolumns: the column *Emerging* represents the number of suggested links that increased their similarity above the “optimal” threshold, as a consequence of some artifact changes, while the column *New* represents the number of suggested links arising as a consequence of new artifacts added to the repository. Both subcolumns are further divided into the number of links traced and the number of links classified as false positives. As we can see, the managers continued to trace links and discard false positives also in the second and third period. Most of these traceability links were due to new software artifacts, but the support provided by the tool in terms of emerging traceability links was also considerable in some projects (e.g., projects 2 and 12).

Table IX shows the results achieved in each project analyzing the set of warning links; in particular, for each period, the table shows the number of new warning links and the number of iterative warning links. Moreover, the number of warning links accepted (column A) and the number of change requests sent through feedbacks (column SF) by the managers are also shown for both new and iterative warning links. It is important to note that in the first period the number of iterative warning links is not shown because by definition this set is empty in the first period. Moreover, for the second and third period the column of new warning links is split in two subcolumns: the column Newly Traced

Table VIII. Analysis of Suggested Links

		1st Period			2nd Period					3rd Period				
			Suggested Links			Suggested Links					Suggested Links			
						Emerging		New			Emerging		New	
Id	Traced Artifacts	MTL	TL	FP	MTL	TL	FP	TL	FP	MTL	TL	FP	TL	FP
1	RAD vs RAD	47	2	9	0	0	0	0	0	0	0	3	0	0
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–
2	RAD vs RAD	27	53	59	9	8	11	30	55	3	5	9	23	20
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–
3	RAD vs RAD	0	29	61	0	0	3	0	0	5	0	5	2	16
	UC vs Code	0	16	23	0	0	0	0	0	0	0	2	0	0
4	RAD vs RAD	3	5	20	5	3	14	44	166	1	1	9	7	72
	UC vs Code	–	–	–	0	0	0	40	51	0	8	8	3	1
5	RAD vs RAD	–	–	–	197	0	0	37	72	13	0	13	1	29
	UC vs Code	–	–	–	–	–	–	–	–	0	0	0	51	78
6	RAD vs RAD	19	1	14	36	3	8	52	73	1	2	11	3	51
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–
7	RAD vs RAD	46	43	187	11	2	28	13	97	4	5	24	13	39
	UC vs Code	–	–	–	–	–	–	–	–	0	0	0	52	78
8	RAD vs RAD	45	21	92	–	–	–	–	–	–	–	–	–	–
	UC vs Code	0	24	97	–	–	–	–	–	–	–	–	–	–
9	RAD vs RAD	46	119	166	0	7	28	22	50	0	11	59	0	0
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–
10	RAD vs RAD	9	37	112	0	1	9	14	50	9	2	9	4	39
	UC vs Code	–	–	–	–	–	–	–	–	10	0	0	21	57
11	RAD vs RAD	7	13	30	8	1	5	5	59	9	2	27	0	0
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–
12	RAD vs RAD	2	36	121	0	3	37	7	87	0	0	11	14	63
	UC vs Code	0	22	67	10	15	15	4	14	0	0	0	18	39
13	RAD vs RAD	0	26	59	0	0	3	0	0	5	2	9	6	22
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–
14	RAD vs RAD	3	8	33	9	0	5	50	155	3	1	18	12	50
	UC vs Code	–	–	–	0	0	0	22	65	0	0	2	13	20
15	RAD vs RAD	–	–	–	0	0	0	112	145	0	1	25	13	83
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–
16	RAD vs RAD	21	48	145	0	5	33	22	77	–	–	–	–	–
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–
17	RAD vs RAD	–	–	–	24	0	0	39	121	0	4	23	22	113
	UC vs Code	–	–	–	–	–	–	–	–	0	0	0	42	60
MTL = Manually Traced Links – TL = Traced Links – FP = False Positives														

represents the number of new warning links highlighted by the tool due to the insertion of new traceability links in the current period, while the column Previously Traced represents the number of new warning links highlighted by the tool due to a decreased similarity between artifacts traced in a previous period (emerging warning links). The latter information is useful to monitor the evolution of both the links and the artifacts, because emerging warning links might help the software engineer in the identification of no longer valid links or of some quality problems (in terms of text description) arising in the traced artifacts as a consequence of some changes.

Table IX. Analysis of Warning Links

Id	Traced Artifacts	1st Period		2nd Period						3rd Period					
		New Warning Links		New Warning Links				Iterative Warning Links		New Warning Links				Iterative Warning Links	
				Newly Traced		Previously Traced				Newly Traced		Previously Traced			
		A	SF	A	SF	A	SF	A	SF	A	SF	A	SF	A	SF
1	RAD vs RAD	0	11	0	0	0	2	1	8	0	0	2	5	3	1
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–	–
2	RAD vs RAD	0	1	0	1	0	4	0	1	1	5	9	2	1	5
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–	–
3	RAD vs RAD	0	4	0	0	0	0	1	2	0	4	1	1	0	0
	UC vs Code	1	5	0	0	1	0	0	3	0	0	0	3	0	2
4	RAD vs RAD	0	1	0	3	0	1	0	1	0	2	0	1	3	2
	UC vs Code	–	–	2	5	0	0	–	–	0	1	0	0	6	0
5	RAD vs RAD	–	–	15	71	0	0	–	–	3	7	0	2	31	22
	UC vs Code	–	–	–	–	–	–	–	–	0	0	–	–	–	–
6	RAD vs RAD	0	4	2	8	1	9	1	3	1	1	14	20	7	9
	UC vs Code		–	–	–	–	–		–	–	–	–	–	–	–
7	RAD vs RAD	0	8	0	1	0	0	2	5	3	1	2	2	2	2
	UC vs Code	–	–	–	–	–	–	–	–	3	4	0	0	–	–
8	RAD vs RAD	3	7	–	–	–	–	–	–	–	–	–	–	–	–
	UC vs Code	2	4	–	–	–	–	–	–	–	–	–	–	–	–
9	RAD vs RAD	0	15	0	0	0	13	0	12	2	0	4	2	7	6
	UC vs Code	–	–	–	–	–	–	–	–	–	–		–		–
10	RAD vs RAD	0	2	0	3	0	9	1	0	2	3	3	0	6	2
	UC vs Code	–	–	–	–	–	–	–	–	1	3	0	0	–	–
11	RAD vs RAD	0	2	0	0	0	1	0	2	0	1	0	0	0	0
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–	–
12	RAD vs RAD	0	9	0	4	0	2	0	6	1	0	1	1	6	2
	UC vs Code	0	0	0	0	0	0	–	–	2	6	6	1	0	0
13	RAD vs RAD	0	2	0	0	0	2	0	1	0	1	1	3	0	0
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–	–
14	RAD vs RAD	0	2	0	3	0	0	0	1	1	1	0	1	2	1
	UC vs Code	–	–	0	3	0	0	–	–	1	1	0	0	2	0
15	RAD vs RAD	–	–	0	2	0	0	–	–	0	1	1	1	1	0
	UC vs Code	–	–	–	–	–	–	–	–	–	–	–	–	–	–
16	RAD vs RAD	0	10	1	0	1	1	2	4	–	–	–	–	–	–
	UC vs Code	–	–		–	–	–	–	–	–	–	–	–	–	–
17	RAD vs RAD	–	–	0	10	0	0	–	–	1	0	1	1	4	1
	UC vs Code	–	–	–	–	–	–	–	–	0	0	–	–	–	–
A = Accept – SF = Send feedback															

It is worth pointing out that if a warning link highlighted by the tool is no longer valid such a link should be removed by the software engineer. Such a situation did not occur in our experience, as none of the traceability links was removed. This was probably due to the small size and short duration of the projects. Nevertheless, the warning links played an important role during the artifact review process; in particular, they were considered a good support for the quality managers to check the completeness and adequacy of the artifacts descriptions, as in general low similarity between traced artifacts mirrors poor quality of the artifact descriptions. In particular, the quality managers reviewed the artifacts involved in warning links and on average about 70% of them actually had poor quality in terms of text description or use of nonmeaningful identifiers in source code. In these cases the quality managers decided to send a feedback to the artifact developers to ask for improvements of the text description (see column *SF* in Table IX). It is important to note that the quality managers in general, accepted the warning links highlighted by the tool when there was a substantial improvement of the similarity due to changes made to the traced artifacts or when the similarity was very closed to the quality threshold. A significant result emerging from Table IX is that in the first two periods the managers decided to send feedbacks in 92% of the cases. Moreover, about 60% of the warning links where feedbacks were sent in the first two periods increased their similarity above the quality threshold at the end of the project, due to modifications made to the artifacts.

7.2 Questionnaire Results

At the end of the projects, the students evaluated ADAMS through a questionnaire. A subset of the questions concerning traceability recovery were only answered by the 34 managers. Each question refers to an attribute of four quality characteristics of interest, namely functionality, usability, robustness, and performance. For each question, four options were available for the answers:

A. “I totally agree”
B. “I agree”
C. “I disagree”
D. “I totally disagree”

Table X shows for each question the distribution of the student answers. The results for the functionality are quite encouraging and the most important achievement concerns the incremental traceability recovery process: all the students preferred the incremental approach to the one-shot approach. In particular, some of them motivated their answer declaring that the identification of new links was simplified working on a relative small set of links to analyze. Moreover, 75% of students declared that discarding false positives was an expensive task, but some of them declared that this task was mitigated by the incremental traceability recovery approach. The other students declared that this task was not expensive, probably because they found only a small number of false positives. Very good results were achieved for the adequacy and usefulness of the tool: over 90% of the students said that the traceability recovery functionality of ADAMS covered their needs and that the analysis of suggested links was useful to identify new links. Moreover, over 90% of the students also declared that the analysis of the warning links was a good support to monitor the artifact quality.

Good results were also achieved for the usability. In particular, about 90% of the students declared that the ADAMS traceability recovery tool was simple to use and to learn. Moreover, about 80% of the students considered intuitive the traceability link definition and the similarity measure between two artifacts.

Table X. Questionnaire Results

Characteristic	Question	Answer (%)			
		A	B	C	D
Functionality	An incremental traceability recovery approach is preferred to an "one-shot" approach	55.88	44.12	0.00	0.00
	The traceability recovery functionality covers your needs	67.65	23.53	8.82	0.00
	The analysis of suggested links is useful to discovery new links	91.18	0.00	8.82	0.00
	Discarding false positives from the set of suggested links is an expensive task	35.29	41.18	20.59	2.94
	The analysis of warning links is useful to monitor the artifact quality	79.41	17.65	2.94	0.00
	The traceability recovery tool provides a support to the analysis of link evolution	61.76	29.41	8.82	0.00
Usability	Traceability link definition in ADAMS is intuitive	61.76	26.47	11.76	0.00
	Similarity measure between two artifact is intuitive	76.47	11.76	11.76	0.00
	It is easy to provide the system with input data	17.65	64.71	14.71	2.94
	It is easy to get output data from the system	26.47	55.88	14.71	2.94
	It is easy to learn the main procedure	50.00	41.18	8.82	0.00
	Interface components are well organized on the screen	55.88	35.29	8.82	0.00
	Terms denoting commands are clear	38.24	50.00	8.82	2.94
	It is easy to learn and remember single interface component roles	35.29	47.06	14.71	2.94
Robustness and Performance	The interface interaction mechanism prevents errors	23.53	58.82	14.71	2.94
	The response time of the ADAMS traceability recovery tool during a recovery execution is good	14.71	44.12	26.47	14.71
	The performances of the traceability management subsystem of ADAMS are good	38.24	47.06	8.82	5.88

Concerning the performances of the ADAMS traceability management subsystem, 85% of the students were satisfied while 60% declared that the performances were also good during the traceability recovery sessions. This is not an excellent result but it is acceptable if we consider that the tool is only a prototype and we included these questions to get suggestions for improvements.

7.3 Threats to Validity

This section describes the threats to validity that can affect our experience. These are important for our study since we aim at concluding that, despite the limitations of IR methods, the ADAMS traceability recovery tool still represents a useful support during traceability link identification and that an incremental approach is more acceptable than a "one-shot" approach.

In our experience, we tried to simulate a real working environment, although we only used master students. There was no abandonment and (as shown from the questionnaire results) the tool usage was clear. Moreover, students did not know exactly the goals of the experiment and they were not evaluated on their performances. Last-year master students have a very good analysis, development and programming experience, and they are not far from junior industry analysts. In addition, subjects are students enrolled in an advanced course of software engineering, so they have both knowledge of software development and project management. Unfortunately, in a real working environment there are other pressures and practitioners might have a much lower tolerance for issues such as discarding false positives. Thus, probably the results cannot be completely generalised to the industrial context and then the experience should be replicated with practitioners. However, we should also consider that the decision of stopping the process was only based on the perceived precision, as the students did not have any indication of the achieved recall. This bias might be in part balanced by integrating prediction models [Buckley and Voorhees 2004] in the traceability recovery tool, to give the software engineer also indications of what is the estimated recall during the incremental traceability recovery process.

Concerning the questionnaire, it was mainly indented to get qualitative insights. It is worth pointing out that the number of "positive" versus the number of "negative" questions was out of balance. This could be a bias because it was easy for students to agree. For this reason, we plan to replicate the experience using a higher number of "negative" questions in the questionnaire.

During the experience, we have left to the students all the decision concerning when and how traceability management had to be done during the development process. We instructed them on how to trace links using and not using the traceability recovery tool, and in the latter case both with a "one-shot" and with an incremental approach. However, they were free to trace links in any of the presented approaches.

The artifact repositories built by the students during the experience have a small/medium size but they represent a good benchmark as they cover many different projects with different goals. However, students mainly traced links between artifacts of the Requirements Analysis Document (RAD), namely functional requirements, scenarios, use cases, and sequence diagrams, as these artifacts were available through the different project phases. Unfortunately, we do not know if the fact that links between code and documentation were not traced in the other eight projects was only due to the fact that source code artifacts were available for a shorter period than RAD artifacts, or also to a greater complexity of such a task (unfortunately, we did not ask this question to the students). However, in the nine projects where links were also traced between code artifacts and use cases we did not observe any meaningful difference, that might induce to argue that this task was more difficult than tracing links between RAD artifacts. To further support this issue, a controlled experiment on traceability recovery has been conducted with master students [De Lucia et al. 2006b]. From the results, we could not find any statistical difference on performing traceability tasks with different types of artifacts when using the tool. However, it is possible that this is due to the fact that code artifacts are written by students using meaningful identifiers and comments that help in the comprehension as well as the terms in higher level artifacts. This might also be a limitation to the generalization of the results to an industrial context.

8. CONCLUSION AND DISCUSSION

In this article, we have integrated a traceability recovery tool into an artifact management system called ADAMS. The traceability recovery tool is based on Latent Semantic Indexing (LSI), an Information Retrieval (IR) technique. The tool helps the software engineer in the identification of potential traceability links not traced yet (*Suggested Links*) and in the identification of possible text description problems in the traced artifacts (*Warning Links*). The implementation of the tool was based on a preliminary study aimed at assessing LSI as a traceability recovery technique, identifying strengths and limitations of using IR techniques for traceability recovery, and devising a way to use them. We experimented the traceability recovery tool during the development of seventeen software projects conducted by undergraduate students with development roles and master students with management roles. The traceability links recovered by the master students were validated in review meetings conducted by all members of each team and supervised by Ph.D. students and academic researchers. The results of the evaluation confirmed the initial findings of the preliminary study on LSI and provided us with a number of lessons learned:

—*Experimental results should drive the design decisions in the implementation of a tool*. The preliminary study discussed in Section 5 was useful to assess LSI as a traceability recovery technique. We devised the need to cut the ranked lists produced by LSI using variable threshold and categorization (a ranked list for each pair of artifact types), to avoid some of the problems deriving from the different verbosity of different artifact types. Moreover, we decided to periodically compute the singular value decomposition of the term-by-document matrix as well as the ranked lists of links, rather than at check-in time (this would consume too many computing resources and too often) or when the traceability recovery tool is used by the software engineer (this would slow-down the interactive functionality). We also devised the need to tune the similarity threshold used to cut the ranked list in an incremental process, due to the fact that an “optimal” threshold is not known a priori; indeed, using a too high threshold would cause loss of too many correct links, while using a too low threshold would cause the retrieval of a high number of false positives that could produce a loss of confidence in the tool. Concerning the size of the LSI subspace, we cannot draw any conclusion. In the projects described in Section 5, we achieved good results when using between 30% and 100% of concepts. It is possible to observe some different behavior with large industrial projects, but this issue is still open also in the information retrieval literature [Deerwester et al. 1990; Dumais 1992]. For this reason, we decided to leave this as a configuration parameter.

—*IR methods provide a useful support to the identification of traceability links, but are not able to identify all traceability links*. The evaluation discussed in Section 7 has revealed that almost all managers traced most links using the traceability recovery tool. For this reason, we propose to use the tool during software development to improve the performances of simply manual tracing. To further support this claim, a controlled experiment has been recently performed to empirically assess the advantages of using the ADAMS traceability recovery tool, with respect to manual tracing [De Lucia et al. 2006b]. The main result achieved in this experiment is that the use of a traceability recovery tool significantly reduces the time spent by the software engineer with respect to manual tracing. In this experiment, the links traced by the software engineers were compared to the traceability matrix provided by the original developers (set of correct links). A practical but not statistically significant result was that subjects using the tool generally traced more correct links than subjects manually performing the traceability recovery tasks (higher recall).

The limitation of IR-based traceability recovery is the fact that these methods cannot help in the identification of all correct links, without forcing the software engineer to analyze and discard a high number of false positives. In addition, it is almost impossible to automatically identify how many links the software engineer needs to analyze in the ranked list to be sure that all correct links have been considered. This means that to be sure that also the last correct link in the ranked list (the correct link with lowest similarity value) has been considered, usually the software engineer has to analyze almost all the links in the ranked list. Unfortunately, this limitation is not definitely mitigated by improving the IR-based traceability method with other IR techniques, such as text pre-processing or relevance feedback analysis. Indeed, in a recent work [De Lucia et al. 2006a] stemming and relevance feedbacks have been incorporated in traceability recovery processes based on both the Vector Space Model and Latent Semantic Indexing. This work revealed that adding these techniques still does not solve the problem of recovering all correct links, as the effort required to discard false positives remains too high. Moreover, when the results of IR methods are already good, due to a good verbosity and quality of the artifacts, these techniques do not provide any improvement at all. This means that the limitations of IR techniques have to be solved by complementing them with other techniques that are not based on text analysis, such as structural or syntactic techniques [Antoniol et al. 2000a; Briand et al. 2003; Maletic et al. 2005; Murphy et al. 2001].

—*The incremental approach to traceability recovery was preferred to a "one-shot" approach*. Although we did not impose the adoption of an incremental traceability recovery process, in our experience, all users preferred an incremental approach, thus confirming the findings of our preliminary study. The number of iterations made in the traceability sessions was greater in the first period and when a high number of artifacts were added in the following periods. In other words, with a high number of potential new links the users tuned again the threshold using an incremental approach, despite the fact that the tool proposes the lowest threshold used in the previous sessions. Only when the number of potential new links was small, one or two iterations were needed starting with the lowest threshold of the previous sessions. Unfortunately, in our experience, we could not collect data about the recall, but based on the considerations above, it is likely that software engineers did not recover all correct links, as they used an incremental approach and discarded only an acceptable number of false positives. This means that they actually decided to stop the process when the number of false positives discarded became too high with respect to the number of correct links traced. It is possible that the behavior was different if the software engineers also had an estimate of the achieved recall [Buckley and Voorhees 2004], besides the perceived precision. As further consideration, we do not know if using a one-shot approach the users would have retrieved more correct links. However, we do not see any reason why using a one shot approach the software engineer would proceeds to lower links in the ranked list and be more patient in discarding false positives. Indeed, while in the upper part of the ranked list the density of correct links is quite good, in the bottom part of the list such a density decreases in such a way that the prioritization made by the IR method does not help anymore. As a matter of fact, in the controlled experiment discussed in [De Lucia et al. 2006b], the software engineers traced more links using the traceability recovery tool with an incremental approach than using a completely manual analysis of the list of all possible links.

—*IR-based methods can help in the identification of quality problems in the text description of software artifacts*. The experiments conducted with real users revealed the usefulness of maintaining the list of warning links for the identification of some quality problems in the text description of traced artifacts, mainly a poor description of the artifacts. The quality managers used this information to organize reviews and ask for changes of artifacts considered not satisfactory. As a result of these changes, over 60% of the warning links highlighted by the tool improved the similarity value above the quality threshold at the end of the project.

Future work will be devoted to further experiment and assess the tool in larger software projects, in particular projects of industrial partners interested in using ADAMS. Moreover, there are a number of directions to improve the performances of the information retrieval method. A first direction aims at integrating in the ADAMS traceability recovery tool a learning algorithm exploiting the feedbacks provided by the user when classifying links during the incremental traceability recovery process [De Lucia et al. 2006a]. However, as discussed in De Lucia et al. [2006a] this would not be a definitive solution to the limitations of IR methods. Therefore, a second direction would be to combine LSI with syntactic-based approaches [Antoniol et al. 2000a; Briand et al. 2003; Maletic et al. 2005; Murphy et al. 2001] and with data mining techniques [Cleland-Huang et al. 2005], in particular by exploiting information about version history [Gall et al. 1998; 2003; Ying et al. 2004; Zimmermann et al. 2005]. We also plan to exploit the structure of documents expressed in XML as the latter would also enable the use of a context-sensitive information retrieval approach. Finally, we aim at integrating prediction models [Buckley and Voorhees 2004] to give the software engineer indications of the estimated recall during the incremental traceability recovery process.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their detailed, constructive, and thoughtful comments that helped us to improve the presentation of the results in this article. We are very grateful to Prof. Mike Berry of the University of Tennessee who kindly provided the Java implementation of the SVD package.

REFERENCES

ALEXANDER, I. 2002. Towards automatic traceability in industrial practice. In *Proceedings of 1st International Workshop on Traceability in Emerging Forms of Software Engineering* (Edinburgh, UK). 26–31.

ANTONIOL, G., CANFORA, G., CASAZZA, G., AND DE LUCIA, A. 2000a. Identifying the starting impact set of a maintenance request. In *Proceedings of 4th European Conference on Software Maintenance and Reengineering* (Zurich, Switzerland, Feb.). IEEE Computer Society Press, Los Alamitos, CA, 227–230.

ANTONIOL, G., CAPRILE, B., POTRICH, A., AND TONELLA, P. 2000b. Design-code traceability for object oriented systems. *Ann. Softw. Eng. 9*, 35–58.

ANTONIOL, G., CASAZZA, G., AND CIMITILE, A. 2000c. Traceability recovery by modelling programmer behavior. In *Proceedings of 7th Working Conference on Reverse Engineering* (Brisbane, Queensland, Australia, Nov.). IEEE Computer Society Press, Los Alamitos, CA, 240–247.

ANTONIOL, G., CANFORA, G., CASAZZA, G., DE LUCIA, A., AND MERLO, E. 2002. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng. 28*, 10, 970–983.

ARNOLD, S. P., AND STEPOWAY, S. L. 1988. The reuse system: Cataloging and retrieval of reusable software. In *Software Reuse: Emerging Technology*, W. Tracz, Ed. IEEE Computer Society Press, Los Alamitos, CA, 138–141.

AVERSANO, L., DE LUCIA, A., GAETA, M., AND RITROVATO, P. 2003. GENESIS: A flexible and distributed environment for cooperative software engineering. In *Proceedings of 15th International Conference on Software Engineering and Knowledge Engineering* (San Francisco, CA, July). 497–502.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison-Wesley, Reading, MA.

BUCKLEY. C. AND VOORHEES, M. 2004. Retrieval evaluation with incomplete information. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Sheffield, UK, July). ACM, New York, 25–32.

BIGGERSTAFF, T. 1989. Design recovery for maintenance and reuse. *IEEE Comput. 22*, 7, 36–49.

BRIAND, L. C., LABICHE, Y., AND O'SULLIVAN, L. 2003. Impact analysis and change management of UML models. In *Proceedings of 19th International Conference on Software Maintenance* (Amsterdam, The Netherlands, Sept.). IEEE Computer Society Press, Los Alamitos, CA, 256–265.

BOLDYREFF, C., NUTTER, D., AND RANK, S. 2002. Active artifact management for distributed software engineering. In *Proceedings of 26th IEEE Annual International Computer Software and Applications Conference* (Oxford, England, UK, Aug.). IEEE Computer Society Press, Los Alamitos, CA, 1081–1086.

BURTON, B. A., ARAGON, R. W., BAILEY, S. A., KOELHER, K., AND MAYES, L. A. 1987. The reusable software library. In *Software Reuse: Emerging Technology*, W. Tracz, Ed. IEEE Computer Society Press, Los Alamitos, CA, 129–137.

CAPRILE B. AND TONELLA, P. 1999. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of 6th IEEE Working Conference on Reverse Engineering* (Atlanta, GA, Oct.). IEEE Computer Society Press, Los Alamitos, CA, 112–122.

CHEN J. Y. J. AND CHOU, S. C. 1999. Consistency management in a process environment, *J. Syst. Softw. 47*, 2–3, 105–110.

CLELAND-HUANG, J., CHANG, C. K., AND CHRISTENSEN, M. 2003. Event-based traceability for managing evolutionary change. *IEEE Trans. Softw. Eng. 29*, 9, 796–810.

CLELAND-HUANG, J., SETTIMI, R., DUAN, C., AND ZOU, X. 2005. Utilizing supporting evidence to improve dynamic requirements traceability. In *Proceedings of International Requirements Engineering Conference* (Paris, France, Aug.). IEEE Computer Society Press, Los Alamitos, CA, 135–144.

CONKLIN J. AND BEGEMAN, M. L. 1988. Gibis: A hypertext tool for exploratory policy discussion. *ACM Trans. Office Inf. Syst. 6*, 4, 303–331.

CUGOLA, G. 1998. Tolerating deviations in process support systems via flexible enactment of process models. *IEEE Trans. Softw. Eng. 24*, 11, 982–1001.

CUGOLA, G., DI NITTO, E., FUGGETTA, A., AND GHEZZI, C. 1996. A framework for formalizing inconsistencies in human-centered systems. *ACM Trans. Softw. Eng. Meth. 5*, 3, 191–230.

CULLUM, J. K. AND WILLOUGHBY, R. A. 1985. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations, vol. 1: Theory*. Chapter 5: "Real rectangular matrices," Brikhauser, Boston, MA.

DAG, J., REGNELL, B., CARLSHAMRE, P., ANDERSSON, M., AND KARLSSON, J. 2002. A feasibility study of automated natural language requirements analysis in market-driven development. *Require. Eng. 7*, 1, 20–33.

DE LUCIA, A., FASANO, F., FRANCESE, R., AND TORTORA, G. 2004a. ADAMS: An artifact-based process support system. In *Proceedings of 16th International Conference on Software Engineering and Knowledge Engineering* (Banff, Alberta, Canada, June). F. Maurer and G. Ruhe, Eds. 31–36.

DE LUCIA, A., FASANO, F., FRANCESE, R., AND OLIVETO, R. 2004b. Recovering traceability links between requirement artifacts: A case study. In *Proceedings of 16th International Conference of Software Engineering and Knowledge Engineering* (Banff, Alberta, Canada, June). F. Maurer, and G. Ruhe, Eds. 453–466.

DE LUCIA, A., FASANO, F., OLIVETO, R., AND TORTORA, G. 2004c. Enhancing an artifact management system with traceability recovery features. In *Proceedings of 20th IEEE International Conference on Software Maintenance* (Chicago, IL). IEEE Computer Society Press, Los Alamitos, CA, USA, 306–315.

DE LUCIA, A., FASANO, F., OLIVETO, R., AND TORTORA, G. 2005a. ADAMS Re-trace: A traceability recovery tool. In *Proceedings of 9th IEEE European Conference on Software Maintenance and Reengineering* (Manchester, UK). IEEE Computer Society Press, Los Alamitos, CA, 32–41.

DE LUCIA, A., FASANO, F., FRANCESE, R., AND OLIVETO, R. 2005b. Traceability management in ADAMS. In *Proceedings of 1st International Workshop on Distributed Software Development* (Paris, France). 135–149.

DE LUCIA, A., FASANO, F., OLIVETO, R., AND TORTORA, G. 2005c. Recovering traceability links in software artifact management systems: Detailed experimental results, Technical Report, Software Engineering Lab, Department of Mathematics and Informatics, University of Salerno, Italy (available from http://www.sesa.dmi.unisa.it/tr/TR05_01.pdf).

DE LUCIA, A., OLIVETO, R., AND SGUEGLIA, P. 2006a. Incremental approach and user feedbacks: A silver bullet for traceability recovery?. In *Proceedings of 22nd International Conference on Software Maintenance* (Sheraton Society Hill, Philadelphia, PA). 299–309.

DE LUCIA, A., OLIVETO, R., AND TORTORA, G. 2006b. Supporting traceability link recovery via information retrieval: A controlled experiment. Technical Report, Software Engineering Lab, Department of Mathematics and Informatics, University of Salerno, Italy, submitted for publication (available from http://www.sesa.dmi.unisa.it/tr/TR06_01.pdf).

DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K., AND HARSHMAN, R. 1990. Indexing by latent semantic analysis. *J. Amer. Soc. Inf. Sci. 41*, 391–407.

DI LUCCA, G. A., DI PENTA, M., AND GRADARA, S. 2002. An approach to classify software maintenance requests. In *Proceedings of the IEEE International Conference on Software Maintenance* (Montréal, Qué., Canada). IEEE Computer Society Press, Los Alamitos, CA, 93–102.

Di Penta, M., Gradara, S., and Antoniol, G. 2002. Traceability recovery in RAD software systems. In *Proceedings of the 10th IEEE International Workshop on Program Comprehension* (Paris, France). IEEE Computer Society Press, Los Alamitos, CA, 207–216.

Domges, R. and Pohl, K. 1998. Adapting traceability environments to project specific needs. *Commun. ACM 41*, 12, 55–62.

Dumais, S. T. 1991. Improving the retrieval of information from external sources. *Behav. Res. Meth. Instrum. Comput. 23*, 229–236.

Dumais, S. T. 1992. LSI meets TREC: A status report. *The First Text REtrieval Conference*, NIST special publication 500-207, D. Harman, Ed. 137–152.

Egyed, A. and Grünbacher, P. 2002. Automating requirements traceability: Beyond the record and replay paradigm. In *Proceedings of 17th IEEE International Conference on Automated Software Engineering* (Edinburgh, UK, Sept.). IEEE Computer Society Press, Los Alamitos, CA, 163–171.

Finkelstein, A., Spanoudakis, G., and Till, D. 1996. Managing interference. In *Joint Proceedings of the 2nd International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development on SIGSOFT '96 workshops* (San Francisco, CA). ACM, New York, 172–174.

Frakes, W. B. and Nejmeh, B. A. 1987. Software reuse through information retrieval. In *Proceedings of 20th Hawaii International Conference on System Science* (Kola, HI). IEEE Computer Society Press, Los Alamitos, CA, 530–535.

Gall, H., Hajek, K., and Jazayeri, M. 1998. Detection of logical coupling based on product release history. In *Proceedings of IEEE International Conference on Software Maintenance* (Bethesda, MD). IEEE Computer Society Press, Los Alamitos, CA, 190–198.

Gall, H., Jazayeri, M., and Krajewski, J. 2003. CVS release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, IEEE Computer Society Press, Los Alamitos, CA, 13–23.

Gotel, O. and Finkelstein, A. 1994. An analysis of the requirements traceability problem. In *Proceedings of 1st International Conference on Requirements Engineering* (Colorado Springs, CO). IEEE Computer Society Press, Los Alamitos, CA, 94–101.

Harman, D. 1992. Ranking algorithms. In *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 363–392.

Holagent Corporation, 2006. RDD-100, http://www.holagent.com/products/product1.html.

Huffman Hayes, J., Dekhtyar, A., and Osborne, J. 2003. Improving requirements tracing via information retrieval. In *Proceedings of 11th IEEE International Requirements Engineering Conference* (Monterey, CA). IEEE Computer Society Press, Los Alamitos, CA, 138–147.

Huffman Hayes, J., Dekhtyar, A., and Karthikeyan Sundaram, S. 2006. Advancing candidate link generation for requirements tracing: The study of methods. *Trans. Softw. Eng. 32*, 1, 4–19.

Leffingwell, D. 1997. Calculating your return on investment from more effective requirements management. *Rational Software Corporation*. (Available online from http://www.rational.com/products/whitepapers).

Lormans, M. and van Deursen, A. 2006. Can LSI help reconstructing requirements traceability in design and test? In *Proceedings of 10th European Conference on Software Maintenance and Reengineering* (Bari, Italy). 45–54.

Maarek, Y., Berry, D., and Kaiser, G. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng. 17*, 8, 800–813.

Maletic, J. I. and Marcus, A. 2001. Supporting program comprehension using semantic and structural information. In *Proceedings of 23rd International Conference on Software Engineering* (Toronto, Ont., Canada). 103–112.

Maletic, J. I., Collard, M. L., and Simoes, B. 2005. An XML based approach to support the evolution of model-to-model traceability links. In *Proceedings of the 3rd ACM International Workshop on Traceability in Emerging Forms of Software Engineering* (Long Beach, CA). 67–72.

Maletic, J. I., Munson, E. V., Marcus, A., and Nguyen, T. N. 2003. Using a hypertext model for traceability link conformance analysis. In *Proceedings of 2nd International Workshop on Traceability in Emerging Forms of Software Engineering* (Montreal, Que., Canada). 47–54.

MARCUS, A. AND MALETIC, J. I. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of 25th International Conference on Software Engineering* (Portland, OR). 125–135.

MARCUS, A., SERGEYEV, A., RAJLICH, V., AND MALETIC, J. I. 2004. An information retrieval approach to concept location in source code. In *Proceedings of 11th IEEE Working Conference on Reverse Engineering* (Delft, The Netherlands). IEEE Computer Society Press, Los Alamitos, CA, 214–223.

MARCUS, A., XIE, X., AND POSHYVANYK, D. 2005. When and how to visualize traceability links? In *Proceedings of the 3rd ACM International Workshop on Traceability in Emerging Forms of Software Engineering* (Long Beach, CA). ACM, New York, 56–61.

MERLO, E., MCADAM, I., AND MORI, R. D. 1993. Source code informal information analysis using connectionist models. In *Proceedings of International Joint Conference on Artificial Intelligence* (Chambéry, France). 1339–1344.

MURPHY, G. C., NOTKIN, D., AND SULLIVAN, K. 2001. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng. 27*, 4, 364–380.

NGUYEN, T. N., THAO, C., AND MUNSON, E. V. 2005. On product versioning for hypertexts. In *Proceedings of the 12th International Workshop on Software Configuration Management* (Lisbon, Portugal). 99–111.

NISTOR, E. C., ERENKRANTZ, J. R., HENDRICKSON, S. A., AND VAN DER HOEK, A. 2005. ArchEvol: Versioning architectural-implementation relationships. In *Proceedings of the 12th International Workshop on Software Configuration Management* (Lisbon, Portugal). 99–111.

NUSEIBEH, B. 1996. Towards a framework for managing inconsistency between multiple views. In *Joint Proceedings of the 2nd International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development on SIGSOFT '96 workshops* (San Francisco, CA). ACM, New York, 184–186.

PALMER, J. D. 2000. Traceability. In *Software Requirements Engineering*, Second Edition, R. H. Thayer and M. Dorfman, Eds. IEEE Computer Society Press, Los Alamitos, CA, 412–422.

PIGHIN, M. 2001. A new methodology for component reuse and maintenance. In *Proceedings of 5th European Conference on Software Maintenance and Reengineering* (Lisbon, Portugal). IEEE Computer Society Press, Los Alamitos, CA, 196–199.

PINHEIRO, F. A. C. AND GOGUEN, J. A. 1996. An object-oriented tool for tracing requirements. *IEEE Softw. 13*, 2, 52–64.

RAMESH, B. AND DHAR, V. 1992. Supporting systems development using knowledge captured during requirements engineering. *IEEE Transactions on Software Engineering 9*, 2, 498–510.

RATIONAL SOFTWARE, 2006. Rational RequisitePro, http://www.rational.com/products/reqpro/index.jsp.

RICHARDSON, J. AND GREEN, J. 2004. Automating traceability for generated software artifacts. In *Proceedings of 19th IEEE International Conference on Automated Software Engineering* (Linz, Austria). IEEE Computer Society Press, Los Alamitos, CA, 24–33.

RITTEL, H. AND KUNZ, W. 1970. Issues as elements of information systems. Working paper N°I 31, Institut fur Grundlagen der Planung I.A. University of Stuttgart.

SALTON, G. AND BUCKLEY, C. 1988. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage. 24*, 5, 513–523.

SARMA, A. AND VAN DER HOEK, A. 2002. Palantír: Coordinating distributed workspaces. In *Proceedings of the 26th Annual IEEE International Computer Software and Applications Conference* (Oxford, UK). IEEE Computer Society Press, Los Alamitos, CA, 1093–1097.

SEFIKA, M., SANE, A., AND CAMPBELL, R. H. 1996. Monitoring compliance of a software system with its high-level design models. In *Proceedings of 16th International Conference on Software Engineering* (Berlin, Germany). 387–396.

SETTIMI, R., CLELAND-HUANG, J., BEN KHADRA, O., MODY, J., LUKASIK, W., AND DEPALMA, C. 2004. Supporting software evolution through dynamically retrieving traces to UML artifacts. In *Proceedings of 7th International Workshop on Principles of Software Evolution* (Kyoto, Japan). IEEE Computer Society Press, Los Alamitos, CA, 49–54.

SMITH, M., WEISS, D., WILCOX, P., AND DEWER, R. 2003. The Ophelia traceability layer. In *Cooperative Methods and Tools for Distributed Software Processes*, A. Cimitile, A. De Lucia, and H. Gall, Eds., Franco Angeli, 150–161.

SPANOUDAKIS, G. AND ZISMAN, A. 2001. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering*, S. K. Chang, Ed. World Scientific Publishing Co., 24–29.

TELELOGIC, 2006. DOORS, http://www.telelogic.com.

VON KNETHEN, A. AND GRUND, M. 2003. QuaTrace: A tool environment for (semi-) automatic impact analysis based on traces. In *Proceedings of IEEE International Conference on Software Maintenance* (Amsterdam, The Netherlands). IEEE Computer Society Press, Los Alamitos, CA, 246–255.

WEIDL, J. AND GALL, H. 1998. Binding object models to source code. In *Proceedings of 22nd IEEE Annual International Computer Software and Applications Conference* (Vienna, Austria). IEEE Computer Society Press, Los Alamitos, CA, 26–31.

YING, A. T. T., MURPHY, G. C., NG, R., AND CHU-CARROLL, M. C. 2004. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng. 30*, 9, 574–586.

ZHAO, W., ZHANG, L., LIU, Y., SUN, J., YANG, F. 2004. SNIAFL: Towards a static non-interactive approach to feature location. In *Proceedings of 26th International Conference on Software Engineering* (Edinburgh, UK). 293–303.

ZIMMERMANN, T., WEISSGERBER, P., DIEHL, S., AND ZELLER, A. 2005. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng. 31*, 6, 429–445.

ZISMAN, A., SPANOUDAKIS, G., PEREZ-MIÑANA, E., AND KRAUSE, P. 2003. Tracing software requirements artifacts. In *Proceedings of International Conference on Software Engineering Research and Practice* (Las Vegas, NV). 448–455.

Received December 2004; revised January 2006 and October 2006; accepted December 2006