Recovering Traceability Links between Code and Documentation

Giuliano Antoniol, *Member*, *IEEE*, Gerardo Canfora, *Member*, *IEEE*, Gerardo Casazza, *Member*, *IEEE*, Andrea De Lucia, *Member*, *IEEE*, and Ettore Merlo, *Member*, *IEEE*

Abstract—Software system documentation is almost always expressed informally in natural language and free text. Examples include requirement specifications, design documents, manual pages, system development journals, error logs, and related maintenance reports. We propose a method based on information retrieval to recover traceability links between source code and free text documents. A premise of our work is that programmers use meaningful names for program items, such as functions, variables, types, classes, and methods. We believe that the application-domain knowledge that programmers process when writing the code is often captured by the mnemonics for identifiers; therefore, the analysis of these mnemonics can help to associate high-level concepts with program concepts and vice-versa. We apply both a probabilistic and a vector space information retrieval model in two case studies to trace C++ source code onto manual pages and Java code to functional requirements. We compare the results of applying the two models, discuss the benefits and limitations, and describe directions for improvements.

Index Terms—Redocumentation, traceability, program comprehension, object orientation, information retrieval.

1 Introduction

Most of the documentation that accompanies large software systems consists of free text documents expressed in a natural language. Examples include requirements and design documents, user manuals, logs of errors, maintenance journals, design decisions, reports from inspection and review sessions, and also annotations of individual programmers and teams. In addition, free text documents often capture the available knowledge of the application domain, for example, in the form of laws and regulations or in technical/scientific handbooks. Even when (semi-)formal models are applied, free text is largely used either to add semantics and context information in the form of comments, or to ease the understanding of the formal models to nontechnical readers. As a matter of fact, diagrammatic representations are often supplemented with free text descriptions that convey information that is not captured by the diagrams themselves and a Z [43] specification document is typically an amalgam of mathematics (that is precise and supports reasoning) and explanatory text that makes the document an effective means of communication.

Establishing traceability links between the free text documentation associated with the development and maintenance cycle of a software system and its source code can be helpful in a number of tasks. A few notable examples are:

- **Program comprehension**. Existing cognition models share the idea that program comprehension occurs in a bottom-up manner [39], [40], a top-down manner [14], [51], or some combination of the two [30], [32], [33], [34]. They also agree that programmers use different types of knowledge during program comprehension, ranging from domain specific knowledge to general programming knowledge [14], [50], [53]. Traceability links between areas of code and related sections of free text documents, such as an application domain handbook, a specification document, a set of design documents, or manual pages, aid both top-down and bottom-up comprehension. In top-down comprehension, once a hypothesis has been formulated, the traceability links provide hints on where to look for beacons that either confirm or refute it. In bottom-up comprehension, the main role of the traceability links is to assist programmers in the assignment of a concept to a chunk of code and in the aggregation of chunks into more abstract concepts.
- **Maintenance**. As the software industry matures, companies have built up a sizable number of legacy systems. A legacy system is an old system which is valuable for the corporation which owns and which often developed it. For the purpose of maintaining legacy systems, design recovery, as defined in [18], may be performed, thus requiring different sources of information, such as source code, design documentation, personal experience, and general knowledge about problem and application domains [12], [35]. Central to design recovery is representation [47], for which different schemes have been used and described in [45] and [13]. Traceability links between code and other sources of information

- *G. Antoniol, G. Canfora, and A. De Lucia are with the Research Centre on Software Technology, Department of Engineering, University of Sannio, Palazzo Bosco Lucarelli, Piazza Roma, I-82100 Benevento, Italy. E-mail: antoniol@ieee.org, {gerardo.canfora, delucia}@unisannio.it.*
- *G. Casazza is with the Department of Informatica e Sistemistica, University of Naples, Federico II, Via Claudio 21, I-80125 Naples, Italy. E-mail: gec@unisannio.it.*
- *E. Merlo is with the Department of Electrical and Computer Engineering, Ecole Politechnique, C.P. 6079—Succ. Centre Ville, Montreal, Quebec, Canada. E-mail: ettore.merlo@polymtl.ca.*

Manuscript received 15 Oct. 2000; revised 5 Apr. 2001; accepted 7 Nov. 2001. Recommended for acceptance by A. Andrews.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 115376.

are a sensible help to perform the combined analysis of heterogeneous information and, ultimately, to associate domain concepts with code fragments and vice-versa.

- **Requirement tracing**. Traceability links between the requirement specification document and the code are a key to locate the areas of code that contribute to implement specific user functionality [42], [28], [44]. This is helpful to assess the completeness of an implementation with respect to stated requirements, to devise complete and comprehensive test cases, and to infer requirement coverage from structure coverage during testing. Traceability links between requirements and code can also help to identify the code areas directly affected by a maintenance request as stated by an end user. Finally, they are useful during code inspection, providing inspectors with clues about the ultimate goals of the code at hand and in quality assessment, for example, singling out loosely-coupled areas of code that implement heterogeneous requirements.
- **Impact analysis**. A major goal of impact analysis is the identification of the work products affected by a proposed change [9]. Changes may initially have been made to any of the documents that comprise a system, or to the code itself, and have then to be propagated to other work products [25], [52]. As an example, enhancing an existing system by adding new functions is, in most cases, initiated at the requirement specification level and changes are then propagated through design documents down to the source code. Conversely, a change of an algorithm or a data structure may start at the code level and is then documented in the relevant sections of the design documentation. This shows a need for a means of establishing traceability links between code and free text documentation.
- **Reuse of Existing Software**. Deriving reusable assets from existing software systems has emerged as a winning approach to promote the practice of reuse in industry [16], [17]. Means to trace code to free text documents are a sensible help to locate reuse-candidate components. Indeed, since existing software often has not been produced with reuse in mind, knowledge about its functionality and the application domain concepts it implements is spread in several places, including requirement specification documents, manual pages, design documents, and the code itself. In addition, traceability links between the code and an application domain handbook are helpful to index reusable assets according to an application domain model and to implement querying facilities that retrieve them for potential reuse based on a user request [10], [15], [24].

Unlike other reverse engineering problems, recovering traceability links between free text documentation and source-code components cannot be simply based on compiler techniques because of the difficulty of applying syntactic analysis to natural language sentences. We propose a method based on Information Retrieval (IR) to establish and maintain traceability links between the source code and free text documents. Automated IR systems are concerned with the retrieval of documents from (usually very large) document databases, based on user information needs [23]. They prepare the collection of documents for retrieval through an indexing process; user needs are captured by phrases which are themselves indexed and used to rank the documents.

IR has proven useful in many areas, including the management of huge scientific and legal literature, office automation, and the support to complex engineering projects such as software engineering projects. We believe that IR techniques can provide a way to semiautomatically recovering traceability links between the documentation of a system and its source code. Similar to Biggerstaff [12] and Biggerstaff et al. [13], a premise of our work is that programmers use meaningful names for program items, such as functions, variables, types, classes, and methods. Much of the application-domain knowledge that programmers process when writing the code is often captured by the mnemonics for identifiers; therefore, the analysis of these mnemonics can help to associate high-level concepts with program concepts, and vice-versa [12], [13], [38]. Moreover, while in [12], [13], the names of program items are used as a clue to suggest concepts implemented in the code, we use the name to locate relevant pieces of documentation.

A widely used approach to retrieve documents from the document space is ranked retrieval, which returns a ranked list of documents [27]. The method proposed in this paper ranks the free-text documents against queries constructed from the identifiers of source code components and can be customized to work with different IR models. In this paper, two IR models have been applied, a probabilistic model and a vector space model [27].

In the probabilistic model, free-text documents are ranked according to the probability of being relevant to a query computed on a statistical basis. To compute this ranking, we exploit the idea of a language model, i.e., a stochastic model that assigns a probability to every string of words taken from a prescribed vocabulary [21]. We estimate a language model (actually, a unigram approximation of the model) for each document, or identifiable section, and use a Bayesian classifier to score the sequences of mnemonics extracted from each source code component against the models. A high score indicates a high probability that a particular sequence of mnemonics be relevant to the document; therefore, we interpret it as an indication of the existence of a semantic link between the component from which the sequence had been extracted and the document.

The vector space model treats documents and queries as vectors in an n-dimensional space, where n is the number of indexing features (in our case, words in the vocabulary). Documents are ranked against queries by computing a distance function between the corresponding vectors. In this paper, the documents are ranked according to a widely used distance function, i.e., the cosine of the angle between the vectors [27], [48].

The two IR models have been applied in two case studies. In the first case study, the C++ classes of the LEDA

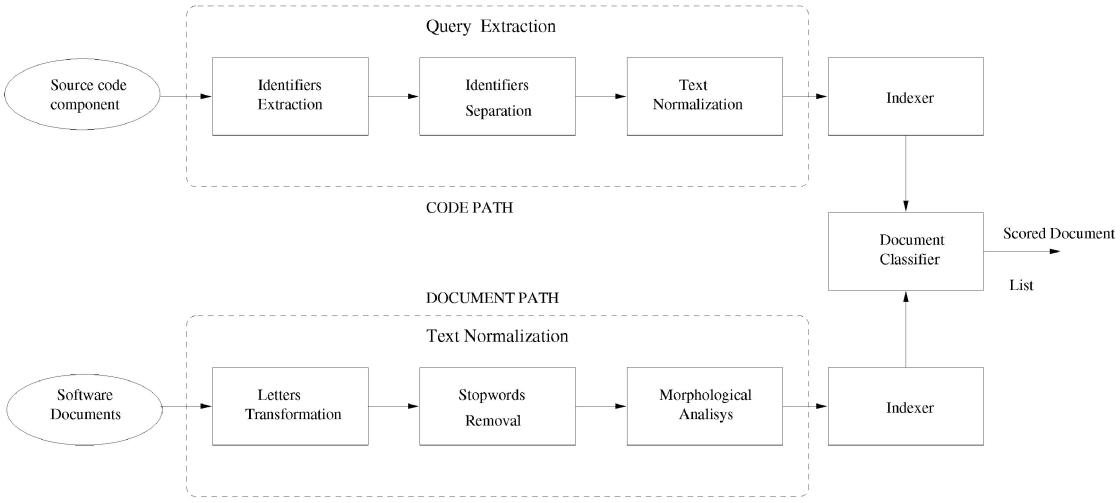

Fig. 1. Traceability Link Recovery Method.

library (Library of Efficient Data Types and Algorithms)[1] have been traced to relevant pages. In the second case study, traceability links have been recovered between the Java classes of a hotel management system, named Albergate, and its functional requirements, as specified in the requirement document. The results, measured in terms of the two well-known IR metrics, precision and recall [23], are in both cases satisfactory. This supports the hypothesis that IR, either probabilistic or vector space models, provides a practicable solution to the problem of semiautomatically recovering traceability links.

The traceability link recovery method based on IR has been evaluated by comparing it with the results achieved using the `grep` UNIX utility, according to the approach proposed by Maarek et al. [31]. Finally, the benefits of the method in helping software engineers to recover traceability links between code and free text documentation has been experimentally evaluated.

The remainder of the paper is organized as follows: Section 2 presents the traceability link recovery method and discusses the IR models exploited. Results from two case studies are presented in Section 3, together with a discussion and comparison of the performances of the two IR models used. Section 4 evaluates the benefits of IR models when applied to the problem of recovering traceability links. Section 5 discusses related work; finally, Section 6 gives concluding remarks and outlines lessons learned and directions for future work.

2 Traceability Link Recovery Method

Our method to recover traceability links between code and free text documentation uses the identifiers extracted from a source code component as a query to retrieve the documents relevant to the component. We assume that programmers use meaningful names (i.e., names derived from the application and problem domain) for their identifiers and/or that identifiers are preprocessed to extract names that share the semantics of the requirements (e.g., splitting sequences of words contained in a single identifier); consequently, "words" were chosen as indexing feature.

This section describes the overall traceability link recovery method, gives background information on the IR models applied, and discusses tool support.

2.1 The Process

Fig. 1 shows the approach for traceability link recovery using IR models. The figure highlights two paths of activities, one to prepare the document for retrieval (document path) and the other to extract the queries from code (code path).

In the document path, documents are indexed based on a vocabulary that is extracted from the documents themselves. The construction of the vocabulary and the indexing of the documents are preceded by a text normalization phase performed in three steps:

1. In the first step, all capital letters are transformed into lower case letters.
2. In the second step, stop-words (such as articles, punctuation, numbers, etc.) are removed.
3. In the third step, a morphological analysis is used to convert plurals into singulars and to transform conjugated forms of verbs into infinitives.

The code path builds and indexes a query for each source code component. The construction of a query consists of three steps:

1. Identifier extraction parses the source code component and extracts the list of its identifiers.
2. Identifier separation splits identifiers composed of two or more words into separate words (i.e., AmountDue and amount_due would be split into the words amount and due).
3. Text normalization applies the three steps described above for document indexing.

Finally, a classifier computes the similarity between queries and documents and returns a ranked list of

1. Freely available for academic research and teaching from http://www.mpi.sb.mpg.de/LEDA.

documents for each source code component. Documents are ranked against a source code component by decreasing similarity.

Of course, indexing the documents and the queries and ranking the documents against a query depend on the particular IR model adopted. In the following sections, two IR models are described, a probabilistic model and a vector space model. In the probabilistic model, free text documents are ranked according to the probability of being relevant to a source code component. The vector space model treats documents and queries as vectors; documents are ranked against queries by computing a distance function between the corresponding vectors.

2.2 Probabilistic IR Model

This model computes the ranking scores as the probability that a document D_i is related to the source code component Q (that is the query Q).

$$Similarity(D_i, Q) = Pr(D_i|Q).$$

Applying Baye's rule [11], the conditioned probability above can be transformed in:

$$Pr(D_i|Q) = \frac{Pr(Q|D_i)Pr(D_i)}{Pr(Q)}.$$

For a given source code component, $Pr(Q)$ is a constant and we can further simplify the model by assuming that all system documents have the same probability. Therefore, for a given source code component Q, all documents D_i are ranked by the conditioned probabilities $Pr(Q|D_i)$.

These conditioned probabilities are computed by estimating a stochastic language model [21] for each document D_i. Indeed, due to the hypothesis that the source code components and the documents insist on the same vocabulary V, a source code component Q can be represented by a sequence of m words $w_1, w_2, \ldots, w_m$ (the identifiers of the source code component) of the vocabulary V and the conditioned probability:

$$Pr(Q|D_i) = Pr(w_1, w_2, \ldots, w_m \mid D_i)$$

can be estimated on a statistical basis by exploiting a stochastic language model for the document D_i. This model collects statistics about the frequence of the occurrences of sequences of words of V in D_i that allow to estimate $Pr(w_1, w_2, \ldots, w_m \mid D_i)$ for any sequence of words $w_1, w_2, \ldots, w_m$ of V. However, the probability above can be written as:

$$\begin{aligned} &Pr(w_1, w_2, \ldots, w_m \mid D_i) \\ &= Pr(w_1 \mid D_i) \prod_{k=2}^{m} Pr(w_k \mid w_1, \ldots, w_{k-1}, D_i). \end{aligned}$$

and when m increases the conditioned probabilities involved in the above product quickly become difficult to estimate for any possible sequence of m words in the vocabulary. A simplification can be introduced by conditioning the dependence of each word to the last $n-1$ words (with $n < m$):

$$\begin{aligned} &Pr(w_1, w_2, \ldots, w_m \mid D_i) \\ &\simeq Pr(w_1, \ldots, w_{n-1} \mid D_i) \prod_{k=n}^{m} Pr(w_k \mid w_{k-n+1}, \ldots, w_{k-1}, D_i). \end{aligned}$$

This n-gram approximation, which formally assumes a time-invariant Markov process [20], greatly reduces the statistics to be collected in order to compute $Pr(Q|D_i)$; clearly, this also introduces an imprecision. However, n-gram models are still difficult to estimate because, if $|V|$ is the size of the vocabulary, all possible $|V|^n$ sequences of words in the vocabulary have to be considered; indeed, the estimation can be very demanding even for a 2-gram (bigram) model.[2] Moreover, the occurrence of any sequence of words in a document D_i is a rare event, as it generally occurs only a few times and most of the sequences will never occur due to the sparseness of data. Therefore, in our approach, we have considered a unigram approximation ($n = 1$) that corresponds to consider all words w_k to be independent. Therefore, each document D_i is represented by a language model where unigram probabilities are estimated for all words in the vocabulary and:

$$\begin{aligned} &Similarity(D_i, Q) = Pr(Q|D_i) \\ &= Pr(w_1, w_2, \ldots, w_m \mid D_i) \simeq \prod_{k=1}^{m} Pr(w_k \mid D_i). \end{aligned}$$

Unigram estimation is based on the term frequency of each word in a document. However, using the simple term frequency would turn the product $\prod_{k=1}^{m} Pr(w_k \mid D_i)$ to zero, whenever any word w_k is not present in the document D_i. This problem, known as the zero-frequency problem [55], can be avoided using different approaches (see [21]). The approach we have adopted consists of smoothing the unigram probability distribution by computing the probabilities as follows:

$$Pr(w_k \mid D_i) = \begin{cases} \frac{c_k - \beta}{N} + \lambda & \text{if } w_k \text{ occurs in } D_i \\ \lambda & \text{otherwise,} \end{cases}$$

where N is the total number of words in the document D_i and c_k is the number of occurrences of words w_k in the document D_i. The interpolation term λ is:

$$\lambda = \frac{n}{N * |V|}\beta,$$

where n is the number of different words of the vocabulary V occurring in the document D_i. The value of the parameter β is computed according to Ney and Essen [37] as follows:

$$\beta = \frac{n(1)}{n(1) + 2 * n(2)},$$

where $n(j)$ is the number of words occurring j times in the document D_i.

2.3 Vector Space IR Model

Vector space IR models map each document and each query onto a vector [27]. In our case, each element of the vector corresponds to a word (or term) in a vocabulary extracted

2. In a bigram model, $Pr(w_1, w_2, \ldots, w_m \mid D_i) \simeq Pr(w_1 \mid D_i) \prod_{k=2}^{m} Pr(w_k \mid w_{k-1} D_i)$.

from the documents themselves. If $|V|$ is the size of the vocabulary, then the vector $[d_{i,1}, d_{i,2}, \ldots d_{i,|V|}]$ represents the document D_i. The jth element $d_{i,j}$ is a measure of the weight of the jth term of the vocabulary in the document D_i. Different measures have been proposed for this weight. In the simplest case, it is a Boolean value, either 1 if the jth term occurs in the document D_i, or 0 otherwise; in other cases, more complex measures are constructed based on the frequency of the terms in the documents.

We use a well-known IR metric called tf-idf [48]. According to this metric, the jth element $d_{i,j}$ is derived from the *term frequency* $tf_{i,j}$ of the jth term in the document D_i and the *inverse document frequency* idf_j of the term over the entire set of documents. The term frequency $tf_{i,j}$ is the ratio between the number of occurrences of word jth over the total number of words contained in the document D_i. The inverse document frequency idf_j is defined as:

$$idf_j = \frac{Total\ Number\ of\ Documents}{Number\ of\ Documents\ containing\ the\ j^{th}\ term}.$$

The vector element $d_{i,j}$ is:

$$d_{i,j} = tf_{i,j} * log(idf_j).$$

The term $log(idf_j)$ acts as a weight for the frequency of a word in a document: the more the word is specific to the document, the higher the weight.

The list of identifiers extracted from a source code component Q is represented in a similar way by a vector $[q_1, q_2, ...q_{|V|}]$. The similarity between a document D_i and a source code component Q is computed as the cosine of the angle between the corresponding vectors:

$$Similarity(D_i, Q) = \frac{\sum_{j=1}^{V} d_{i,j} q_j}{\sqrt{\sum_{h=1}^{V} (d_{i,h})^2 * \sum_{k=1}^{V} (q_k)^2}}.$$

2.4 Tool Support

We have developed a toolkit that supports and partially automates the method shown in Fig. 1. In particular, we consider source code components consisting each of an object-oriented class, written either in C++ or in Java. We use top-down recursive parsers to analyze C++ and Java source code. The parse trees are traversed and, each time a class is encountered, the comments, if any, and the identifiers of attributes, methods, and method parameters are stored in support files. For the present study, comments were disregarded: the entire traceability link recovery method relies on the mnemonics used for classes, attributes, methods, and parameters.

We have integrated public domain facilities and tools developed in house to assist text processing for the English and Italian languages. Identifier separation is performed in two steps: the first step is completely automated and recognizes words separated by underscore and sequences of words starting with capital letters. The second step is semi-automatic: the tool exploits spelling facilities to prompt the software engineer with the words that might be separated. The first two steps of text normalization, namely letter transformation and stop-word removing, have also been completely automated. Finally, we have implemented a semiautomatic tool that uses thesaurus facilities to help users to transform words into their roots.

The document indexer and document classifier have been implemented according to the two IR models experimented. For the probabilistic model, we used the CMU tool suite [46] to estimate the stochastic language models and we have implemented a Bayesian classifier that computes, for each language model, the $Pr(Q|D_i)$ for the given input text. For the vector space IR model, the computation of the vector elements and the final step of cosine computation and document ranking are implemented by simple Perl scripts.

3 Case Study

We have applied the traceability link recovery method based on both the probabilistic IR model and the vector space IR model in two case studies with different characteristics. The results have been assessed using two widely accepted IR metrics, namely, *recall* and *precision* [23]. *Recall* is the ratio of the number of relevant documents retrieved for a given query over the total number of relevant documents for that query. *Precision* is the ratio of the number of relevant documents retrieved over the total number of documents retrieved. It is worth noting that each query retrieves a ranked list of documents. We use a cut level N to select the first N documents in the list and analyze the behavior of Recall and Precision with different values of N.

Recovering traceability links is a semiautomatic process. The main role of IR tools consists of restricting the document space, while recovering all documents relevant to each source code component. Without tool support, one must analyze all the documents before discovering that a given class is not described by any document; with a restricted document space the number of documents to analyze is generally much smaller. This means that high recall values (possibly 100 percent) should be pursued; of course, in this case higher precision values reduce the effort required to discard false positives (documents that are retrieved but are not relevant to a given query).

It is worth noting that the 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. To take into account such queries, we used the following aggregate formulas:

$$Recall = \frac{\sum_i \#(Relevant_i \wedge Retrieved_i)}{\sum_i \#Relevant_i}\%,$$

$$Precision = \frac{\sum_i \#(Relevant_i \wedge Retrieved_i)}{\sum_i \#Retrieved_i}\%,$$

where i ranges over the entire query set, including the queries with no associated documents. These queries do not affect the computation of the recall ($Relevant_i$ is the empty set), while they negatively affect the computation of the precision whenever $Retrieved_i$ is not the empty set. This negative influence takes into account the effort required to discard false positives.

TABLE 1
LEDA Results

		Probabilistic IR model			Vector Space IR model		
Cut	Retrieved	Relevant	Precision	Recall	Relevant	Precision	Recall
1	208	81	38.94 %	82.65 %	52	25.00 %	53.06 %
2	416	88	21.15 %	89.79 %	71	17.06 %	72.44 %
3	624	93	14.90 %	94.89 %	79	12.66 %	80.61 %
4	832	93	11.17 %	94.89 %	82	9.85 %	83.67 %
5	1040	93	8.94 %	94.89 %	85	8.17 %	86.73 %
6	1248	93	7.45 %	94.89 %	89	7.13 %	90.81 %
7	1456	94	6.45 %	95.91 %	90	6.18 %	91.83 %
8	1664	94	5.64 %	95.91 %	93	5.58 %	94.89 %
9	1872	95	5.07 %	96.93 %	95	5.07 %	96.93 %
10	2080	95	4.56 %	96.93 %	96	4.61 %	97.95 %
11	2288	95	4.15 %	96.93 %	96	4.19 %	97.95 %
12	2496	96	3.84 %	97.95 %	98	3.92 %	100.00 %

3.1 LEDA Case Study

The first case study was a freely available C++ library of foundation classes, called LEDA (Library of Efficient Data types and Algorithms), developed and distributed by Max-Planck-Institut für Informatik, Saarbrücken, Germany. We analyzed the code and the documentation of release 3.4, consisting of 95 KLOC, 208 classes, and 88 manual pages. The aim was to map source code classes onto manual pages.

The LEDA manual pages contain a high number of identifiers that also appear in the source code. Actually, the LEDA team generated manual pages with scripts that extract comments from the source files. A markup language was used to identify the comment fragments to be extracted. Function names, parameter names, and data type names contained in these comments appear in the manual pages, thus making the traceability link recovery task easier. For this reason, we applied a simplified version of the method shown in Fig. 1. The simplification concerned the identifier separation and the text normalization activities; in particular, identifier separation only consisted of splitting identifiers containing underscores, while text normalization was performed only at the first level of accuracy, i.e., the transformation of capital letters into lower case letters.

To validate the results, we used a 208×88 traceability matrix linking each class to the manual page describing it [6]. Each class was described by at most one manual page and many classes (110) were not described by any manual page. The number of links in the traceability matrix was 98. Ten manual pages did not describe LEDA classes, but basic concepts and algorithms, thus, the number of relevant manual pages was 78. This means that some manual pages described more than one class: for example, very often an abstract class and its derived concrete classes were described by the same manual page (we discovered 20 of such cases).

Table 1 shows the results. The first two columns show the number of documents retained for each query (first N documents in the ranked list) and the total number of documents retrieved by all queries for each cut level. The table also shows for each IR model and each cut level the total number of relevant documents retrieved by all queries and the aggregate precision and recall values.

The poor results of the precision with both IR models are due to the fact that most of the queries (110) were derived from classes without relevant manual pages associated (these queries contribute to the total number of retrieved documents). The main difference between the two IR models is that the probabilistic model retrieves most of the documents with smaller cut values. However, the vector space model achieves 100 percent of recall sooner than the probabilistic model (see Fig. 2), i.e., cutting the ranked list of documents at 12 candidates, whereas 17 is the cut level required to achieve 100 percent of recall with the probabilistic model (not shown in the table).

3.2 Albergate Case Study

The second case study was a software system, called Albergate, developed in Java according to a waterfall process. For this system, all the documentation related to the entire software development process was available (e.g.,

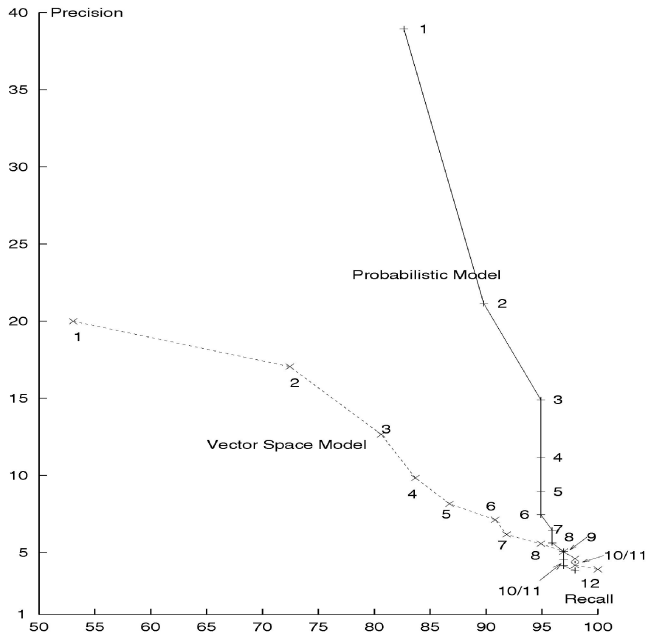

Fig. 2. LEDA precision/recall results.

TABLE 2
Albergate Results with Improved Process

		Probabilistic IR model			Vector Space IR model		
Cut	Retrieved	Relevant	Precision	Recall	Relevant	Precision	Recall
1	60	29	48.33 %	50.00 %	29	48.33 %	50.00 %
2	120	41	34.16 %	70.68 %	34	28.33 %	58.62 %
3	180	45	25.00 %	77.58 %	46	25.55 %	79.31 %
4	240	51	21.25 %	87.93 %	51	21.25 %	87.93 %
5	300	57	19.00 %	98.27 %	54	18.00 %	93.10 %
6	360	58	13.80 %	100.00 %	55	15.27 %	94.82 %
7	420	58	13.80 %	100.00 %	58	13.80 %	100.00 %

requirement documents, design documents, test cases, etc.). Albergate is a software system designed to implement all the operations required to administer and manage a small/medium size hotel (room reservation, bill calculation, etc.). It was developed from scratch by a team of final year students at the University of Verona (Italy) on the basis of 16 functional requirements written in Italian (as well as all other system documentation). Albergate consists of 95 classes and about 20 KLOC and exploits a relational database. The aim of this case study was to trace source code classes to functional requirements [5]. We focused on the 60 classes implementing the user interface of the software system.

To validate the results, the original developers were required to provide a 16×60 traceability matrix linking each requirement to the classes implementing it. Most of the functional requirements were implemented by a low number of classes: on the average, a requirement was implemented by about 4 classes with a maximum of 10. Most classes were associated with one requirement, only 6 classes were associated with two requirements, and 8 classes were not associated with any functional requirement. The total number of links in the traceability matrix was 58.

In this case study, we applied the full version of the text processing steps described in section 3.1 (see Fig. 1). The motivation was that the relative distance between source code and documents was higher than in the LEDA case study. Common words between requirements and classes were quite infrequent in the Albergate system: In fact, unlike LEDA manual pages, Albergate functional requirements were produced in the early phases of the software development life cycle. Moreover, the Italian language has a complex grammar: Verbs have many more conjugated variants than English verbs, plurals are almost always irregular, and adverbs and adjectives have irregular forms, too.

Table 2 shows the results of this case study (the meaning of the columns is the same as in Table 1). Unlike the LEDA case study, the results of the vector space model are not very different than those produced by the probabilistic model (see Fig. 3). However, for the probabilistic model 100 percent of recall was obtained by considering the first 6 documents for each class, while for the vector space model all traceability links were recovered by considering the first 7 documents for each class.

3.3 Probabilistic vs. Vector Space Model

The two case studies suggest that both IR models (vector space and probabilistic) are suitable for the problem of recovering traceability links between code and documentation. The results are very similar, in particular, with respect to the number of documents a software engineer needs to analyze to get very high values of recall. However, the data show that the probabilistic model achieves higher values of recall with smaller cut values and makes little progress towards 100 percent of recall. On the other hand, the vector space model starts with lower recall values and makes regular progress with higher cut values towards 100 percent of recall.

A possible explanation lies in the nature of the two models. The probabilistic model associates a source code component (in our case studies a class) with a document based on the product of the unigram probabilities with which each code component identifier appears in the software document [6], [5] (see Section 2.2).

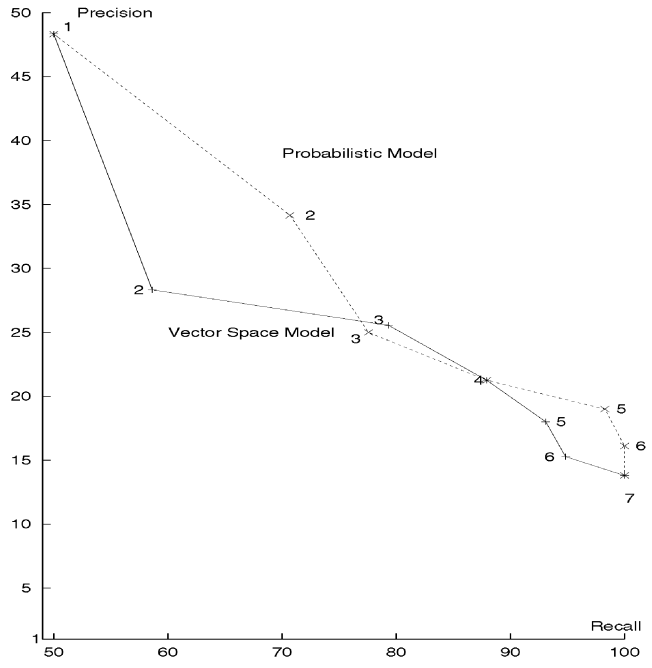

Fig. 3. Albergate precision/recall results with improved process.

TABLE 3
Albergate Results with Simplified Process

		Probabilistic IR model			Vector Space IR model		
Cut	Retrieved	Relevant	Precision	Recall	Relevant	Precision	Recall
1	60	15	25.00 %	25.86 %	23	38.33 %	39.65 %
2	120	17	14.16 %	29.31 %	33	27.50 %	56.89 %
3	180	20	11.11 %	34.48 %	38	21.11 %	65.51 %
4	240	23	9.58 %	39.65 %	46	19.16 %	78.86 %
5	300	28	9.33 %	48.27 %	48	16.00 %	82.75 %
6	360	30	8.33 %	52.72 %	52	14.44 %	89.65 %
7	420	32	7.61 %	55.17 %	54	12.85 %	93.10 %

These probabilities are computed on a statistical basis and code component identifiers that do not appear in the document are assigned a very low probability. Conversely, the similarity measure of a vector space model only takes into account the code identifiers that also appear in the document and weigh the frequencies of the occurrences of such words in the document (code component) with respect to a measure of their distribution in other documents (code components, respectively).

Therefore, the probabilistic model is more suitable for cases where the presence of code component identifiers that do not belong to the software document is low: This is also the reason why, with respect to the best match, the probabilistic model performs better in the LEDA case study (82.65 percent of recall) than in the Albergate case study (50 percent of recall). It is worth noting that the probabilistic model exploited in this paper is also used in speech recognition [21] and information theory [20] fields, where the aim is to associate a received sentence with a possible transmitted sentence, with a very low error probability. Conversely, the vector space model fits cases where each group of words is common to a relatively small number of software documents [4]. This means that the vector space model does not aim for the best match, but rather to regularly achieve the maximum recall with a moderate number of retained documents.

This hypothesis is supported by the results obtained by applying the simplified version of the text processing steps in Fig. 1 to the Albergate case study, with both the probabilistic and the vector space models. The simplified versions of the identifier separation and text normalization steps produce code components and software documents with a higher number of different words. Table 3 shows the results achieved, while Fig. 4 depicts the Precision/Recall curves of the two IR models, in both simplified and improved processes. For the vector space model, the results of the simplified and improved processes are not very different. Conversely, the differences are evident when applying the two versions of the text processing steps with the probabilistic model [5]. This means that, unlike the probabilistic model, the vector space model is able to achieve higher recall values based on a smaller number of relevant words in a source code component.

4 Evaluation

The traceability link recovery method presented in this paper has been evaluated with respect to the following criteria:

- performances of an IR model with respect to standard tools,
- effectiveness of an IR model in helping a software engineer,
- effort saving with respect to the granularity of the document space, and
- effectiveness of a fixed cut versus a variable cut.

These issues will be discussed in the next sections.

4.1 Comparing IR Models with `grep`

We compared the results achieved in the two case studies with the probabilistic and vector space IR models with the results obtained by using the `grep` UNIX utility, as proposed by Maarek et al. [31]. In fact, `grep` provides the simplest way to trace source code components (e.g., classes)

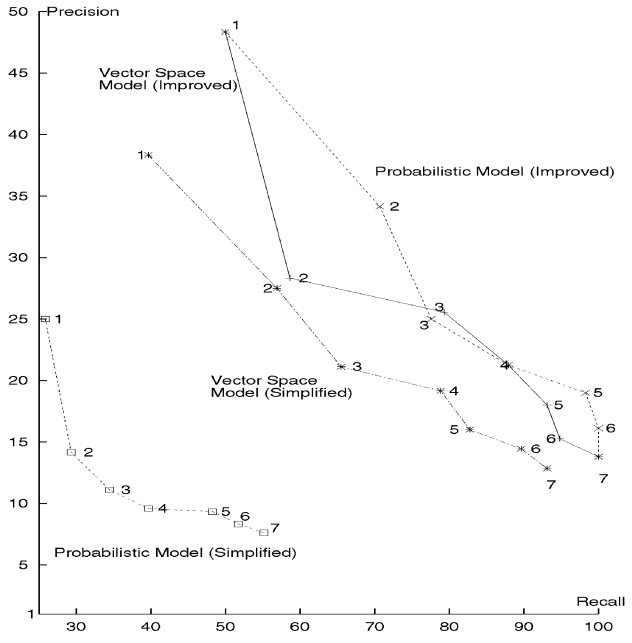

Fig. 4. Albergate precision/recall results.

TABLE 4
`grep` Results

	Single Code Item				Code Items *or* Combined			
	#Queries	#Empty Set	Mean Size	Max Size	#Queries	#Empty Set	Mean Size	Max Size
Albergate	4834	4575	5	14	60	0	11	13
LEDA	4670	451	20	88	208	1	75	88

onto high-level documentation (e.g., manual pages and/or requirements). The search can be done at least in two ways: In the first approach, each class identifier is used as the string to be searched in the files of high-level artifacts, while the second approach considers the *or* of the class identifiers.

Table 4 shows the results of the `grep` approach: It is worth noting that, for the Albergate system, 94 percent of the single item queries gave empty results, while if items are *or* combined 94 percent of classes were traced to 10 or more requirements. Empty sets are less frequent for `LEDA`; however, the average number of traced manual pages is quite high (20 and 75, respectively). Even worse, the `grep` approach did not offer any way to rank the retrieved requirements. From a practical point of view, this means that the maintainer has to examine a large number of candidates with the same priority.

4.2 Benefits of IR in a Traceability Recovery Process

To assess the effectiveness of the proposed approach, a preliminary *in-field* experiment was designed. The experiment concerned the Albergate system and involved eight students, four final year undergraduate students, and four postgraduate students. All students were familiar with the procedural and the object-oriented programming paradigms; however, their experience with the Java programming language was quite different. The undergraduate students had been introduced to Java just six months before the experiment took place. In particular, they attended a course during which they developed a small size project using Java. On the other hand, the postgraduate students had gained a significative experience of Java. In fact, they learned the language during the undergraduate courses and used it to develop the project for their graduation thesis. Moreover, at the time of our experiment, they were involved in other Java-based projects.

Two groups were formed: Group A (three undergraduate and one postgraduate students) and Group B (three postgraduate and one undergraduate students). The same task was assigned to both groups: the reconstruction of the Albergate traceability matrix. A copy of the requirements document and of the Albergate source code was given to each student. In addition, students of Group A also received, for each source code class, the ranked list of requirements obtained by applying the traceability link recovery method to the probabilistic IR model. However, no indication of where the ranked lists had to be cut (see Section 4) was provided.

On average, the performance of the two groups was better than the performance achieved by the probabilistic IR model on the best match, as shown in Table 5. However, the best results were obtained on the average by the students that exploited the results of the traceability link recovery method (Group A). It is worth noting that this group was mainly composed of undergraduate students, less expert with Java. Also, note that the best performance within Group A was obtained by the postgraduate student, as shown in Table 6. Table 7 shows the results achieved by students of Group B.

4.3 Considerations on Effort Saving and Document Granularity

Although the limited sample of our experiment does not allow to generalize conclusions, the preliminary data demonstrate the benefits of helping a software engineer with an automated approach based on IR models. Future work will be devoted to experiments involving a larger number of software engineers from industry to determine whether there is some statistical evidence of the benefits of using our method. With these experiments, we also aim to demonstrate a correlation between the use of the results of an IR model and the effort required to recover the tracebility link matrix.

In the previous sections, we have evaluated the results using the IR metrics *recall* and *precision*. To achieve an indication of the benefits of using an IR approach in a traceability link recovery process, we can also introduce a *Recovery Effort Index* (*REI*), defined as the ratio between the number of documents retrieved and the total number of documents available:

$$REI = \frac{\#Retrieved}{\#Available}\%.$$

TABLE 5
Average Results

	Recall	Precision
Prob. Appr.	50%	48%
Group A	65%	60%
Group B	57%	53%

TABLE 6
Results of Group A

	Recall	Precision
Und. Std1	65%	63%
Und. Std2	65%	56%
Und. Std3	58%	44%
Post. Std1	72%	76%

TABLE 7
Results of Group B

	Recall	Precision
Post. Std2	53%	56%
Post. Std3	70%	40%
Post. Std4	53%	59%
Und. Std4	55%	58%

This metric can be used to estimate the percentage of the effort required to manually analyze the results achieved by an IR tool (and discard false positive), when the recall is 100 percent, with respect to a completely manual analysis. For a given software system, the quantity $1 - REI$ can be used to estimate the effort saving due to the use of an IR method to recover traceability links, with respect a completely manual analysis. The lower the REI, the higher the benefits of the IR approach.

At present, we do not have statistical evidence of a correlation between the REI and the effort required to accomplish the recovery task, although, in the student experiment, we have noticed that students of Group A on the average took less time to do the job; the statistical validation of the REI will be part of future work.

It is interesting to observe that the REI also measures the ratio between the precision of the results achieved on the same software system by a completely manual process, namely, P_m, and a semiautomatic IR tool-based process, namely, P_t, when the recall is 100 percent:

$$\frac{Precision_m}{Precision_t} = \frac{\#(Relevant \wedge Retrieved_m)}{\#(Relevant \wedge Retrieved_t)} \frac{\#Retrieved_t}{\#Retrieved_m}.$$

Note that the number of relevant documents retrieved is the same in both processes (all relevant documents) and that the documents retrieved with a manual analysis are just all documents available, then

$$\frac{Precision_m}{Precision_t}\% = \frac{\#Retrieved_t}{\#Available}\%$$

that is the REI for the semiautomatic process.

The values of REI registered in the two case studies for the vector space IR model are rather different: Albergate requires 43.75 percent REI to achieve 100 percent recall, whereas, LEDA only requires 13.63 percent REI. A possible explanation is that the set of available documents in the Albergate case study is smaller (16 functional requirements versus the 88 manual pages of LEDA); to get the same REI as in the LEDA case study the maximum recall would have to be achieved with about two documents retrieved (that also means about 50 percent precision). However, this is very unlikely to be achieved with IR methods. They generally aim to retrieve a small percentage of a huge document space. Therefore, it is likely to hypothesize that greater benefits (and lower values of REI) are achieved for document spaces of greater size. Larger document spaces are also achieved when different concepts are included in different documents. Therefore, it is likely to think that greater benefits of this approach are achieved when the granularity of the concepts included in the documents is finer.

TABLE 8
Albergate Results Using a Threshold

Percentage	Retrieved	Relevant	Precision	Recall
90 %	59	29	49.15 %	50.00 %
70 %	101	38	37.62 %	65.51 %
50 %	158	50	31.64 %	86.20 %
30 %	265	55	20.75 %	94.82 %
10 %	484	58	11.98 %	100.00 %
min(10 %, best 7)	329	58	17.62 %	100.00 %

Alternatively, the REI could be computed with respect to a manual analysis supported by `grep` (queries with *or* combined items). In this case, the REI is computed as the ratio between the number of relevant documents retrieved with an IR approach and the number of documents retrieved by `grep`.[3] For the vector space model the values for REI are 54.54 percent in the Albergate case study and 16 percent in LEDA.

4.4 Retrieving a Variable Number of Documents

In our case studies, we have retained a fixed number of documents for each query. The results achieved for the recall can be considered good, as, in both case studies, we were able to achieve 100 percent recall with a moderate number of retained candidates per query.

We wondered if with a variable number of retained candidates per query we could improve precision and REI, while maintaining a maximum recall. The approach adopted to test this hypothesis consisted of using a threshold on the similarity values to prune the ranked list of documents retrieved by a query. In particular, for each query Q, we computed the value of such a threshold t_Q as a percentage of the similarity measure of the best match:[4]

$$t_Q = c * [\max_i s_{i,Q}],$$

where $0 \leq c \leq 1$. A query Q returns all and only the documents D_k such that $s_{k,Q} \geq t_Q$. Of course, the higher the value of the parameter c, the smaller the set of documents returned by a query.

Table 8 shows the results achieved with the vector space IR model for the Albergate case study using different values of the parameter c (and then different thresholds). The results are not very encouraging, as the maximum recall is achieved when setting the threshold to only 10 percent of the highest similarity measure. Using this percentage, the average number of retrieved documents per query is 9, while 3 documents are retrieved in the best case, and 15 documents in the worst case.

Although the results for the precision are worse than the results achieved with a fixed cut (first seven documents in Table 2), they still demonstrate the benefits of using an IR approach: Indeed, when the recall is 100 percent ($c = 10$ percent), the REI is 50.41 percent; this means that

3. Of course, this requires that the `grep`-based approach achieves 100 percent recall, as in our case.

4. For the sake of simplicity $Similarity(D_i, Q)$ is expressed as $s_{i,Q}$.

presumably about 50 percent of the effort can be saved by only discarding the documents whose similarity measure is below 10 percent of the best match.

Of course, the results can be improved by mixing a variable and fixed cut: Each query retrieves only the documents with a similarity measure greater than a given threshold, but no more than a fixed number. As an example, the last row in Table 8 shows the results achieved by considering as the number of documents retrieved by each query the minimum between seven and the number of documents whose similarity value is higher than 10 percent of the best match. In this case, the results are better than the results achieved with a fixed cut (the first seven documents in Table 2): The average number of retrieved documents is six and the REI is 34.27 percent, that means that the percentage of effort saved might be more than 65 percent.

5 Related Work

Several papers have been proposed in the literature that deal with recovering traceability links between source code and documentation, or that apply analysis of informal information or information retrieval to software engineering. Impact analysis is related to traceability link recovery. Most papers in the field assume the existence of some form of ripple propagation graph describing relations between software artifacts, including code and documentation, and focus on the prediction of the effects of a maintenance change request on both the source code and the specification and design documents [9], [25], [52].

TOOR [42], IBIS [28], and REMAP [44] are a few examples of CASE tools that maintain traceability links among various software artifacts. However, these tools are focused on the development phase and either force naming conventions or require human interventions to define the links.

Some methods used to recover traceability links between source code and design documents have been presented in the literature. Sefika et al. [49] have developed a hybrid approach that integrates logic-based static and dynamic visualization and helps determining design-implementation congruence at various levels of abstraction. Murphy et al. [36] exploit software reflexion models to match a design expressed in the Booch notation against its C++ implementation. Regular expressions are used to exploit naming conventions and map source code model entities onto high-level model entities. Similarly, Antoniol et al. [7], [22] present a method to trace C++ classes to an OO design. Both the source code classes and the OO design are translated into an Abstract Object Language (AOL) intermediate representation. Then, the AOLs extracted from design and code are compared using a maximum match algorithm [19] that computes the best mapping between source code classes and entities of the OO design based on the Levenstein string edit distance [26]. The idea of adopting a more tolerant string matching is also central to [54], where procedural applications are rearchitectured into OO systems. Central to the approach is a binding step to link candidate objects (extracted from the code) to elements of an object model recovered from the system documentation. Binding is a semiautomatic step, guided by a similarity measure inspired to the Dice's coefficient [23] computed over n-gram substrings: For 3-gram, the similarity between terms is based on the shared unique sequences of three characters [23], [2], [1]. Our work mostly differs from the above contributions by the relative higher distance between software artifacts; moreover, an exact string matching was adopted. However, it is worth noting that the effect of text normalization is similar to adopt a more tolerant string matching. For example, the 3-gram similarity for the terms "program" and "programming" is 76.92 percent; once text normalization is performed, in this particular case, any n-gram similarity corresponds to exact string matching. In other words, text normalization may alleviate the adoption of a more stringent similarity criterion.

Analysis of informal information in the source code (comments and mnemonics for identifiers) can help to associate domain concepts with program fragments and vice-versa. The importance of informal information analysis has been discussed in [12] where an approach based on structures similar to semantic networks has been proposed and the possibility of using some kind of neural networks has been addressed. Comments and mnemonics have an information content with an extremely large degree of variance between systems and, often, between different segments of the same system. Furthermore, this informal information is rarely parsable by Natural Language (NL) grammars because it is mostly based on incomplete sentences combined with abbreviations in a way which is not described by NL grammars. The problem presents some analogies with spoken sentence interpretation [29]. Thus, classification methods based on Artificial Neural Networks (ANN) or the stochastic approach exploited in this paper, rather than formal parsers, are more suitable for analyzing this type of information. These methods relate concepts to patterns of word sequences. As an example, ANNs for source code informal information analysis have been investigated in [35], where a connectionist method, that can be used for design recovery in conjunction with more traditional approaches, is proposed for analyzing the informal information (comments and mnemonics) in programs. The proposed approach uses a combination of top-down domain analysis (the creation of a concept hierarchy by a domain expert, to be used in the construction of the training set) and a bottom-up approach (the analysis of the informal information using the network).

Several software reuse environments use IR to index and retrieve the reusable assets. The RSL [15] system extracts free-text single-term indices from comments in source code files looking for keywords like "author," "date created," etc. REUSE [10] is an information retrieval system which stores software objects as textual documents in view of retrieval for reuse. ALICE [41] is another example of a system that exploits information retrieval techniques for an automatic cataloguing of software componets for reuse. Similarly, CATALOG [24] stores and retrieves C components, each of which is individually characterized by a set of single-term indexing features automatically extracted from natural language headers of C programs. Maarek et al. [31] 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 IBM RISC System/6000 AIX 3 documentation to build a browsing hierarchy which accepts queries expressed in natural language.

6 Conclusion

We have presented an IR method to recover traceability links between code and free text documentation and have applied it to trace C++ and Java source classes to manual pages and functional requirements, respectively. The paper discussed the differences between two IR models, a probabilistic model and a vector space model. The results achieved in the two case studies with both IR models support the hypothesis that IR provides a practicable solution to the problem of semiautomatically recovering traceability links between code and documentation.

In particular, both models achieve 100 percent of recall with almost the same number of documents retrieved. However, the probabilistic model achieves the highest recall values (less then 100 percent) with a smaller number of documents retrieved and then performs better when 100 percent of recall is required. On the other hand, the vector space model shows regular progresses in the recall values when increasing the number of documents retrieved. Also, it requires less effort in the preparation of the query and document representations.

Vector space and probabilistic models need to be further validated on larger systems to assess the relative performance: In our case studies, the probabilistic approach may be preferred when high recall values, but not 100 percent of recall, are required with low cut values (in this cases effort saving may be preferred over the recovery of a complete mapping). However, other researchers on different systems may obtain different results. We did not obtain any statistical evidence to prefer one approach over the other. Also, we did not obtain any evidence to prefer a fixed cut criterion over a variable threshold-based cut criterion as the results are comparable. Future work is required to assess this issue.

Concerning the adequacy of a probabilistic model, a quick comparison between the size of software engineering documentation to natural language processing corpora makes clear that we suffer from a poor training even when unigram models are considered. Sparseness of data and zero frequency may be alleviated by smoothing techniques; different smoothing techniques were tested [21], [37], [55]: in our case studies shift-β gave the best results. In the meantime, we observed a contradictory phenomenon: Smoothing gives very low nonzero probabilities to unseen words; as a result, sometimes, a query is "killed" by the weight of word unseen in the training material. It is worth noting, that the vector space model simply discards these words.

Benchmarking the approaches against a `grep` brute force traceability link recovery demonstrates the benefits of the more sophisticated technologies. As in [31] `grep` is overwhelmed by IR approaches. Our experience clearly indicates that, as the distance between software artifacts increases, the poorer the `grep` performance are.

The best results were achieved once text normalization was applied, thus implicitly introducing a more tolerant matching criterion, according to the approaches presented in [7], [54]. Not surprisingly, in the Albergate case study, where the higher distance between artifacts makes the recovery task more difficult, the effect of text normalization was considerably higher.

However, text normalization in some cases can fail to reconduct software documents and source code to a common vocabulary. Indeed, the key idea of the method presented in this paper is that the application-domain knowledge processed by programmers when writing the code is captured by mnemonic identifiers. Under this assumption, the source code identifier vocabulary shares a significant number of items with the documentation vocabulary. Although this conjecture is supported by the results obtained in both case studies discussed in this paper, the effectiveness of the method becomes less pronounced when the number of common words between the source code component identifiers and the documention items decreases.

This limitation can be overcome by extending our approach as in [8]. Indeed, it often happens that programmers tend to process application-domain knowledge in a consistent way when writing code: Program item names of different code regions related to a given text document are likely to be, if not the same, at least very similar. Under the above assumption, the knowledge of few known existing traceability links can be exploited to recover new traceability links even when the number of common words between the source code component identifiers and the documention is null. In other words, once programmer behavior can be modeled, no matter where the knowledge comes from, few links suffice to recover all the other traceability links [8]. Programmer behaviors can be captured through stochastic modeling: programmers process high-level documentation and domain concepts producing low-level artifacts (e.g., program item names). Once a subset of traceability links is available, for any given link, the joint probability distribution of the text document (words) and a set of linked source code components is estimated together with the document words marginal probability distributions. The estimated probability distributions are used in a Bayesian classifier to score sequences of mnemonics extracted from a not yet *classified* code component (i.e., a component not belonging to the subset of known traceability links). Higher scores suggest the existence of links between the component from which a particular sequence of mnemonics is extracted and the document that generated the marginal probability distribution. Preliminary results have shown that this approach represents a valid alternative to the method presented in this paper in that the case documents and code insist on different vocabularies [8].

As a final remark, we are currently investigating the use of IR to support impact analysis [9], [25], [52] and, particularly, the identification of the components that are thought to be initially affected by a change request. Our idea is to use the maintenance request text to build a query

to retrieve the relevant software documents directly impacted by the maintenance request [3].

ACKNOWLEDGMENTS

The authors would like to thank Professor Aniello Cimitile for his precious suggestions. A special thanks to the `Albergate` programmer team: Claudio Ciccone, Andrea Colombari, Francesca Danzi, Daria Girelli, Roberto Martini, Matteo Meneghini, Andrea Porta, and Paola Vincenti, who kindly provided the source code and the documentation of the system and the requirement to the code traceability matrix. They also want to thank the anonymous reviewers for their comments that helped to improve the original version of this paper.

This research is partially supported by the Agenzia Spaziale Italiana (ASI) grant I/R/091/00 and by the project "Virtual Software Factory," funded by Ministero della Ricerca Scientifica e Tecnologica (MURST) and jointly carried out by EDS Italia Software, University of Sannio, University of Naples "Federico II," and University of Bari.

REFERENCES

[1] G. Adamson and G. Boreham, "The Use of an Associative Measure Based on Character Structure to Identify Semantically Related Pairs of Words and Document Titles," *Information Storage and Retrieval,* vol. 2, no. 10, pp. 253–260 Oct. 1974.

[2] R.C. Angell, G.E. Freund, and P. Willett, "Automatic Spelling Correction Using a Trigram Similarity Measure," *Information Processing and Management,* vol. 19, no. 4, pp. 255–261 Apr. 1983.

[3] G. Antoniol, G. Canfora, G. Casazza, and A. DeLucia, "Identifying the Starting Impact Set of a Maintenance Request: A Case Study," *Proc. European Conf. Software Maintenance and Reeng.,* pp. 227–230, Mar. 2000.

[4] G. Antoniol, G. Canfora, G. Casazza, and A. DeLucia, "Information Retrieval Models for Recovering Traceability Links between Code and Documentation," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 40–49, Oct. 2000.

[5] G. Antoniol, G. Canfora, G. Casazza, A. DeLucia, and E. Merlo, "Tracing Object-Oriented Code into Functional Requirements," *Proc. Eighth Int'l Workshop Program Comprehension,* pp. 227–230, June 2000.

[6] G. Antoniol, G. Canfora, A. DeLucia, and E. Merlo, "Recovering Code to Documentation Links in Object-Oriented Systems," *Proc. IEEE Working Conf. Reverse Eng.,* pp. 136–144, Oct. 1999.

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

[8] G. Antoniol, G. Casazza, and A. Cimitile, "Traceability Recovery by Modeling Programmer Behavior," *Proc. Seventh Working Conf. Reverse Eng.,* pp. 240–247, Nov. 2000.

[9] R.S. Arnold and S.A. Bohner, "Impact Analysis—Towards a Framework for Comparison," *Proc. Int'l Conf. Software Maintenance,* pp. 292–301, 1993.

[10] S.P. Arnold and S.L. Stepowey, "The Reuse System: Cataloging and Retrieval of Reusable Software," *Software Reuse: Emerging Technology,* W. Tracz, ed., 1987.

[11] L. Bain and M. Engelhardt, *Introduction to Probability and Mathematical Statistics.* Belmont, Calif.: Duxburry Press, 1992.

[12] T. Biggerstaff, "Design Recovery for Maintenance and Reuse," *IEEE Computer,* July 1989.

[13] T. Biggerstaff, B. Mitbander, and D. Webster, "The Concept Assignment Problem in Program Understanding," *Proc. Int'l Conf. Software Engineering,* pp. 482–498, May 1993.

[14] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *Int'l J. Man-Machine Studies,* vol. 18, pp. 543–554, 1983.

[15] B.A. Burton, R.W. Aragon, S.A. Bailey, K. Koelher, and L.A. Mayes, "The Reusable Software Library," *Software Reuse: Emerging Technology,* W. Tracz, ed., pp. 129–137, 1987.

[16] G. Caldiera and V.R. Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer,* pp. 61–70, 1991.

[17] G. Canfora, A. Cimitile, M. Munro, "Re2: Reverse Engineering and Reuse Re-Engineering," *J. Software Maintenance—Research and Practice,* vol. 6, pp. 53–72, 1994.

[18] E. Chikofsky and J.C. II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software,* vol. 7, no. 1, pp.13–17, Jan. 1990.

[19] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introductions to Algorithms,* MIT Press, 1990.

[20] T.M. Cover and J.A. Thomas, *Elements of Information Theory.* Wiley Series in Telecommunications, New York: John Wiley & Sons, 1992.

[21] R. DeMori, *Spoken Dialogues with Computers,* Orlando, Fla.: Academic Press, Inc., 1998.

[22] R. Fiutem and G. Antoniol, "Identifying Design-Code Inconsistencies in Object-Oriented Software: A Case Study," *Proc. Int'l Conf. Software Maintenance,* pp. 94–102, Nov. 1998.

[23] W.B. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms.* Englewood Cliffs, N.J.: Prentice-Hall, 1992.

[24] W.B. Frakes and B.A. Nejmeh, "Software Reuse through Information Retrieval," *Proc. 20th Ann. HICSS,* pp. 530–535, Jan. 1987.

[25] M.J. Fyson and C. Boldyreff, "Using Application Understanding to Support Impact Analysis," *J. Software Maintenance—Research and Practice,* vol. 10, pp. 93–110, 1998.

[26] D. Gusfield, *Algorithms on Strings, Trees, and Sequences.* New York: Cambridge Univ. Press, 1997.

[27] D. Harman, "Ranking Algorithms," *Information Retrieval: Data Structures and Algorithms,* pp. 363–392, 1992.

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

[29] R. Kuhn and R.D. Mori, "Learning Speech Semantics with Keyword Classification Trees," *Proc. IEEE Int'l Conf. Acoustics, Speech and Signal Processing,* Apr. 1993.

[30] S. Letovsky, *Cognitive Processes in Program Comprehension: First Workshop.* E. Soloway and S. Iyengar eds., Ablex, 1986.

[31] Y. Maarek, D. Berry, and G. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Trans. Software Eng.,* vol. 17, no. 8, pp. 800–813, 1991.

[32] A.V. Mayrhauser and A. Vans, "From Program Comprehension to Tool Requirements for an Industrial Environment," *Proc. IEEE Workshop Program Comprehension,* pp. 78–86, 1993.

[33] A.V. Mayrhauser and A. Vans, "Dynamic Code Cognition Behaviours for Large Scale Code," *Proc. Third IEEE Workshop Program Comprehension,* pp. 74–81, 1994.

[34] A.V. Mayrhauser and A.M. Vans, "Identification of Dynamic Comprehension Processes During Large Scale Maintenance," *IEEE Trans. Software Eng.,* vol. 22, no. 6, pp. 424–437, June 1996.

[35] E. Merlo, I. McAdam, and R.D. Mori, "Source Code Informal Information Analysis Using Connectionist Models," *Proc. Int'l Joint Conf. Artificial Intelligence,* pp. 1339–1344, Sept. 1993.

[36] G.C. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Source and High-Level Models," *Proc. Third ACM Symp. Foundations of Software Eng.,* 1995.

[37] H. Ney and U. Essen, "On Smoothing Techniques for Bigrambases Natural Language Modelling," *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing,* vol. 12, no. 11, pp. 825–828, 1991.

[38] P.W. Oman and C.R. Cook, "The Book Paradigm for Improved Maintenance," *IEEE Software,* vol. 7, no. 1, pp. 39–45, Jan. 1990.

[39] N. Pennington, *Comprehension Strategies in Programming. In: Empirical Studies of Programmers: Second Workshop.* G.M. Olsen, S. Sheppard, and S. Soloway eds., Nordwood, Englewood Cliffs, N.J., Ablex, 1987.

[40] N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psychology,* vol. 19, pp. 295–341, 1987.

[41] M. Pighin, "Tracing Object-Oriented Code into Functional Requirements," *Proc. Fifth Conf. Software Maintenance and Reeng.,* pp. 196–199, Mar. 2001.

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

[43] D.L. Potter, J. Sinclair, D. Till, *An Introduction to Formal Specification and Z.* Englewood Cliffs, N.J.: Prentice-Hall, 1991.

[44] B. Ramesh and V. Dhar, "Supporting Systems Development Using Knowledge Captured During Requirements Engineering," *IEEE Trans. Software Eng.*, vol. 9, no. 2, pp. 498–510, June 1992.
[45] C. Rich and R. Waters, *The Programmer's Apprentice.* Reading, Mass.: Addison-Wesley, 1990.
[46] R. Rosenfeld, "Adaptive Statistical Language Modeling: A Statistical Approach," PhD thesis, School of Computer Science, Carnegie Mellon Univ., Apr. 1994.
[47] S. Rugaber and R. Clayton, "The Representation Problem in Reverse Engineering," *Proc. Working Conf. Reverse Eng.*, pp. 8–16, 1993.
[48] G. Salton and C. Buckley, "Term-Weighting Approaches in Automatic Text Retrieval," *Information Processing and Management,* vol. 24, no. 5, pp. 513–523, 1988.
[49] M. Sefika, A. Sane, and R.H. Campbell, "Monitoring Compliance of a Software System with Its High-Level Design Models," *Proc. Int'l Conf. Software Eng.*, pp. 387–396, 1996.
[50] B. Shneiderman and R. Mayer, "Syntactic/Semantic Interactions in Programmer Behaviour: A Model and Experimental Results," *Int'l J. Computer and Information Sciences,* vol. 8, no. 3, pp. 219–238, Mar. 1979.
[51] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. Software Eng.*, vol. 10, no. 5, pp. 595–609, 1994.
[52] R.J. Turver and M. Munro, "An Early Impact Analysis Technique for Software Maintenance," *J. Software Maintenance—Research and Practice,* vol. 6, no. 1, pp. 35–52, 1994.
[53] I. Vessey, "Expertise in Debbugging Computer Programs: A Process Analysis," *Int'l J. Man-Machine Studies,* vol. 23, pp. 459–494, 1985.
[54] J. Weidl and H. Gall, "Binding Object Models to Source Code," *Proc. 22nd Computer Software and Applications Conf. (COMPSAC '98),* pp. 26–31, Aug. 1998.
[55] I.H. Witten and T.C. Bell, "The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression," *IEEE Trans. Information Theory,* vol. 37, pp. 1085–1094, 1991.

Giuliano Antoniol received the doctoral degree in electronic engineering from the University of Padua in 1982. He worked at Irst for 10 years where he leads the the Irst Program Understanding and Reverse Engineering (PURE) Project team. He is member of the editorial board of *Journal Software Testing Verification and Reliability*. He is currently an associate professor at the University of Sannio, Faculty of Engineering, where he works in the area of software metrics, process modeling, software evolution, and maintenance. He is a member of the IEEE and the IEEE Computer Society.

Gerardo Canfora received the Laurea degree in electronic engineering from the University of Naples, Federico II, Italy, in 1989. He is currently a full professor of computer science at the Faculty of Engineering and the Director of the Research Centre on Software Technology (RCOST) of the University of Sannio in Benevento, Italy. From 1990 to 1991, he was with the Italian National Research Council (CNR). During 1992, he was at the Department of Informatica e Sistemistica of the University of Naples, Federico II, Italy. From 1992 to 1993, he was a visiting researcher at the Centre for Software Maintenance of the University of Durham, UK. In 1993, he joined the Faculty of Engineering of the University of Sannio in Benevento, Italy. He has served on the program committees of a number of international conferences; he was program cochair of the 1997 International Workshop on Program Comprehension and of the 2001 International Conference on Software Maintenance. His research interests include software maintenance, program comprehension, reverse engineering, reuse, reengineering, migration, workflow management, and document management. He is a member of the IEEE and the IEEE Computer Society.

Gerardo Casazza received the Laurea degree in computer engineering from the University of Salerno, Italy, in 1997. From 1998 to 2001, he was a PhD student in electronic engineering and computer science at the University of Naples, Federico II. In October 2001, he completed his PhD program concerned with software maintenance in cooperative environments. His research interests include cooperative supports for software maintenance, impact analysis, reverse engineering, and program comprehension. He is a member of the IEEE and the IEEE Computer Society.

Andrea De Lucia received the Laurea degree in computer science from the University of Salerno, Italy, in 1991, the MSc degree in computer science from the University of Durham, UK, in 1995, and the PhD degree in electronic engineering and computer science from the University of Naples, Federico II, Italy, in 1996. He is currently an associate professor of computer science in the Faculty of Engineering of the University of Sannio in Benevento, Italy. Previously, he was with the Department of Informatica e Applicazioni of the University of Salerno, Italy, and with the Department of Informatica e Sistemistica of the University of Naples, Federico II, Italy. From 1994 to 1995, he was a visiting researcher at the Centre for Software Maintenance of the University of Durham, UK. In 1996, he joined the Faculty of Engineering of the University of Sannio in Benevento, Italy. He serves on the program and organizing committees of several international conferences and was program cochair of the 2001 International Workshop on Program Comprehension. His research interests include software maintenance, reverse engineering, reuse, reengineering, migration, program comprehension, workflow management, document management, and visual languages. He is a member of the IEEE and the IEEE Computer Society.

Ettore Merlo received the PhD degree in computer science from McGill University (Montreal) in 1989 and the Laurea degree (summa cum laude) from University of Turin (Italy) in 1983. He was the lead researcher of the software engineering group at Computer Research Institute of Montreal (CRIM) until 1993 when he joined Ecole Polytechnique de Montreal where he is currently an associate professor. His research interests are in software analysis, software reengineering, user interfaces, software maintenance, artificial intelligence, and bio-informatics. He has collaborated with several industries and research centers in particular on software reengineering, clone detection, software quality assessment, software evolution analysis, testing, architectural reverse engineering, and dynamic genetic linkage analysis. He is a member of the IEEE and the IEEE Computer Society.