Incremental Latent Semantic Indexing for Automatic Traceability Link Evolution Management

Hsin-yi Jiang[1], Tien N. Nguyen[2], Ing-Xiang Chen[3], Hojun Jaygarl[1], and Carl K. Chang[1]
[1]Computer Science Department, Iowa State University, USA
[2]Electrical and Computer Engineering Department, Iowa State University, USA
[3] Computer Science and Engineering Department, Yuan Ze University, Taiwan

***Abstract*—Maintaining traceability links among software artifacts is particularly important for many software engineering tasks. Even though automatic traceability link recovery tools are successful in identifying the semantic connections among software artifacts produced during software development, no existing traceability link management approach can effectively and automatically deal with software evolution. We propose a technique to automatically manage traceability link evolution and update the links in evolving software. Our novel technique, called *incremental Latent Semantic Indexing* (iLSI), allows for the fast and low-cost LSI computation for the update of traceability links by analyzing the changes to software artifacts and by re-using the result from the previous LSI computation before the changes. We present our iLSI technique, and describe a complete automatic traceability link evolution management tool, *TLEM*, that is capable of interactively and quickly updating traceability links in the presence of evolving software artifacts. We report on our empirical evaluation with various experimental studies to assess the performance and usefulness of our approach.**

I. Introduction

Extensive effort in the software engineering community has been brought forth to improve the explicit semantic connections among software artifacts, especially between documentation and source code. Those semantic connections are often called *traceability links*. Recovering traceability links in legacy systems is particularly important and helpful for a variety of software engineering tasks. These tasks include program understanding, requirement tracing and validation, analysis for the integrity and completeness of the implementation, monitoring and controlling the impact of changes, reverse engineering for reuse and re-development, and many other software maintenance tasks. For these reasons, much effort has been spent on building automatic tools for traceability link recovery (TLR) from software systems [1], [17], [21], [23]. TLR technology to scan the software artifacts including documentation and source code, and then to identify the semantic connections is advanced and successful [34].

Unfortunately, many of these methods are inherently unsuited to being applied to *evolving* software systems. Simply recovering the traceability links is not sufficient due to the evolutionary nature of software. The links are probably only valid at a certain state or certain version of the software. A link represents a semantic relation between software documents. However, as software evolves during the development and maintenance process, software artifacts change as well. Therefore, changes to software artifacts might potentially *invalidate* the traceability links. At the same time, some other traceability links might not be affected by the changes. In other words, traceability links also evolve and need to be up-to-date during software development. To cope with software evolution, a naive approach for the automatic update of traceability links is to re-run a TLR tool, which is a computationally costly solution for *interactive* use during software development. Furthermore, after a TLR tool is re-run, changes to the links themselves can not be automatically captured. For example, users must manually compare the results from a TLR tool before and after the changes to figure out which links are added, deleted, etc.

In general, this represents a fundamental issue with existing traceability link management approaches. No existing approach can *effectively* and *automatically* deal with software changes. They require either human intervention or users' feedbacks. Although traceability link recovery tools are automated, they can not automatically evolve the links in an evolving software system. Traceability links are needed to be not only recovered but also *automatically managed* as the software changes. This is not a small task for developers without the *automatic* tools that support link evolution management and link update. Without dedicated and automatic support, developers who want to maintain and evolve traceability links for their tasks face the following challenges:

- Current TLR technology produces traceability links, which are easily *invalidated* as soon as changes occur to documentation and source code. That is, TLR does not adapt well with software evolution.
- Changes to software artifacts can be managed with a version control system. However, traceability links are not *automatically managed* and *updated* when software evolves. Therefore, developers must use ad hoc methods to maintain links or manually update them.
- When changes occur, invalidated links, which no longer connect the right documentation or source code, cannot be updated without the re-run of a TLR process.
- If a TLR tool is not re-run, existing information about traceability links cannot be reused in future tasks.

In this paper, we propose a technique to automatically manage traceability link evolution and update the links in evolving software. Our automatic link updating technique relies on a novel incremental version of the well-known Latent Semantic Indexing (LSI) algorithm that has been used for

Systems	Source Files	Documentation	Number of Terms	LSI Matrix	Pre-processing Time	Total TLR Time
Jigsaw_2.6.6	954	1455 (files)	2240	2240 x 2409	12 sec	39 minutes
SC_01_07	2231	751 (sections)	3645	3645 x 2982	20 sec	97 minutes
Apache httpd_2.2.4	4874	994 (files)	5112	5112 x 5868	34 sec	8 hours 41 mins
JDK1.5	3280	2814 (files)	12798	8000 x 6094	62 sec	11 hours 12 mins

TABLE I
TIME COMPLEXITY OF LSI FOR TRACEABILITY LINK RECOVERY WITH COSINE THRESHOLD OF 0.7

TLR [21]. This novel technique extends our preliminary work on *Incremental LSI* (iLSI) [14]. Our technique allows for the fast and low-cost computation of traceability links by re-using the result from previous LSI computation for TLR before the changes. iLSI avoids the full cost of LSI computation for TLR by analyzing the changes to the software artifacts in a system, then deriving the changes to the set of traceability links.

We developed a complete automatic traceability link evolution management tool, called *TLEM*, which was integrated into Eclipse environment. The tool operates in connection with the Eclipse editor and with a back-end software configuration management (SCM) tool, named Molhado [24]. Molhado is used to store information on traceability links and software artifacts at different versions. Initially, TLEM uses the traditional LSI technique to derive traceability links and store them into the SCM repository. For a subsequent version of software, before a check-in or on users' command, TLEM analyzes the changes and the new version of software artifacts and uses the incremental LSI algorithm to update the links. Using SCM repository for link storage, TLEM provides for developers the tracking support of traceability links through different versions of a system. We conducted an empirical evaluation on its great reduction of time complexity (by 100–1500 times) and on its maintenance of high precision and recall values for TLR.

The key contributions of this paper include a novel traceability link updating technique using our novel incremental LSI algorithm (iLSI) that can cope with software changes, the automatic traceability link evolution management tool TLEM, and a number of case studies and an empirical evaluation on the usefulness of this approach. Importantly, with the great reduction in time complexity, iLSI allows TLEM to be *interactively* used in the Eclipse editor during software development to automatically maintain and update the links. Our key departure point from existing traceability approaches is the leverage of an advanced TLR technique to provide the support for *automatic* traceability link evolution management that can cope with *software evolution*.

In Section 2, we present a real example of traceability link evolution and explain the difficulties associated with TLR. Sections 3-5 describe the iLSI algorithm and the computational complexity analysis for it. Section 6 presents our tool TLEM. Our empirical evaluation is in Section 7. We discuss related work in Section 8 and conclude in Section 9.

II. MOTIVATION

We illustrate the present state of the practice for automatic TLR and the need for an automatic traceability link evolution management technique with an empirical study.

The first case study is of our own software system, *Software Concordance/Molhado* (SC) [24], that we have been developing for more than five years with several versions and variants. SC has about 102kLOC in Java with 2231 source code files. There are 25 files of documentation (design documents and API manuals) with 751 sections of texts and the total size of 2.61MB. A developer was required to understand and to add new functionality into SC. He wanted to recover the traceability links from design documents and manuals to source code. He decided to run an automatic TLR tool that was built by our lab using the well-known LSI technique described in [19], [21]. LSI's parameters that were set for the LSI execution are as follows: number of terms (3645), number of documents (2231 + 751 = 2982), LSI dimensions (3645 x 2982), K = 10%, and cosine threshold of 0.7. After approximately *97 minutes* (on Windows XP, Intel Pentium 4 3Ghz, 1GB RAM), the TLR tool completed and returned a list of 482 links. This is too computationally expensive to be interactively used during daily software development.

We also conducted an experiment to study the time complexity and the scalability of the LSI technique for traceability link recovery on large software systems. Table I shows the results. The pre-processing time includes the time to parse the documentation and source code files, and to build the LSI's Term-Document matrix (Section III). We observed that the time complexity to run TLR on a large system is extremely high due to the computational cost of the required Singular Value Decomposition (SVD) process [6] of the LSI technique on a large LSI's Term-Document matrix. It is also observed that the pre-processing time is relatively small compared to the total TLR time, and most of the time is spent on the SVD computation for the LSI matrix. In brief, LSI is computationally expensive for TLR in interactive use.

III. iLSI APPROACH FOR LINK UPDATE

For the automatic update of traceability links after a change, we introduce an incremental approach to the well-known LSI technique [6]. In this section, we present the traditional LSI method for TLR and our setting of LSI parameters in our tool.

IMPORTANT DEFINITIONS

Notations: An uppercase letter (e.g. X) denotes a matrix. The [i,j] entry of matrix X is denoted by $X[i,j]$ or x_{ij}. A matrix with m rows and n columns is said to be the size of $m \times n$. "." is used for the product between two matrices or two vectors, depending on the context of the operands. A vector can be considered as a matrix with one column.

Square matrix: is a matrix with the number of rows equal to the number of columns.

	$\mathbf{doc_1}$	$\mathbf{doc_2}$	...	$\mathbf{doc_d}$
$\mathbf{term_1}$	x_{11}	x_{12}	...	x_{1d}
$\mathbf{term_2}$	x_{21}	x_{22}	...	x_{2d}
...	...	...	...	...
$\mathbf{term_t}$	x_{t1}	x_{t2}	...	x_{td}

TABLE II
TERM-DOCUMENT MATRIX X_1

Diagonal matrix: is a square matrix with zeroes in its non-diagonal entries.

Identity matrix: is a diagonal matrix whose diagonal entries are all ones. It is denoted by I_m where m is its width or height. When the dimensions are clear from the context, the subscript will be disregarded.

The **transpose** of a matrix X, denoted by X', is a matrix whose rows are the columns of X. The columns of X are **orthonormal** if $X'.X = I$. A real matrix X is **orthogonal** (**unitary**) if $X'.X = X.X' = I$. **Inner product** of two vectors d_1 and d_2 is defined as the value $d_1'.d_2$. The cosine inner product of d_1 and d_2 is the length-normalized inner product: $cos(d_1, d_2) = (d_1'.d_2)/(|d_1|.|d_2|)$.

In our method, a software system is considered as a set of documents including *source code* and *documentation*. A source code document *doc* is any contiguous set of lines of source code and/or text [21]. Typically, a source code document is a file of source code or a program entity such as a class, function, interface, etc. A non-source-code document is any contiguous set of lines of text from documentation (e.g. design or requirement specifications). Typically, it is a section, a chapter, or an entire file of texts. A file can be composed of multiple documents. Now, let us describe three important phases that are involved in our LSI method for TLR.

A. Pre-processing Phase

The first phase is for pre-processing. It includes the tasks of parsing source code and documentation, building the corpus, and creating Term-Document matrix (Table II). This phase is carried out in a similar way as the process described in [21].

In general, to construct the corpus, the source files and the documentation need to be broken up into the proper granularity to define the *documents*, which will then be represented as vectors [6]. The length of a vector representing for a document is equal to the number of *words* (also called *terms*) in the corpus. Grammatical terms such as "a", "the", "and", etc are filtered [21]. The value of the vector at a position corresponding to a term can be either equal to the frequency of that term occurring in that document or equal to a weight value. That weight value is computed based on a *term significance measure algorithm* such as Term frequency - Inverse document frequency algorithm (Tf-Idf) [30]. This weight is a statistical measure used to evaluate how important a term is to a document in the collection or the corpus. Details of this computation are given in [30]. In TLEM, we chose Tf-Idf for high accuracy [21]. Putting all document vectors together as columns, we have *Term-Document matrix* X_1 (Table II). In matrix X_1, the cells (x_{ij}s) are Tf-Idf significance measures.

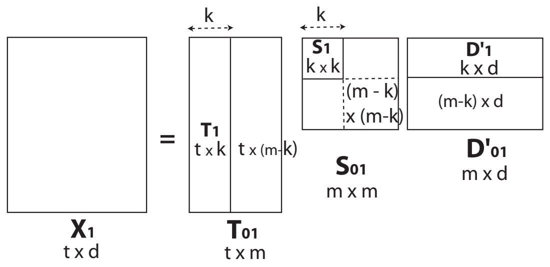


Fig. 1. Singular Value Decomposition for X_1

B. Singular Value Decomposition Phase

As illustrated in Section II, this matrix X_1 is usually very large. Therefore, the LSI technique for TLR requires the use of Singular Value Decomposition (SVD) [6]. This is the second phase in the LSI method. According to the SVD computation of the LSI method [6], X_1 (a $t \times d$ matrix) can be decomposed into the product of three matrices:

$$X_1 = T_{01} \cdot S_{01} \cdot D_{01}'$$

S_{01} is the diagonal matrix of singular values. T_{01} is the matrix of eigenvectors of the square symmetric matrix $X_1 \cdot X_1'$ (X_1' is the *transpose* of X_1). T_{01} is unitary. Since $X_1 \cdot X_1'$ is real, T_{01} has orthonormal columns and rows, i.e., $T_{01}' \cdot T_{01} = I$ and $T_{01} \cdot T_{01}' = I$. Meanwhile, D_{01} is the matrix of eigenvectors of $X_1' \cdot X_1$. Similarly, D_{01} also has orthonormal columns and rows, i.e., $D_{01}' \cdot D_{01} = I$ and $D_{01} \cdot D_{01}' = I$. Note that T_{01} is a $t \times m$ matrix, S_{01} is a $m \times m$ matrix, and D_{01}' is a $m \times d$ matrix ($t, d, m \in N$).

The beauty of SVD is that it enables a simple strategy for optimal approximate fit using smaller matrices, thus, reducing the computational cost. The key idea is that if the singular values in S_{01} can be ordered by size, the first k largest may be kept and the remaining smaller ones set to zero. Since zeroes were introduced into S_{01}, the representation can be simplified by deleting the zero rows and columns of S_{01} to obtain a new diagonal matrix S_1, and then by deleting the corresponding columns of T_{01} and D_{01} to obtain T_1 and D_1 such that

$$X_1 \approx X_1^a = T_1 \cdot S_1 \cdot D_1'$$

The matrices involved in SVD are illustrated in Figure 1. If all singular values in S_{01} are kept, T_1 and D_1 will be equal to T_{01} and D_{01}, respectively. Note that the sizes of S_1, D_1, and T_1 are relatively small since we keep only a small set of singular values. We have T_1 is the $t \times k$ matrix, S_1 is the $k \times k$ diagonal matrix, and D_1' is the $k \times d$ matrix. The product of these matrices is a matrix X_1^a, which is only approximately equal to X_1. It was shown that the new matrix X_1^a is the matrix of rank k, which is closest in the least squares sense to X_1 [6]. X_1^a is the rank-k model with the best possible least-squares fit to X_1 and with less noise.

Since m is the rank of S_{01} (also X_1), it is obvious that for any k we choose, $k \leq m$. For our purpose of reducing the computational complexity, it is assumed that k can be chosen such that $k \ll m$. Let us call the process of selecting k and deriving S_1, T_1, and D_1 *the reduction step* (Figure 1).

C. Link Recovery Phase

The third phase is to use the matrix X_1^a for the purpose of identifying the traceability links. To apply for traceability link recovery, we want to compute the similarity between two documents. The similarity between two documents is equal to the cosine inner product of two respective column vectors of the approximate matrix X_1^a. If it is over a threshold, we consider that those two documents are semantically linked to each other. Note that the product matrix $X_1^{a\prime} \cdot X_1^a$ contains document-to-document cosine inner products. Thus, the values of the cells in that product matrix represent the semantic similarity, and are used to determine traceability links among documents. TLEM uses 0.7 as the default threshold as suggested in [21].

IV. Incremental LSI Technique

As described in Section III, a TLR process using LSI technique has three phases. The first one is the pre-processing phase. The experiments described in Section II showed that the pre-processing time is very small compared to the total TLR time. The second phase is the SVD computation, which is the most expensive process, especially for large systems because it takes a long time to singularly decompose large Term-Document matrices. The matrix X_1^a is the result of the SVD process. The third phase (identification of the links) is computationally inexpensive, because it involves only a few matrix multiplications (see Section III). Therefore, our method aims to reduce the cost of the SVD computation for the new Term-Document matrix after changes occur. The matrices resulted from the SVD computation for X_1 (i.e. T_{01}, S_{01}, and D_{01}) and the set of traceability links are maintained in the repository and/or in the memory (for the case of interactive editing sessions). Only the resulting matrices from the immediately preceding SVD computation need to be maintained.

A. New Term-Document Matrix X_2

Let us denote the time/version of the system before and after the change by v_1 and v_2, respectively. Remind that X_1 is the Term-Document matrix before the change, and the resulting matrices of the LSI computation for X_1 are maintained. Let us denote the new Term-Document matrix after the change by X_2. Firstly, the parsing and the computation of Tf-Idf significance measures are performed on version v_2 of the system. After learning from the SCM system and/or the Eclipse environment which documents were added, removed, renamed, or modified (see [7]), TLEM constructs the matrix X_2 as follows. For **terms** (i.e. rows):

- A term that still appears at v_2 is positioned in X_2 at the same row as in X_1.
- A term that was deleted out of entire system at v_2 (but existed at v_1) is still positioned at the same row as in X_1. However, the corresponding cells at that row will be filled with zeroes. That is, the significance measure of the term in a document is zero.
- A term that was added at v_2 is positioned below the rows of existing and deleted terms mentioned above.

For **documents** (source code, documentation), i.e. columns:

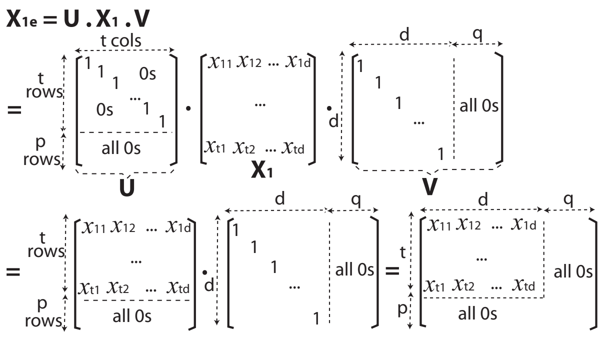


Fig. 2. Expanded Matrix X_{1e}

- A document that still exists at v_2 is positioned in X_2 at the same column as in X_1.
- A document is assigned unique identifier across versions. Therefore, a *renamed* document will be positioned at the same column as in X_1.
- A document that was deleted out of entire system at v_2 (but existed at v_1) is still placed at the same column as in X_1. However, that entire column will be filled with zeroes as in the case of deleted terms.
- A document that was added at v_2 is positioned on the right side of all the columns corresponding to existing and deleted documents.

Due to the nature of Tf-Idf algorithm [30], a vector corresponding to an un-changed document still needs to be updated. For example, an element of that vector, which is the significance value of a term with respect to that document, depends on the number of documents where the term occurs.

Note that X_1 is an $t \times d$ matrix. With the aforementioned method of constructing X_2, the height and the width of X_2 are always larger than or equal to those of X_1, respectively. Let us assume that X_2 is an $(t+p) \times (d+q)$ matrix. Also, note that no links will be computed for deleted documents.

B. Expanded Matrix X_{1e} and its SVD computation

First of all, we want to make X_1 to have the same size as X_2 by transforming X_1 into its **expanded matrix** X_{1e}. The process is described in Figure 2. We take two matrices U of the size $(t+p) \times t$ and V of the size $d \times (d+q)$. Basically, U and V are the extensions of the identity matrices I_t and I_d with p rows and q columns of all zeroes, respectively (see Figure 2). The expanded matrix X_{1e} is the resulting product of $U \cdot X_1 \cdot V$. As shown in Figure 2, X_{1e} is an expanded matrix of X_1 with the size of $(t+p) \times (d+q)$.

As a result of SVD, we have $X_1 = T_{01} \cdot S_{01} \cdot D_{01}'$. Thus:

$$\begin{aligned} X_{1e} &= U \cdot X_1 \cdot V = U \cdot T_{01} \cdot S_{01} \cdot D_{01}' \cdot V \\ &= (U \cdot T_{01}) \cdot S_{01} \cdot (V' \cdot D_{01})' \\ &= T_{01e} \cdot S_{01} \cdot D_{01e}' \quad (*) \end{aligned}$$

where $T_{01e} = U \cdot T_{01}$ and $D_{01e} = V' \cdot D_{01}$. Let us examine

$$\begin{aligned} T_{01e} \cdot T_{01e}' &= (U \cdot T_{01}) \cdot (U \cdot T_{01})' \\ &= U \cdot T_{01} \cdot T_{01}' \cdot U' \\ &= U \cdot I \cdot U' = U \cdot U' = I \end{aligned}$$

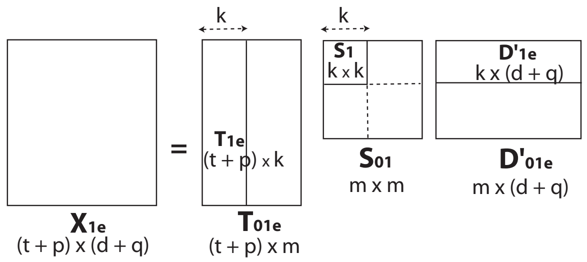


Fig. 3. Singular Value Decomposition for X_{1e}

Similarly, we have $T'_{01e} \cdot T_{01e} = I$, $D_{01e} \cdot D'_{01e} = I$, and $D'_{01e} \cdot D_{01e} = I$ because $T'_{01} \cdot T_{01} = T_{01} \cdot T'_{01} = I$ and $D'_{01} \cdot D_{01} = D_{01} \cdot D'_{01} = I$. Thus, Equation (*) gives the result of a Singular Value Decomposition for the matrix X_{1e}, and $T_{01e} = U \cdot T_{01}$ and $D_{01e} = V' \cdot D_{01}$. In other words, we can **easily compute a SVD for X_{1e} from the result of the SVD computation for X_1**. In fact, with the definitions of U and V in Figure 2, to compute T_{01e} and D_{01e} for X_{1e}, one just needs to expand p and q rows of all zeroes to T_{01} and D_{01}, respectively. The diagonal matrix S_{01} of the SVD computation for X_{1e} is the same as the one for X_1.

From the SVD for $X_{1e} = T_{01e} \cdot S_{01} \cdot D'_{01e}$, we can perform the reduction by keeping k largest singular values as in the case of X_1. See Figure 3 for SVD of X_{1e}. That is, zeroes are introduced into S_{01}; the representation can be simplified by deleting the zero rows and columns of S_{01} to obtain a new diagonal matrix S_1; and then deleting the corresponding columns of T_{01e} and D_{01e} to obtain T_{1e} and D_{1e} such that

$$X_{1e} \approx X^a_{1e} = T_{1e} \cdot S_1 \cdot D'_{1e}$$

C. SVD for new Term-Document matrix X_2

So far, we have shown that the SVD computation for X_{1e} can be derived from the SVD computation of X_1. This section demonstrates how the computational cost of SVD for X_2 can be reduced. Let us examine the following product:

$$P = T'_{1e} \cdot X_2 \cdot D_{1e}$$

Since T'_{1e} is a $k \times (t+p)$ matrix, X_2 is a $(t+p) \times (d+q)$ matrix, and D_{1e} is a $(d+q) \times k$ matrix, P's dimension is $k \times k$. Thus, P is a square matrix. If we apply SVD onto P:

$$P = T'_{1e} \cdot X_2 \cdot D_{1e} = T_{temp} \cdot S_2 \cdot D'_{temp} \quad (1)$$

$$\Longrightarrow T_{1e}.T'_{1e}.X_2.D_{1e}.D'_{1e} = T_{1e}.T_{temp}.S_2.D'_{temp}.D'_{1e} \quad (2)$$

$$\Longrightarrow X_2 = T_{1e}.T_{temp}.S_2.D'_{temp}.D'_{1e} \quad (3)$$

Let $T_{1e} \cdot T_{temp} = T_{02}$, and $D_{1e} \cdot D_{temp} = D_{02}$, then

$$X_2 = T_{02} \cdot S_2 \cdot D'_{02} \quad (4)$$

We have $T_{02} \cdot T'_{02} = (T_{1e} \cdot T_{temp}) \cdot (T_{1e} \cdot T_{temp})' = T_{1e} \cdot T_{temp} \cdot T'_{temp} \cdot T'_{1e} = I$ because $T_{temp} \cdot T'_{temp} = I$ and $T_{1e} \cdot T'_{1e} = I$. Similarly, $T'_{02} \cdot T_{02} = I$, $D_{02} \cdot D'_{02} = I$, and $D'_{02} \cdot D_{02} = I$ because of the orthonormal property of T_{1e}, D_{1e}, T_{temp}, and D_{temp}.

Therefore, Equation (4) gives the result of SVD for X_2.

D. Automatic Link Update Algorithm

Equations (*), (4), and others show us that using the matrices T_{01}, S_{01}, and D_{01} obtained from the previous SVD computation for X_1, the SVD computation for X_2 is reduced to the SVD computation for P, a much smaller matrix derived from both X_1 and X_2, and a few matrix multiplications. Based on those equations of iLSI, an automatic traceability link update algorithm is presented (Figure 4).

1) Obtain new Term-Document matrix X_2 (Section IV-A).
2) Compute $T_{01e} = U \cdot T_{01}$ and $D_{01e} = V' \cdot D_{01}$. Figure 2 shows U and V. Then, derive T_{1e} and D_{1e} with selected k by deleting corresponding columns in T_{01e} and D_{01e} (i.e. the reduction step in Figure 3).
3) Compute $P = T'_{1e} \cdot X_2 \cdot D_{1e}$.
4) Perform SVD for the small matrix P to get $P = T_{temp} \cdot S_2 \cdot D'_{temp}$. The matrix P is $k \times k$.
5) Compute $T_{02} = T_{1e} \cdot T_{temp}$, and $D_{02} = D_{1e} \cdot D_{temp}$. Then, derive T_2 and D_2 from T_{02} and D_{02}, respectively by the reduction step with selected k.
6) Compute $X^a_2 = T_2 \cdot S_2 \cdot D'_2$. Compute the product matrix $X^{a\prime}_2 \cdot X^a_2$, which contains similarity measures between any two documents.
7) Traceability links between documents are determined with a chosen threshold. Then, the link set is updated.

Fig. 4. Automatic Link Update with incremental LSI (iLSI)

The result of step 1, which is X_2, is the result from the pre-processing phase and the arrangement (Section IV-A) after the changes occur. To reduce the time cost of pre-processing for the interactive link updating (even though it is relatively small), the results for lexical analysis of un-changed documents are re-used. In step 2, T_{01e} and D_{01e} are derived from T_{01} and D_{01} of the previous SVD computation for X_1. Step 3 is given by Equation (1). Note that the cost of the SVD computation for matrix P is much lower than that of X_2 because the size of P is $k \times k$. Steps 4 and 5 are the results of Equations (1)–(4). The product matrix $X^{a\prime}_2 \cdot X^a_2$ in step 6 contains similarity measures between any two documents, thus, it is used to derive traceability links (see Section III-C). In step 7, based on the chosen threshold, the currently maintained set of traceability links is updated, and later, will be checked into the SCM repository along with the changes to documents.

V. Computational Complexity

The TLR process using the traditional LSI method consists of three phases: pre-processing, SVD, and link recovery. As illustrated earlier, time complexity for pre-processing and link recovery is very small compared to the cost of SVD. In the traditional LSI method, SVD computation must be performed on the new Term-Document matrix, which is usually very large in large-scale software systems. Meanwhile, our incremental LSI method also requires the similar three phases, but focuses on reducing the computational cost of the SVD phase. For comparing two methods, let us analyze the computational

Steps	Complexity
Matrix Multiplication	$s^2r + r^2s$
Householder Transformation	$s^4 - \frac{5}{3}s^3 + r^4 - \frac{5}{3}r^3$
QR Algorithm	$\frac{s^3}{6} + \frac{s^2}{2} + \frac{r^3}{6} + \frac{r^2}{2}$
Matrix Reduction step	0

TABLE III
COMPLEXITY OF TRADITIONAL LSI/SVD

Steps	Complexity
Compute T_{1e} and D_{1e}	0
Compute P	$rk(s+k)$
SVD for P	Table V
Compute T_2 and D_2	$k^2(s+r)$

TABLE IV
COMPLEXITY OF INCREMENTAL LSI/SVD

Steps	Complexity
Matrix Multiplication	$2k^3$
Householder Transformation	$2k^4 - \frac{10}{3}k^3$
QR Algorithm	$\frac{k^3}{3} + k^2$
Matrix Reduction step	0

TABLE V
COMPLEXITY OF SVD FOR MATRIX P $(k \times k)$

complexity of the SVD phase because both methods require the same two other phases and the time cost for the tasks in those two phases is small compared to that of the SVD phase.

A. Traditional LSI Method in Batch Mode

In TLEM, we chose to implement SVD function using a well-known and fast SVD approach among existing ones. The approach is the combination of Householder method and QR method [13]. SVD process is summarized as follows:

1) Compute $M = X_2 \cdot X_2'$ and $N = X_2' \cdot X_2$.
2) Compute the eigenvalues and matrices of eigenvectors of both M and N. Since M and N are symmetric, the next steps are as follows:
   a) Reduce a symmetric matrix to tridiagonal matrix using Householder method [13].
   b) Apply QR algorithm [13] to that tridiagonal matrix, compute the matrix for eigenvectors at each iteration, and derive T_{02}, S_{02}, and D_{02}.
   c) With selected k, apply the reduction step to T_{02}, S_{02}, and D_{02} to get T_2, S_2, and D_2 (Figure 1).

Note that X_2 is a $(t+p) \times (d+q)$ matrix. Let us denote $s = t+p$ and $r = d+q$. In this analysis, we use the approach that analyzes the number of operations required to perform a particular computation. This approach has been used in the computational complexity analysis for tasks involving matrix operations [13]. For example, the product of two vectors $u' \cdot v$, where u and v are $n \times 1$ vectors, is considered to have a complexity of n operations. Firstly, we analyze the complexity of the computations that involve the matrix M (the result for N will be similarly derived):

- The complexity in term of the number of multiplication operations for doing the matrix multiplication step for M (i.e. Step 1 above) is s^2r.
- In Householder transformation step (Step 2a) for M, the complexity is $s^3(s-2) + \sum_{i=1}^{s-2} i^2 \approx s^4 - \frac{5}{3}s^3$ [13].
- In the QR algorithm step (Step 2b) for M, the cost of finding Q and R is $3(t+p)$ [13], and the complexity of computing the eigenvector matrix is $\sum_{i=1}^{s} \frac{i(i+1)}{2} \approx \frac{s^3}{6} + \frac{s^2}{2}$ for each iteration.
- Step 2c (reduction) does not cost multiplications.

Taking N into consideration, Table III summarizes the computational complexity of the traditional LSI method for TLR running in a batch mode after the changes occur.

B. Complexity of Incremental LSI Method

Let us analyze the computational complexity of our incremental LSI method described in Figure 4. For comparison, let us examine the steps 2-5 because steps 1, 6, and 7 are also required by the traditional LSI technique.

Step 2 in Figure 4 does not require any multiplication operation. The reasons are: 1) to compute T_{01e} and D_{01e}, T_{01} and D_{01} are expanded with p and q rows of all zeroes at the end, respectively; and 2) the reduction step to get T_{1e} and D_{1e} does not cost any multiplication operation.

Based on the size of the matrices T_{1e}' $(k \times s)$, X_2 $(s \times r)$, and D_{1e} $(r \times k)$, we know that it requires $rk(s+k)$ operations to compute $P = T_{1e}' \cdot X_2 \cdot D_{1e}$. Therefore, step 3 to compute P has the complexity of $rk(s+k)$. The computation in step 5 costs $s.k^2 + r.k^2 = k^2(s+r)$. The complexity analysis for step 4 in Figure 4 is carried out in the same way as the process described in Section V-A, but on the matrix P.

Tables IV and V summarize the computational complexity of our incremental LSI method. Note that k is selected such that $k \ll t$ and $k \ll d$. Let us call $K = max\{k/t, k/d\}$ the *reduction percentage*. We can see that since P has small dimensions ($k \ll t \leq s$ and $k \ll d \leq r$), the computational complexity listed in Table V is *much less* than the one listed in Table III. The total *overhead* of incremental LSI is the complexity: $rk(s+k) + k^2(s+r)$ (Table IV). However, it is $O(n^3)$ for a matrix with the order n, which is one degree below the cost of Householder transformation ($O(n^4)$). The cost of that transformation dominates the total cost (Table III).

According to [6], setting K around 10% gives the better retrieval performance. Comparing Tables III and V at the Householder lines, if K=10% (i.e. $k \ll s$, $k \ll r$), our method saves a great deal of time in Householder transformation, thus, reduces time cost in iLSI. In brief, with small K, the computational cost of iLSI is much smaller than that of LSI.

VI. TLEM: A LINK MANAGEMENT TOOL

We implemented the iLSI method in our Traceability Link Evolution Management tool, TLEM. This section presents an overview of TLEM from the users' perspective. TLEM is integrated into Eclipse Platform as a plug-in. It uses Molhado [7], [24] (also an Eclipse plug-in) as a back-end SCM repository, to store the artifacts and traceability links at different versions.

	Correct-links retrieved	Incorrect-links retrieved	Missed Links	Total-links retrieved	Precision	Recall
Threshold	LSI, iLSI	LSI, iLSI	LSI, iLSI	LSI, iLSI	LSI, iLSI	LSI, iLSI
0.6	73,69	45,47	39,43	118,116	61.8%,59.4%	65.1%,61.6%
0.65	65,60	20,23	47,52	85,83	76.5%,72.3%	58.0%,53.6%
0.7	50,48	9,12	62,64	59,60	84.7%,80.0%	44.6%,42.8%

TABLE VII
COMPARISON BETWEEN LSI AND iLSI: PRECISION AND RECALL USING COSINE THRESHOLD

	LEDA 3.4	LEDA 3.4.1
Source files	497	517
Manual sections	115	117
Number of terms	1317	1360
LSI matrix dimensions	1317 x 612	1360 x 634

TABLE VI
SETTING FOR LEDA EXPERIMENT

With TLEM, a developer concerned about the traceability links between documentation and source code first triggers the execution of TLEM. TLEM connects to Molhado and checks if this is the first time of execution, that is, if no information on traceability links has been stored in Molhado from the previous execution of TLEM. If it is the first time, TLEM performs an TLR process using LSI technique (Section III). The outputs of TLR are traceability links between documentation and source code. Depending on the setting of parameters, the links can be fine-grained among fragments of artifacts or coarse-grained among files. TLEM maintains crucial LSI information about this TLR execution (Section IV) in order to quickly update traceability links when software artifacts change. At any time, the developer can inspect the set of current links and view the artifacts corresponding to a link via Eclipse editor.

The developer can then start modifying the documents. TLEM can be explicitly invoked to update traceability links by a user command or be invoked right before the changes are checked into the SCM repository. At this time, TLEM analyzes the changes to software artifacts and uses the incremental LSI algorithm for the automatic update of the links (Section IV). When the developer checks his/her changes into Molhado, not only the new version of software artifacts is saved, but also traceability links and crucial LSI information for later link updating are stored. Next time, any developer can check out altogether software artifacts and traceability links among them for any maintenance tasks. LSI information stored in the SCM repository allows TLEM to manage and update the traceability links during the subsequent editing sessions. Since the update time for links after the changes occur is very short, TLEM can be used in interactive development within Eclipse.

The choice of Molhado has manifolds. Firstly, it provides a generic versioned data model and can be easily tailored to support any type of data. Secondly, it is equipped with *versioned hypermedia support* [24] to store traceability links along with source code at any version. Thirdly, it has a rich set of API functions for *change management* that can be called from any Eclipse plugin [7]. Finally, it maintains a unique identifier for each source code or documentation entity, helping keep track of renamed or moved entities. Details on the storage representation for documents and links were described in [24].

VII. EVALUATION AND RESULTS

This section reports on our empirical evaluation on the performance/usefulness of our approach. In all experiments, we used WindowsXP, Intel Pentium 4 3Ghz, 1GB RAM.

A. Precision and Recall

The goal of this experiment is to compare the performance of iLSI method with that of traditional LSI in term of how precise and complete the iLSI method recovers the links.

The good performance of TLR is defined with two common information retrieval measures for TLR: *precision* and *recall* [21]. The **precision** value is defined as the number of correctly retrieved documents over the total number of retrieved documents. The **recall** value is defined as the number of correctly retrieved documents over the total number of relevant documents. In addition to the threshold approach (Section III), we used the *top rank* approach (also called *cut point*), which imposes a threshold for the number of recovered links, without considering the actual values of similarity measures.

We used all source code and manual pages of two versions 3.4 and 3.4.1 of LEDA (Library of Efficient Data types and Algorithms) (see Table VI). Two versions of LEDA are needed because we want to evaluate the iLSI method in an evolving software. We use each source code *file* and *manual section* as an individual document in the Term-Document matrix.

Firstly, we run TLEM using the traditional LSI in batch mode for the LEDA system at version 3.4. LSI's matrices for this version are computed during the process and later used in TLR for the system at version 3.4.1. For the version 3.4.1 of LEDA, we performed two executions: one with traditional LSI and one with incremental LSI. At both times, we measured the precision and recall values using both threshold and top rank approaches. Table VII summarizes the precision and recall results for LEDA 3.4.1 using *thresholds* for both methods LSI and iLSI. The first value in each pair corresponds to the traditional LSI method, and the second value to the iLSI method. We used the thresholds from 0.6-0.7, which have been suggested to provide better retrieval results for texts in previous experiments [1], [21]. K was chosen at 10%. We manually inspected the links and found 112 *correct* links which we used in computing precision and recall. Table VII shows that with those selected thresholds (0.6-0.7), iLSI has a slightly decrease in term of precision and recall. Precision values drop about 3-4%. Recall values drop about 4-5%.

Cut	Correct	Incorrect	Missed	Total	Precision	Recall
1	69	21	43	90	76.67%	61.60%
2	94	85	18	179	52.10%	83.92%
3	97	174	15	271	35.79%	86.61%
4	99	260	13	359	27.57%	88.39%
5	100	348	12	448	22.32%	89.29%
6	102	438	10	540	18.88%	91.07%
7	103	524	9	627	16.42%	91.96%
8	104	616	8	720	14.44%	92.86%
9	107	702	5	809	13.22%	95.54%
10	110	791	2	901	12.2%	98.21%
11	111	879	1	990	11.21%	99.1%
12	112	968	0	1080	10.37%	100%

TABLE VIII
PRECISION AND RECALL WITH TOP RANK FOR iLSI

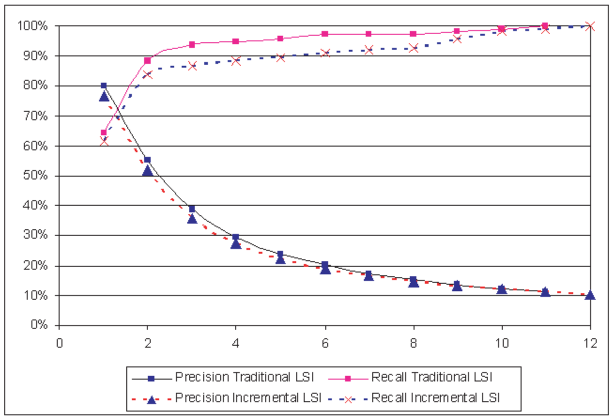


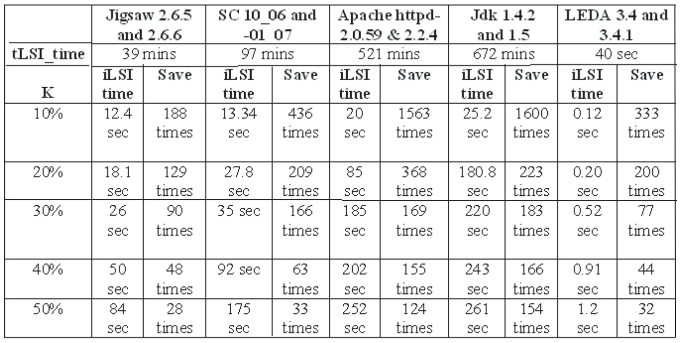
Fig. 5. Comparison LSI and iLSI with Top Rank

We repeated our experiment with the top rank approach (Table VIII). Table VIII is similar to Table VII except that the first column is cut points, instead of thresholds. Figure 5 shows the comparison between LSI and iLSI using the top rank approach. The results analyzed with the top rank approach agree with the results analyzed with the threshold approach. The precision and recall values are slightly decrease but not as much as the ones with thresholds. Precision values drop about 1-3%, while recall values drop about 4%. Thus, the results are a bit better with the top rank approach. In brief, the results showed that iLSI maintains almost as good performance as the traditional LSI method. We will investigate the impacts on precision and recall values in successive runs of iLSI.

B. Experiment on Time Cost

In this experiment, we evaluate time efficiency of iLSI in comparison with that of LSI. To observe the correlation between the reduction percentage K and the time iLSI could save, we run TLEM on several systems with different Ks.

For each software system, we have two versions: v_1 and v_2. Remind that the time cost for TLR on v_2 using incremental LSI method depends on k because the size of matrix P is $k \times k$ (Section IV). Thus, it depends on K. In contrary, the time to run traditional LSI method on v_2 of a software system does not depend on k or K because traditional LSI would require the re-run of SVD computation for the large Term-Document matrix. With traditional LSI, k is chosen during



	Jigsaw 2.6.5 and 2.6.6		SC 10_06 and -01_07		Apache httpd-2.0.59 & 2.2.4		Jdk 1.4.2 and 1.5		LEDA 3.4 and 3.4.1	
tLSI_time	39 mins		97 mins		521 mins		672 mins		40 sec	
K	iLSI time	Save	iLSI time	Save	iLSI time	Save	iLSI time	Save	iLSI time	Save
10%	12.4 sec	188 times	13.34 sec	436 times	20 sec	1563 times	25.2 sec	1600 times	0.12 sec	333 times
20%	18.1 sec	129 times	27.8 sec	209 times	85 sec	368 times	180.8 sec	223 times	0.20 sec	200 times
30%	26 sec	90 times	35 sec	166 times	185 sec	169 times	220 sec	183 times	0.52 sec	77 times
40%	50 sec	48 times	92 sec	63 times	202 sec	155 times	243 sec	166 times	0.91 sec	44 times
50%	84 sec	28 times	175 sec	33 times	252 sec	124 times	261 sec	154 times	1.2 sec	32 times

Fig. 6. Time Cost with iLSI, threshold=0.7

the reduction step (Figure 1), i.e., after the SVD computation. For each chosen K, we run TLEM on v_1 with the traditional LSI configuration. Then, we run it with traditional LSI on v_2. We recorded the time spent on v_2 to produce the links (`tLSI_time` in Figure 6). Then, for each K value, we run TLEM with the incremental LSI configuration on v_2 using the LSI result from the previous run on v_1. We recorded the time for each execution (`iLSI time`) and compared it with the time that TLEM run with the traditional LSI configuration on v_2. Figure 6 shows the results.

As seen in Figure 6 (see `Save` column), the incremental LSI method saves a large amount of time. With $K = 10\%$ and in large systems such as JDK or Apache, iLSI can reduce the time cost by 1500-1600 times. Importantly, link recovery time cost now is at the range of seconds to ten seconds, which is suitable for link evolution management during interactive development. In Figure 6, more time was saved for larger systems because the key point of iLSI is to reduce the SVD's computational cost on large LSI matrices. Furthermore, when K increases, time cost in using iLSI method increases as well because the LSI matrix is larger. However, with K from 10-40%, the TLR time is still small enough for interactive use. In brief, these results are consistent with our complexity analysis in Section V.

C. Tradeoff: Precision, Recall, and Time Cost

As seen in Figure 6, the time saving ratio depends on LSI's reduction percentage K of LSI matrices. However, precision and recall might be affected as K varies. This experiment aims to explore this correlation. We chose Jigsaw-2.6.5 and 2.6.6. We varied K, executed TLEM using incremental LSI, and recorded precision and recall values, and time cost.

In Figure 7, with a larger K, the incremental LSI process took longer time. However, it produced a better precision value and a lower recall value. This is because the approximate matrix is closer to the original one in term of least squares [6]. With $K = 100\%$, the tool had to run SVD for a matrix of the size 2240×2409. It took 39 mins 12s. With $K = 10\%$, i.e. with a matrix of 224×241, it took only 12.4s.

D. Experiment on Space Complexity

When a developer checks their software artifacts into the repository, in addition to changes to those artifacts, TLEM also stores *traceability links* among them as well as *LSI matrices*

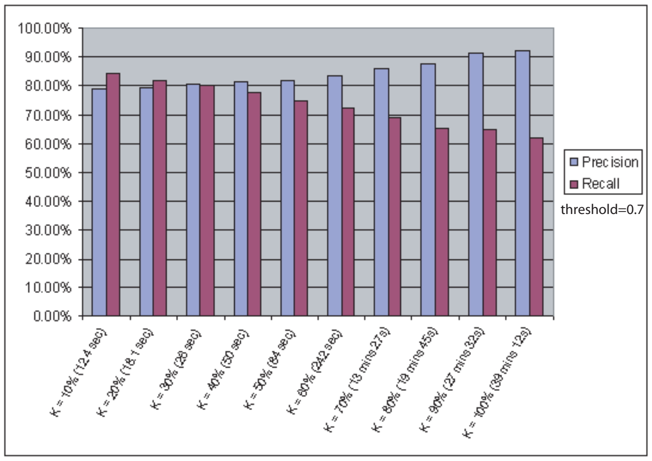


Fig. 7. Precision, Recall and Time Cost Tradeoff

Systems	Storage cost for links	Storage cost for matrices	Total source + documentation
LEDA 3.4.1	4.8 kB	65 kB	6.8 MB
Jigsaw-2.6.6	18.0 kB	204 kB	49 MB
SC_01_07	23.5 kB	220 kB	51 MB
Apache-2.2.4	42.1 kB	311 kB	72 MB

TABLE IX
SPACE COMPLEXITY

(i.e. T_{01}, S_{01}, and D_{01}) of the latest LSI computation for later use. Thus, we concerned with two sources of storage overhead in our approach: 1) the storage cost of LSI matrices, and 2) that of the traceability links themselves.

For each system in Table IX, we picked one version, filtered out irrelevant files, checked them into Molhado repository, and measured the total space required for that system. Then, we used TLEM to recover the traceability links in that system. We checked into the repository all artifacts, traceability links, and LSI matrices, and measured the overhead and total spaces.

Table IX shows the results. The sizes of corresponding LSI matrices were given in Table I. Table IX shows that the storage cost for traceability links themselves is very small, compared to the total cost for source code and documentation. More importantly, although the sizes of LSI's matrices are large, the storage cost for them is reasonably small. The main reason is that the LSI matrices are sparse, and S_{01} is even a diagonal matrix. T_{01} and D_{01} are the matrices of eigen vectors of the square symmetric matrices $X_1 \cdot X_1'$ and $X_1' \cdot X_1$, respectively. Therefore, storing eigen vectors and values, and X_1 is sufficient. Also, X_1 is a sparse matrix because each document contains only a small number of *distinctive* terms. Furthermore, if an SCM system has a good compression scheme, space saving ratio is even higher.

VIII. RELATED WORK

Information retrieval (IR) methods have been popularly used to address the problem of *automatic* recovery of traceability links among software artifacts. Antoniol *et al.* investigated two IR methods based on both vector space and probabilistic models [1], [2]. They used users' feedbacks as link training sets to improve the performance. Di Penta *et al.* [25] used this approach for TLR in systems developed with automatically generated code and COTS. Marcus *et al.* [21] showed the success of traditional LSI in TLR. In TLEM, iLSI method is leveraged to evolve traceability links for software evolution management. Poshyvanyk *et al.* combined LSI with scenario-based probabilistic ranking for feature identification [28].

De Lucia *et al.* [17] proposed an incremental traceability link recovery process using *users' feedbacks*, aiming at gradually identifying a threshold that achieves a good balance between retrieved correct links and false positives. Similar to our tool, COCONUT [18] is able to show the similarity level between high-level artifacts and source code during development. However, COCONUT runs LSI in the batch mode on a snapshot of the system in development. ADAMS Re-Trace is a LSI-based TLR tool for different types of artifacts in ADAMS, an artifact management system [16]. Our preliminary iLSI [14] was limited and could not handle general changes. LSI was successfully used by Lormans *et al.* [15] for TLR among design artifacts and test case specifications.

Using TLR in requirement engineering, Hayes *et al.* [12] developed RETRO (REquirements TRacing On-target) that used both vector space model and LSI method. It allows users' feedbacks that change the weights in a Term-Document matrix. Using RETRO on industrial projects, they examined the compromising level between recall and precision [11].

In addition to IR techniques, other approaches have been used for *(semi-)automatic* recovery of traceability links. Egyed and Grunbacher [9], [10] introduced a *rule-based*, scenario-driven approach in which links are generated using observed scenarios of a system. Cleland-Huang *et al.* [4] adopted a change event-based architecture to link requirements and artifacts through a publish-subscribe relationship. In Goal Centric Traceability [5], a probabilistic model is used to dynamically retrieve links between classes affected by a change and elements within a softgoal interdependency graph. PRO-ART [27] uses a process-driven approach in which links are generated as a result of modifying a product via development tools.

In [33], rules are used to identify syntactically related terms in requirements and use cases with semantically related terms in an object model. In TOOR [26], links are defined and derived in terms of axioms. ArchTrace uses a policy-based infrastructure for automatically updating traceability links every time an architecture or source code evolves [22]. APIS traceability environment [23] integrates different engineering tools using a warehouse strategy with search capabilities. Baniassad *et al.* [3] introduced the design pattern rationale graph to makes explicit the relationships between design concepts to code. Maletic *et al.* [20] gave a formal definition of model-to-model links and discuss traceability issues involved with evolution. Sharif *et al.* [31] proposed to evolve traceability links by examining fine-grained differences of models that lie on the links' endpoints.

In addition to TLR tools, there have been many approaches for managing traceability links. Spanoudakis and Zisman [34] provide an excellent survey on software traceability. According to them, approaches to traceability link management include 1) a single centralized database [8], [26], [29], 2) software repository [27], 3) hypermedia [19], [32] 4) mark-up documents [33], 5) event-based [4], and 6) hypertext versioning [19]. In general, in those approaches, there are two major problems: (1) limited automatic tools for link creation, and (2) tools cannot *automatically* manage the links in *evolving* software.

IX. Conclusions

Traceability of software artifacts has been considered as an important factor for supporting various software development activities. Despite successes, existing approaches for *automatic* traceability link recovery (TLR) cannot be used in *evolving* software during development due to their high computational cost. Existing tools for link management rely on developers to create links and cannot effectively and automatically update the links when changes occur.

We developed a technique to automatically manage traceability link evolution and update the links in evolving software. Our novel technique, Incremental LSI (iLSI), effectively computes and updates the set of traceability links by analyzing changes to software artifacts and by re-using the result from the previous LSI computation. Our key contribution is the leverage of LSI, an advanced automatic TLR technique, to provide the automatic supports for traceability link evolution management. Based on iLSI, we have developed TLEM, an automatic traceability link evolution management tool, which operates in connection with Eclipse and an SCM repository. When artifacts change, TLEM uses iLSI technique to quickly update the links in tens of seconds, thus, well-suited for interactive use during software development. The results of our experimental studies showed that while iLSI greatly reduces the time cost, it still maintains high precision and recall values for TLR, and requires little extra space. Importantly, other LSI-based approaches can apply the mathematic foundation in iLSI to address software evolution issues in traceability.

Acknowledgment

We would like to thank Giuliano Antoniol and Andrian Marcus for their kindness to provide us the LEDA data.

References

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

[2] G. Antoniol, G. Canfora, A. de Lucia, and G. Casazza, "Information retrieval models for recovering traceability links between code and documentation," in *ICSM'00*. IEEE CS, 2000, p. 40.

[3] E. L. Baniassad, G. C. Murphy, and C. Schwanninger, "Design pattern rationale graphs: Linking design to source," in *ICSE'03*. IEEE CS, 2003, pp. 352–361.

[4] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event-based traceability for managing evolutionary change," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 796–810, 2003.

[5] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina, "Goal-centric traceability for managing non-functional requirements," in *ICSE '05*. ACM Press, 2005, pp. 362–371.

[6] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society for Info. Sci.*, vol. 41, no. 6, pp. 391–407, Sept 1990.

[7] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen, "Refactoring-aware Configuration Management for Object-Oriented Programs," in *ICSE'07: the 29th International Conference on Software Engineering*. IEEE CS, 2007, pp. 427-436.

[8] "Teleologic DOORS," http://www.teleologic.com/.

[9] A. Egyed, "A scenario-driven approach to trace dependency analysis," *IEEE Transactions on Software Engineering*, vol. 9, no. 2, 2003.

[10] A. Egyed and P. Grunbacher, "Automating requirements traceability: Beyond the record and replay paradigm," in *ASE'02: Int. Conference on Automated Software Engineering*. IEEE CS, 2002, pp. 163–171.

[11] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Improving After-the-Fact Tracing and Mapping: Supporting Software Quality Predictions," *IEEE Software*, vol. 22, no. 6, pp. 30–37, 2005.

[12] ——, "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods," *IEEE TSE*, vol. 32, no. 1, pp. 4–19, 2006.

[13] R. A. Horn and C. R. Johnson, *Matrix Analysis*. Cambridge University Press, 1985.

[14] H. Jiang, T. N. Nguyen, C. Chang, and F. Dong, "Traceability Link Evolution Management with Incremental Latent Semantic Indexing," in *COMPSAC'07*. IEEE CS, 2007, pp. 309–316.

[15] M. Lormans and A. V. Deursen, "Can LSI help Reconstructing Requirements Traceability in Design and Test?" in *CSMR'06: European Conf. on Software Maintenance & Reengineering*. IEEE CS, 2006, pp. 47–56.

[16] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "ADAMS Re-Trace: A Traceability Recovery Tool," in *CSMR'05: European Conf. on Software Maintenance & Reengineering*. IEEE CS, 2005, pp. 32–41.

[17] A. D. Lucia, R. Oliveto, and P. Sgueglia, "Incremental approach and user feedbacks: a silver bullet for traceability recovery," in *ICSM'06: Int. Conference on Software Maintenance*. IEEE, 2006, pp. 299–309.

[18] A. D. Lucia, M. D. Penta, R. Oliveto, and F. Zurolo, "COCONUT: COde COmprehension Nurturant Using Traceability," in *ICSM'06: Int. Conference on Software Maintenance*. IEEE CS, 2006, pp. 274–275.

[19] J. Maletic, E. Munson, A. Marcus, and T. N. Nguyen, "Using a hypertext model for traceability link conformance analysis," *TEFSE'03: Workshop on Traceability in Emerging Forms of Softw. Eng.*. ACM Press, 2003.

[20] J. Maletic, M. Collard, and B. Simoes, "An XML-Based Approach to Support the Evolution of Model-to-Model Traceability Links," in *TEFSE'05*. ACM Press, 2005, pp. 67–72.

[21] A. Marcus and J. I. Maletic, "Recovering documentation to source code traceability links using latent semantic indexing," in *ICSE'03*. IEEE CS, 2003, pp. 125–135.

[22] L. Murta, A. van der Hoek, and C. Werner, "ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links," in *ASE'06*. IEEE CS, 2006, pp. 135–144.

[23] C. Neumuller and P. Grunbacher, "Automating software traceability in very small companies: A case study and lessons learne," in *ASE'06*. IEEE CS, 2006, pp. 145–156.

[24] T. N. Nguyen, E. Munson, J. Boyland, and C. Thao, "An Infrastructure for Development of Multi-level, Object-Oriented Configuration Management Services," in *ICSE'05: the 27th International Conference on Software Engineering*. ACM Press, 2005, pp. 215–224.

[25] M. D. Penta, S. Gradara, and G. Antoniol, "Traceability Recovery in RAD Software Systems," in *IWPC'02*. IEEE CS, 2002, pp. 207–216.

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

[27] K. Pohl, "PRO-ART: Enabling requirements pre-traceability," *ICRE'96: Int. Conference on Requirements Engineering*, 1996, pp. 76–85.

[28] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *ICPC'06*. IEEE CS, 2006, pp. 137–148.

[29] "Integrated chipware, RTM," www.chipware.com.

[30] G. Salton and C. Yang, "On the specification of term values in automatic indexing," *Journal of Documentation*, vol. 29, no. 4, pp. 351–372, 1973.

[31] B. Sharif and J. Maletic, "Using fine-grained differencing to evolve traceability links," in *GCT'07: Intl. Symp. on Grand Challenges in Traceability*. ACM Press, 2007, pp. 76–81.

[32] S. Sherba, K. Anderson, and M. Faisal, "A framework for mapping traceability relationships," in *TEFSE'03*. ACM Press, 2003.

[33] G. Spanoudakis, A. Zisman, E. Perez-Minana, and P. Krause, "Rule-Based Generation of Requirements Traceability Relations," *Journal of Systems and Software*, pp. 105–127, 2004.

[34] G. Spanoudakis and A. Zisman, "Software Traceability: A Roadmap," *Handbook of Software Engineering and Knowledge Engineering*, vol. 3, World Scientific, 2005.