Using a Hypertext Model for Traceability Link Conformance Analysis

Jonathan I. Maletic[1], Ethan V. Munson[2], Andrian Marcus[3], Tien N. Nguyen[2]

[1]*Department of Computer Science*
Kent State University
Kent Ohio 44242
330 672 9039

[2]*Department of EECS*
University of Wisconsin-Milwaukee
Milwaukee, WI 53201
414 229 4438

[3]*Department of Computer Science*
Wayne State University
Detroit, MI 48202
313 577 5408

jmaletic@cs.kent.edu, munson@cs.uwm.edu, amarcus@cs.wayne.edu, tien@cs.uwm.edu

Abstract

A number of techniques for semi-automated traceability link recovery between source code and documentation have recently been proposed to support the reverse engineering and maintenance of legacy systems. This is only the first step in supporting the long term maintainability of such systems. A crucial issue, after recovering traceability links is analyzing their general conformance over time. That is, as a system is changed during evolution the validity and conformance of the links may change. Thus, conformance analysis must be performed to identify possible non-conformance of the links. The paper presents a holistic view of how to combine link recovery with conformance analysis that is facilitated by a formal hypertext model. This hypertext model not only supports complex linking structures (e.g., multi links) but also supports versioning of individual links. Such a model preserves and maintains over time the results of the reverse engineered traceability links.

1 Introduction

Extensive effort in the software engineering community (both research and commercial) has been brought forth to improve the explicit connection of documentation and source code. A number of integrated development environments and CASE tools are particularly focused on this issue. These tools and techniques have made great strides in documentation to source code traceability for the development of new software systems. Unfortunately, recovery of these types of links from legacy systems and maintaining the validity of these links as a system is evolved is still quite problematic.

The need for tools and techniques to recover documentation to source code traceability links in legacy systems is particularly important for a variety of software engineering tasks. These include general maintenance tasks, impact analysis, program comprehension, and more encompassing tasks such as reverse engineering for redevelopment and systematic reuse.

The major obstacle facing construction of useful tools for this type of link recovery and link maintenance is that the links are rarely explicit and (not without exception) based on the semantic meaning of the prose in the documentation. Relating some sort of natural language analysis of the documentation with that of the source code is an obviously difficult problem.

Our solution to this problem is to utilize an advanced information retrieval technique (i.e., latent semantic analysis) to extract the meaning (semantics) of the documentation and source code [17, 19]. We then use this information to identify traceability links based on similarity measures. The method utilizes all the comments and identifier names within the source code to produce semantic meaning with respect to the entire input document space. This is supported well by the work of Anquetil [1] and others in determining the importance of this information in existing software. This implies the assumption that the comments and identifiers are reasonably named however, the alternative bares little hope of deriving a meaning automatically (or even manually).

We represent these links (relationships) via a formal hypertext model that is easily supported by using underlying XML representations such as seen in [6, 21]. This hypertext model supports and maintains a fine-grained versioning of the links to facilitate process of *conformance analysis*, which identifies possible sources of inconsistency among software documents. By providing a fine grained hypertext revision control framework we can address the versioning of both documents and relationships simultaneously. This is necessary because otherwise linking would be limited to entire documents (and possibly single documents).

The general problem of traceability is discussed followed by our method for link recovery is given. Then the hypertext model is described. The goal of this paper is to describe how these two issues can be bridged together to support both reverse engineering and evolution of software systems.

2 Traceability and Inconsistency

Conformance analysis is related to the well-known processes of requirements traceability [12, 34] and inconsistency management [13, 24, 30]. A large amount of research has been conducted on both of these areas. Our general approach assumes that the source and external documentation are available but that no explicit relationships exist between the two.

Requirements traceability and its importance in software development process have been well described [12, 34]. A number of requirements tracing tools have been developed and integrated into software development environments [2-4, 20, 25, 26, 28]. Other research seeks to develop a reference model for requirements traceability that defines types of requirement documentation entities and traceability relationships [15, 27, 32]. Dick [9] extends the traceability relationships to support more consistency analyses. Inconsistency management and impact analysis have been studied since the late 1980s [30]. According to Spanoukadis and Zisman [30], inconsistency management can be viewed as a process composed of six activities: detection of overlaps between software artifacts, detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, history tracking of inconsistency management process, and specification of an inconsistency management policy. The activities to be taken depend on the type of inconsistency being addressed [24].

3 Latent Semantic Indexing

We utilize an information retrieval method, latent semantic indexing, to drive the link recovery process. A distinct advantage of using our link recovery method is that it does not rely on a predefined vocabulary or grammar for the documentation or source code. This allows the method to be applied without large amounts of preprocessing or manipulation of the input, which drastically reduces the costs of link recovery [20].

Latent Semantic Indexing (LSI) [8, 10] is a vector space model (VSM) based method for inducing and representing aspects of the meanings of words and passages reflective in their usage. Work applying LSI to natural language text [5, 16] has shown that that LSI not only captures significant portions of the meaning of individual words but also of whole passages such as sentences, paragraphs, and short essays. The central concept of LSI is that the information about word contexts in which a particular word appears, or does not appear, provides a set of mutual constraints that determines the similarity of meaning of sets of words to each other.

In its typical use for text analysis, LSI uses a user-constructed corpus to create a term-by-document matrix. Then it applies Singular Value Decomposition (SVD) [29] to the term-by-document matrix to construct a subspace, called an LSI subspace. New document vectors (and query vectors) are obtained by orthogonally projecting the corresponding vectors in a VSM space (spanned by terms) onto the LSI subspace.

According to the mathematical formulation of LSI, the term combinations which are less frequently occurring in the given document collection tend to be precluded from the LSI subspace. This fact, together with our examples above, suggests that one could argue that LSI does “noise reduction” if it was true that less frequently co-occurring terms are less mutually-related, and therefore less sensible.

The formalism behind SVD is rather complex and lengthy to be presented here. The interested reader is referred to [29] for details. Intuitively, in SVD a rectangular matrix $\mathbf{X}$ is decomposed into the product of three other matrices. One component matrix ($\mathbf{U}$) describes the original row entities as vectors of derived orthogonal factor values, another ($\mathbf{V}$) describes the original column entities in the same way, and the third is a diagonal matrix (Σ) containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed (i.e., $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^{\mathbf{T}}$). The columns of $\mathbf{U}$ and $\mathbf{V}$ are the left and right singular vectors, respectively, corresponding to the monotonically decreasing (in value) diagonal elements of Σ which are called the singular values of the matrix $\mathbf{X}$. The first $\mathbf{k}$ columns of the $\mathbf{U}$ and $\mathbf{V}$ matrices and the first (largest) $\mathbf{k}$ singular values of $\mathbf{X}$ are used to construct a rank-$\mathbf{k}$ approximation to $\mathbf{X}$ through $\mathbf{X}_\mathbf{k} = \mathbf{U}_\mathbf{k}\Sigma_\mathbf{k}\mathbf{V}_\mathbf{k}^{\mathbf{T}}$. The columns of $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, such that $\mathbf{U}^\mathbf{T}\mathbf{U} = \mathbf{V}^\mathbf{T}\mathbf{V} = \mathbf{I}_\mathbf{r}$, where $\mathbf{r}$ is the rank of the matrix $\mathbf{X}$. $\mathbf{X}_\mathbf{k}$ constructed from the $\mathbf{k}$-largest singular triplets of $\mathbf{X}$ (a singular value and its corresponding left and right singular vectors are referred to as a *singular triplet*), is the closest rank-$\mathbf{k}$ approximation (in the least squares sense) to $\mathbf{X}$.

Once the documents are represented in the LSI subspace, the user can compute similarities measures between documents by the cosine between their corresponding vectors or by their length. These measures can be used for clustering similar documents together, to identify “concepts” and “topics” in the corpus. This type of usage is typical for text analysis tasks. The LSI representation can also be used to map new documents (or queries) into the LSI subspace and find which of the existing documents are similar (relevant) to the query. This usage is typical for information retrieval tasks.

4 The Traceability Recovery Process

Our traceability recovery process centers on LSI and is partially automated. However, user input is necessary and the degree of user involvement depends on the type of source code and the user’s task. As mentioned, recovering the links between source code and documentation may support various software engineering tasks. Different tasks (and users) typically require different types of information. For example, there are times completeness is important. That is, the user needs to recover ALL the correct links even if that means recovering many incorrect ones at the same time. Other times, precision is preferred and the user restricts the search space so all the recovered links will be correct ones, even if this means not finding them all. Our system tries to accommodate both needs (separately of course). One way to accommodate the user needs is by offering multiple ways to recovering the traceability links.

Figure 1 depicts the major elements in the traceability recovery process. This process in based on our previous work presented in [20]. The user's involvement in the

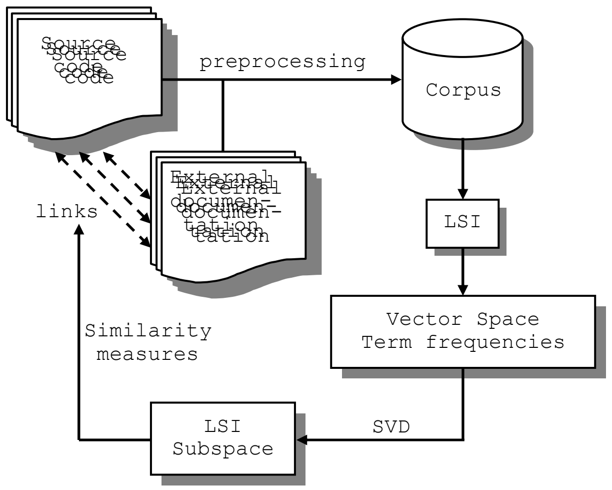


Figure1. The traceability recovery process

process occurs in the beginning for selecting the source code and documentation files. Then the user selects the dimensionality of the LSI subspace. After the LSI subspace is generated, the user determines what type of threshold will be used in determining the traceability links.

4.1 Building the Corpus

The input data consists of the source code and external documentation. In order to construct a corpus that suits LSI, a simple preprocessing of the input texts is required. Both the source and the documentation need to be broken up into the proper granularity to define the documents, which will then be represented as vectors.

In general, when applying LSI to natural text, a paragraph or section is used as the granularity of a document. Sentences tend to be to small and chapters too large. In source code, the analogous concepts are function, structure, module, file, class, etc. Obviously, statement granularity is too small. More than that, the choice of the granularity level is influenced by the particular software engineering task. In previous experiments involving LSI and source code, we used functions as documents in procedural source code [17, 19] and class declarations in OO source code [18]. The goal there was to cluster elements of the source code based on semantic similarity, rather than mapping them to documentation.

In this application, a part of the documentation may refer to different structures in the source code (i.e., a class, a hierarchy of classes, a set of functions or methods, a data structure, etc.). Therefore, in order to allow for flexibility and simplicity, which in turn better support automation, we define each file as a document. Obviously, some files will be too large. In those situations, the files are broken up into parts roughly the size of the average document in the corpus. This ensures that most of the documents have a close number of words and thus may map to vectors of similar lengths. Of course, in some cases this break up of the files could be rather unfortunate, causing some documents from the source code to appear related to the wrong manual sections. It is a trade-off we are willing to take in favor of simplicity and low-cost of the preprocessing. If this situation is unacceptable for the user, they have the option of (re)combining a number of documents into a new one and identify which existing documents are most similar.

As far as documentation is concerned, the chosen granularity is determined by the division in sections of the documents, defined by the original authors (usually summarized in the table of content). Important to note is that in this process, no grammar based parsing of the source code is necessary. LSI does not use a predefined vocabulary, or a predefined grammar, therefore no morphological analysis or transformations are required.

One can argue that the mnemonics and words used in constructing the identifier may not occur in the documentation. That is certainly true. It is in fact the reason why we chose to also use the internal documentation (i.e., comments) in constructing the corpus. Of course, our assumption is that developers define and use the identifiers with some rationale in mind and not completely randomly.

4.2 Defining the Semantic Similarity Measure

Before we give a detailed explanation of this process, some mathematical background and definitions are necessary.

Definition. In this process a *source code document* (or simply document) *d* is any contiguous set of lines of source code and/or text. Typically a document is a file of source code or a program entity such as a class, function, interface, etc.

Definition. An *external document* (*e*) is any contiguous set of lines of text from external documentation (i.e., manual, design documentation, requirement documents, test suites, etc.). Typically an external document is a section, a chapter, or maybe an entire file of text.

Definition. The external documentation is also a set of documents $M = \{e_1, e_2, \ldots, e_m\}$. The total number of documents in the documentation is $m = |M|$.

Definition. A *software system* is a set of documents (source code and external) $S = D \cup M = \{d_1, d_2, \ldots, d_n\} \cup \{e_1, e_2, \ldots, e_m\}$. The total number of documents in the system is $n+ m = |S|$.

Definition. A *file* f_i, is then composed of a number of documents and the union of all files is S. Size of a file, f_i, is the number of documents in the file, noted $|f_i|$.

LSI uses the set $S = \{d_1, d_2, \ldots, d_n, e_1, e_2, \ldots, e_m\}$ as input and determines the *vocabulary* V of the corpus. The number of words (or terms) in the vocabulary is $v = |V|$. Based on the frequency of the occurrence of the terms in the documents and in the entire collection, each term is weighted with a combination of a local log weight and a global entropy weight. A term-document matrix $\mathbf{X} \in \boldsymbol{R}^{v\times n}$ is constructed. Based on the user-selected dimensionality (k), SVD creates the LSI subspace. The term-document matrix is then projected onto the k-dimensional LSI subspace. Each document $d_i \in D \cup M$, will correspond to a vector $\mathbf{x_i} \in \mathbf{X}$ projected onto the LSI subspace.

Definition. For two documents d_i and d_j, the *semantic similarity* between them is measured by the cosine between their corresponding vectors $\text{sim}(d_i, d_j) = \cos(\mathbf{x_i}, \mathbf{y_i})$ The value of the measure will be between [-1, 1] with value (almost) 1 representing that the two are (almost) identical.

One important aspect to consider is the granularity of the documents. The external documentation is usually composed of paragraphs, sections, or chapters. These are then natural choices in determining the definition of an external document in particular cases. The organization of the source code differs from one programming language to another. The simplest way to determine the documents granularity is using the file decomposition. Obviously this will not suffice for all tasks. Therefore choosing classes, functions, or interfaces as source code documents is often time more desirable. Since one of the goals is for the framework to be as flexible as possible, using a full parser for each possible language is impractical. Instead, we developed a simple lexical parser that can be used to divide C, C++, and Java source code into documents of different granularity levels (i.e., functions, methods, interfaces, and classes).

5 Link Representation Model

Software documents (i.e., source code and external documents) along with relationships between documents can be modeled as a network (web) of nodes and links where each node represents a document (or some part of a larger document) and each link represents a relationship between these documents. While it is common to use graph-based representations for various software engineering processes such as software configuration management [11], it is not common to use a hypertext model. This section discusses the nature of document relationships and describes how we use the hypertext model to represents those relationships to support conformance analysis.

5.1 Software Document Relationships

Relationships among software documents can be broadly divided two categories 1) conformance and 2) non-conformance (those that do not need to conform). Conformance relationships represent logical semantic dependencies between documents, such as a particular requirement “motivating” a specific design feature. Non-conformance relationships support concepts such as navigational and organizational tasks (e.g., “next document” links on individual pages). Non-conformance relationships are not directly relevant to conformance analysis and their maintenance can often be automated using standard browser management technology.

Conformance relationships can be further divided into the causal and non-causal. Causal conformance relationships (causal relationships for short) represent those relationships that carry with them an implied logical ordering of the documents involved. For example, testing and bug reports cannot be produced until an implementation is available, and while it is not necessarily the case that requirement documents will be written before designs, there is certainly a logical relationship between them that makes the design depend on requirements. In conformance analysis, the most important characteristic of causal relationships is that causation establishes a partial ordering in time among entities [31]. Therefore, when this partial order in time is violated, it is possible that the conformance between documents has been broken. A causal relationship can be considered as a relation between entities, that is something happens and causes something else to happen. Mathematically, the causal relationship is transitive, irreflexive, and anti-symmetric.

Non-causal conformance relationships (non-causal relationships for short) exist when documents or parts of them must agree with each other, but the causality cannot be clearly identified. For example, multiple versions of the same document in different languages must agree, but there need not be a causal relationship among them. Since changes to any of the entities in a non-causal relationship may cause the logical semantics of the relationship to become invalid, conformance analysis must be able to account for such relationships. Mathematically, the non-causal relationship is transitive, reflexive, and symmetric.

Regardless of type, document relationships also have variable arity and can have fine granularity.

Variable arity: While simple binary relationships are common in software documents, it is also common for relationships to connect many entities. An entity can have multiple causes with none of them alone sufficient to cause it. It can in turn produce multiple effects. So, causal relationships may have multiple sources and multiple targets. For instance, several items listed in a particular requirement specification (non-functional) could "affect" the design of an algorithm. The same pattern holds true for our non-causal relationship example, where documents in several languages may need to agree.

Fine granularity: Document relationships can exist between documents and within a single document. In software documents, which can be rather lengthy, it will be common that relationships will connect relatively small sections of material, such as functions, classes, paragraphs, or subsections. So, document relationships must be fine-grained.

In summary, software document relationships can be divided into three broad classes (non-conformance, causal conformance, and non-causal conformance), have variable arity, and connect fine grained entities within software documents.

5.2 The Hypertext Model

Because conformance analysis is performed in the domain of linked documents, it is natural to use a hypertext model [7] as a basis for representing these relationships.

Though different hypertext systems have variations of the notion, the hypertext model can be defined as a set of intellectual works and their inter- and intra-work relationships, represented by links, in combination with a user interface for viewing instances of these works and navigating from instance to instance across links [35]. A work is an artifact that can be drawn from any medium, such as text, image, or video. A wide variety of terms have been used to describe a work in hypertext systems including document, card, frame, node, object, and component. In the hypertext model, a link (or hyperlink) is a first-class entity and defined as an association among a set of works or anchors. Although the notion of anchor varies from system to system, anchors always denote regions of interest within a work and form the endpoints for links.

In the hypertext model, links have two important properties: arity and directionality. The arity of a link specifies the number of its endpoints. Many hypertext representations, such as HTML, only support binary (two endpoint) links. More sophisticated hypertext representations (e.g., XLink) permit n-arity links, which can have a variable numbers of endpoints and have also been called multi-links or multi-headed links.

Link directionality takes two forms: navigational and logical. Navigational directionality refers to the direction(s) in which a link may be traversed. For example, HTML links are navigationally unidirectional, only supporting traversal from the source of the link to its destination (history mechanisms in browsers are used to provide a form of reverse navigation). If a link can also be traversed backwards, from destination to source, it is navigationally bi-directional.

Logical directionality is a semantic quality that is independent of how a link can be traversed. For example, the logical direction in a link that represents the relationship "A causes" is from the source of the cause, A, to the destination B. Logical directionality has not been widely discussed in the hypertext literature. Our research depends heavily on separating logical and navigational directionality.

5.3 Hypertext and Document Relationships

We are using the hypertext model as the basis for representing software documents and their relationships. Each software document will be a work in the sense used by the hypertext model. Software documents will have internal structure, such as that provided by XML, and it will be possible to define an anchor corresponding to any well-defined structural unit. Anchors may also be defined over contiguous selections using an approach such as the Document Object Model Range [33].

Links will be divided into three classes corresponding to the three classes of document relationships identified earlier: non-conformance, causal, and non-causal. All links are fundamentally n-ary, though in practice binary links are very common. All links have named types (such as "requires," "must agree," or "next item"), but the set of types is not fixed, so that a variety of software processes can be supported.

Non-conformance links are intended to support navigational and organizational relationships that have little or no relevance to determine agreement between documents, such as table of contents and index links. Their navigational directionality is user-specifiable. They do not participate in the conformance analysis process. Each non-conformance link has an identifier and an extensible set of attributes.

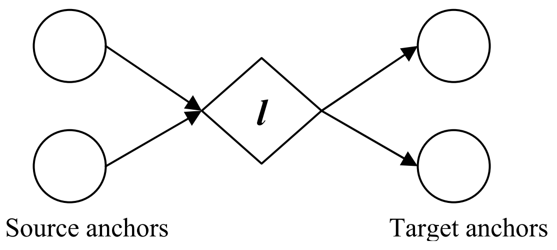
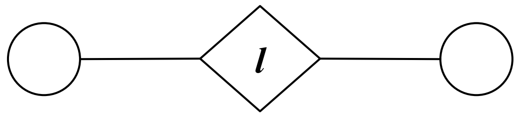


Figure 2. Causal relationship representation

Each causal link represents a causal relationship, $\boldsymbol{R}$. Causal links are always directional, connecting a set of source anchors and a set of target anchors. The source and target sets of a link $\boldsymbol{l}$ are denoted by $\boldsymbol{I_l}$ and $\boldsymbol{O_l}$, respectively. If a link's source and target sets each have one element, the link is binary. The directionality of the link is used to represent the causal semantic dependency, but the navigational directionality is user-specifiable. Figure 2 shows a graphical representation of a causal link. In Figure 2, circles represent anchors and the diamond represents a link. Directed edges connect the source anchors to the link and connect the link to its target anchors.

Each non-causal link represents a non-causal relationship, designated by the symbol ***r***. Non-causal links participate in conformance analysis and are logically non-directional. Their navigational directionality can be specified by the user. Let us use $\boldsymbol{r_l}$ to denote the set of nodes connected by the non-causal link ***l***. Figure 3 illustrates this representation, where an undirected edge connects each anchor to a non-causal link.

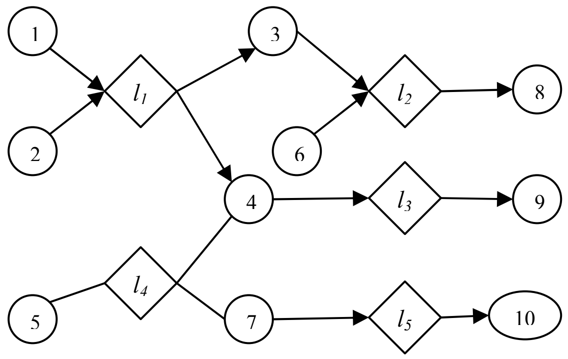


Anchor i

Figure 3. Non-causal relationship representation

5.4 Conformance Graph

The conformance graph is a five-tuple $G = \{S, L_C, L_N, E_D, E_N\}$ where $S = D \cup M$ is the set of documents (nodes) , L_C is a finite set of causal link nodes, L_N is a finite set of non-causal (conformance) link nodes, E_D is a finite set of directed edges and E_N is a finite set of non-directed edges.



Each node in L_C represents a causal link. Each node in L_N represents a non-causal link. Each directed edge in E_D either connects a source anchor to a causal link or a causal link to a target anchor (Figure 2). Each non-directed edge in E_N connects an anchor to a non-causal link (Figure 3). It is important each relationship is represented by a node, rather than by an edge, as is often the case in other graph representations of dependencies.

Figure 4 shows an example of a conformance graph. There are ten document nodes in S. The links l_1, l_2, l_3, l_5 are causal links, which are all in L_C. L_N contains only l_4. Causal links have incoming and outgoing edges that are directed, while l_4 and its anchors are connected by non-directed edges. The conformance graph must be acyclic with respect to conformance links, because cycles would imply that a node causes itself. It must also satisfy the unique producer condition that Gunter defined for the p-net [14]. Conformance links may not be degenerate: 1) Non-causal links must connect at least two anchors, and 2) Causal links must have non-empty source and target sets. There are additional restrictions on the conformance graph when structural relationships among nodes are present [22].

5.5 A Timestamp Strategy

Our research has developed a method [23] to partially automate the detection of conformance problems using timestamps on links and anchors in a manner similar to that used by the make [11] build optimization tool.

We assume that both anchors and links are first-class objects that are under version control. New versions of anchors are created each time the material within an anchor is altered in some way. Link versions change when the set of anchors connected by a link is altered. Both anchors and links have unique IDs and version information including version ID, version creator, and a modification timestamp. In addition, links must have a validation timestamp that records the time when the link was last validated by human inspection. When a link gets updated to a new version, the validation timestamp is copied from the previous version, even though validation predates the creation of the version.

5.6 Conformance Ratings

Our conformance analysis method compares anchor modification timestamps and link validation timestamps to produce a heuristic conformance rating of the likely seriousness of conformance problems. Conformance ratings are made on a scale from zero (likely conformance) to ten (likely non-conformance) and have yet to be validated empirically. The conformance rating for a link is determined in one of two ways, depending on whether the link is causal or non-causal.

For causal links, four timestamps are relevant: t_{srcmax}, the maximum version timestamp of the source anchors; $t_{destmin}$, the minimum version timestamp of the destination anchors; $t_{destmax}$, the maximum version timestamp of the destination anchors; and t_{valid}, the validation timestamp of the link. A case analysis of the order of these timestamps has been made, defining conformance ratings for each case, as shown in Table 1. For example, if $t_{srcmax} \le t_{destmin}$ and $t_{destmax} \le t_{valid}$ then the destinations are newer than the sources and the link relationship was validated by a human sometime after the last changes to both. So, the rating is zero. In contrast, if $t_{srcmax} \leq t_{valid} \leq t_{destmax}$ then one of the destinations has changed since the last validation and the rating are five. We have arrived at these ratings only by human judgment. A correct and useful formula for these ratings remains an open question and will probably require empirical research.

When causality is not available to provide clues, we turn to a more formulaic approach. Assume that a non-causal link connects N anchors and let t_i be the timestamp of the i^{th} anchor. Then, if there exists any t_i such that $t_i > t_{valid}$ (where is the number of such anchors) then the conformance rating $CR = \alpha + (n/N) \times (10 - \alpha)$. Otherwise, $CR = 0$. α is an adjustable parameter representing the minimum rating when the timestamps suggest that a problem exists.

This is supported by a formal conformance analysis model with a number of desirable properties that support automated analysis.

6 Bridging the Gap

In [20] we showed how our link recovery method can be successfully applied to existing software systems and associated documentation. Each of these recovered links is then represented within the hypertext model presented.

There are a number of interesting problems at this point. The first is how to determine the type of the recovered links (non-conformance, causal conformance and non-causal conformance relationships) including the arity of each link and its direction. Then we need to determine the initial assignment of conformance ratings to each link.

The arity of the links is based on the semantic similarity measure. We pick a threshold for the measure on each document and this can result in multiple links tied to each document. That is, one document can be semantically similar to a more than one other document.

We can assume that the links recovered are not navigational in nature and as such we only need to determine if they are causal or non-causal. The directionality of the links can be partially determined by the type of the documents being linked. Requirements documents should (in general) cause design documents and in turn cause implementation. This combined with other types of heuristics can deal with a large number of situations. User intervention will be required in the remaining cases.

The initial assignment of conformance ratings to links remains an open problem. Since links are only recovered when there is evidence of a meaningful semantic relationship between documents, at least a modest level of conformance is likely. Statistics drawn from LSI are likely to provide evidence to the level of conformance, but this approach will require empirical validation. In some cases, human inspection will probably be required to validate recovered links.

Other issues arise from the problem of maintaining links in an evolving system. It is unlikely that developers of a large system will have either the time or the motivation to inspect every link that might be affected by a set of document changes. The timestamp approach to conformance analysis, as described earlier, is designed to address just this problem, but its conformance ratings are based solely on the existence of changes, not on the underlying semantics of the changes.

LSI-derived statistics can be used to construct *semantic signatures* that correct this problem. The semantic signature for document i is a vector $\mathbf{s}$ of k real values, where $\mathbf{s} = \mathbf{U_k v_i}$, $\mathbf{U_k}$ is the first k columns of $\mathbf{U}$, and $\mathbf{v_i}$ is the term vector for document i. The values of the semantic signature represent the extent to which a particular document's semantics match those of the k strongest semantic themes in the document collection.

Semantic signatures can be used to enhance conformance analysis. After LSI has been performed on a software project's documents, semantic signatures can be computed for each document and recorded. When a document is modified, its semantic signature can be recomputed. If the semantic signature changes substantially, this is evidence that the document's semantics have changed in an important way and that conformance problems are more likely. When new documents are added to a project, semantic signatures can be used to generate candidate traceability links for developer evaluation. Of course, when a project's scope or features are changed substantially, a new LSI computation should be performed and semantic signatures regenerated.

7 Conclusions

The paper presents a means to combine traceability link recovery with conformance analysis and inconsistency management. This work is a vital step towards long term maintainability of reverse engineered legacy systems.

By utilizing an IR based approach we can take advantage of the derived semantic similarity measure to determine a conformance rating of retrieved links. Additionally the hypertext model supports multi links and conformance analysis.

This work was supported in part by grants from the National Science Foundation (CCR-02-04175 and CCR-97-34102).

8 References

[1] Anquetil, N. and Lethbridge, T., "Assessing the Relevance of Identifier Names in a Legacy Software System", in *Proceedings Annual IBM Centers for Advanced Studies Conference (CASCON'98)*, December 1998, pp. 213-222.

[2] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A., "Information Retrieval Models for Recovering Traceability Links between Code and Documentation", in *Proceedings IEEE International Conference on Software Maintenance (ICSM'00)*, San Jose, CA, October 11-14 2000, pp. 40-51.
[3] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering*, 28, 10, October 2002, pp. 970 - 983.
[4] Antoniol, G., Canfora, G., De Lucia, A., and Merlo, E., "Recovering Code to Documentation Links in OO Systems", in *Proceedings 6th IEEE Working Conference on Reverse Engineering (WCRE'99)*, Atlanta, GA, October 6-8 1999, pp. 136-144.
[5] Berry, M. W., "Large Scale Singular Value Computations", *International Journal of Supercomputer Applications*, 6, 1992, pp. 13-49.
[6] Collard, M. L., Maletic, J. I., and Marcus, A., "Supporting Document and Data Views of Source Code", in *Proceedings ACM Symposium on Document Engineering (DocEng'02)*, McLean VA, November 8-9 2002, pp. 34-41.
[7] Conklin, J., "Hypertext: An introduction and survey", *IEEE Computer*, 20, 9, 1987, pp. 17-41.
[8] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, 41, 1990, pp. 391-407.
[9] Dick, J., "Rich traceability", in *Proceedings Automated Software Engineering*, Edinburgh, Scotland, 2002, pp.
[10] Dumais, S. T., "Improving the retrieval of information from external sources", *Behavior Research Methods, Instruments, and Computers*, 23, 2, 1991, pp. 229 - 236.
[11] Feldman, S., "Make: A program for maintaining computer programs", *Software Practice*, 9, 1979, pp. 255-265.
[12] Gotel, O. and Ginkelstein, A., "An analysis of the requirement traceability problem", in *Proceedings International Conference on Requirements Engineering*, Colorado Springs, Colorado, 1994, pp. 94-102.
[13] Grundy, J., Hosking, J., and Mugridge, W., "Inconsistency management for multiple-view software development environments", *IEEE Transactions on Software Engineering*, 24, 11, 1998, pp. 960-981.
[14] Gunter, C. A., "Abstracting dependencies between software configuration items", *ACM Transactions on Software Engineering and Methodology*, 9, 1, 2000, pp. 94-131.
[15] Knethen, A., "Automatic change support based on a trace model", in *Proceedings Automated Software Engineering*, Edinburgh, Scotland, 2002, pp.
[16] Landauer, T. K. and Dumais, S. T., "A Solution to Plato's Problem: The Latent Semantic Analysis Theory of the Acquisition, Induction, and Representation of Knowledge", *Psychological Review*, 104, 2, 1997, pp. 211-240.
[17] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in *Proceedings 23rd International Conference on Software Engineering (ICSE'01)*, Toronto, Ontario, Canada, May 12-19 2001, pp. 103-112.
[18] Maletic, J. I. and Valluri, N., "Automatic Software Clustering via Latent Semantic Analysis", in *Proceedings 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, Cocoa Beach Florida, October 1999, pp. 251-254.
[19] Marcus, A. and Maletic, J. I., "Identification of High-Level Concept Clones in Source Code", in *Proceedings Automated Software Engineering (ASE'01)*, San Diego, CA, November 26-29 2001, pp. 107-114.
[20] Marcus, A. and Maletic, J. I., "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing", in *Proceedings 25th IEEE/ACM International Conference on Software Engineering (ICSE'03)*, Portland, OR, May 3-10 2003, pp. 125-137.
[21] Munson, E., "The Software Concordance: Bringing Hypermedia to Software Development Environments", in *Proceedings SBMIDIA '99: V Simposio Brasileiro de Sistemas Multimidia e Hipermidia*, Anais. Goiania, Goias, Brasil, June 1999, pp. 1-12.
[22] Nguyen, T. N., Gupta, S. C., and Munson, E. V., "Three issues in the use of versioned hypermedia for software development systems", in *Proceedings Brazilian Symposium on Multimedia and Hypermedia Systems*, 2002, pp. 3-18.
[23] Nguyen, T. N., Gupta, S. C., and Munson, E. V., "Versioned hypermedia can improve software document management", in *Proceedings Conference on Hypertext and Hypermedia*, College Park, Maryland, 2002, pp. 192-193.
[24] Nuseibeh, B., Easterbrook, S., and Russo, A., "Leveraging inconsistency in software development", *IEEE Computer*, 33, 4, 2000, pp. 24-29.
[25] Pinheiro, F. and Goguen, J., "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*, 13, 2, 1996, pp. 52-64.
[26] Pohl, K., "PRO-ART: Enabling requirements pre-traceability", in *Proceedings International Conference on Requirements Engineering*, Colorado Springs, Colorado, 1996, pp. 76-85.
[27] Ramesh, B. and Jarke, M., "Toward reference model for requirements traceability", *IEEE Transactions on Software Engineering*, 27, 1, 2001, pp. 58-93.
[28] Reiss, S., "The Desert environment", *ACM Transactions on Software Engineering and Methodology*, 8, 4, 1999, pp. 297-342.
[29] Salton, G. and McGill, M., *Introduction to Modern Information Retrival*, McGraw-Hill, 1983.
[30] Spanoudakis, G. and Zisman, A., "Inconsistency management in software engineering: Survey and open research issues", in *Handbook of Software Engineering and Knowledge Engineering*, Chang, S. K., Ed., 2001, pp. 24-29.
[31] Spirtes, P., Glymour, C., and Scheines, R., *Causation, Prediction, and Search*, MIT Press, 2000.
[32] Toranzo, M. and Castro, J., "A comprehensive traceability model to support the design of interactive systems", in *Proceedings ECOOP Workshops*, 1999, pp. 283-284.
[33] W3C, "Document Object Model Range", Online at http://www.w3.org/TR/DOM-Level-2-Traversal-Range/ranges.html, 2000.
[34] Watkins, R. and Neal, M., "Why and how of requirements tracing", *IEEE Software*, 11, 4, 1994, pp. 104-106.
[35] Whitehead, E. J., *An Analysis of the Hypertext Versioning Domain*, University of California - Irvine, PhD Thesis, 2000.