Using Fine-Grained Differencing to Evolve Traceability Links

Bonita Sharif and Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242

{bsimoes, jmaletic}@cs.kent.edu

ABSTRACT

An approach to support the sustained evolution of traceability links is proposed and outlined. A fine-grained differencing approach on the link endpoints is used to maintain the links in a scalable manner. Here scalable refers to large software systems with thousands of links. Details of the link model and representation are given followed by the process used to evolve traceability links.

Categories and Subject Descriptors

D.2.7. **Software Engineering**: Distribution, Maintenance, and Enhancement – documentation, restructuring, reverse engineering and reengineering.

General Terms

Management, Design, Documentation.

Keywords

Traceability, traceability link model, software evolution

1. INTRODUCTION

During maintenance, the design documentation that is initially created slowly deteriorates and can become completely useless since it does not correctly reflect the functional state of the system. Design documents are usually outdated the second a line of source code is typed in and compiled. There are explicit and implicit traceability links that exist between the different software artifacts/models such as a UML class model and source code. The lack of maintaining design documentation in parallel with changes to source code causes traceability links present between these artifacts to decompose. The evolution of traceability links is an important process that attempts to keep these links in synch with the models during evolution. Existing change management systems and software configuration management do not solve the traceability link evolution problem. They mostly deal with versioning of software artifacts in repositories and the building of a system.

We address the issue of maintaining useful and valid relationships that exist between various software models during development. Establishing traceability links is a pre-requisite to evolving them. Our assumption is that a base set of preliminary links already exist that were manually created, generated via information retrieval methods [30] or through scenarios [12]. Here we focus on link evolution and not the construction or recovery of links.

This paper addresses some of the following questions. How do we evolve fine and coarse grained traceability links between software artifacts that are constantly evolving independently? How do we detect in a practical and scalable manner if a link is valid due to the independent evolution of system models? Our goal is to address the "Supporting Evolution" aspect of the grand challenges of traceability by addressing the above questions.

We take the position here that fine-grained differencing on the link endpoints (models) is a viable manner to support evolving links. Keeping the linkage between artifacts simple is another important aspect of successfully evolving traceability links. If the linking is too complex to edit or code it will not be adopted in an industrial setting. We feel this is a practical and simple way to solve traceability link evolution. In our previous work [21], we give a formal definition of traceability links and discuss issues involved with evolution.

The paper outlines our preliminary work on this subject and is organized as follows. Section 2 describes the link scheme we adopt, fine-grained differencing as well as sample links. The architecture and process of evolving links is given in Section 3. Section 4 describes related work in link evolution followed by our conclusions and future work.

2. PROPOSED LINK METHODOLOGY

In this section we discuss the types of models supported by our approach followed by a fine-grained differencing example. We also present a link model and a sample link.

Many different artifacts such as use cases, UML class diagrams, source code, test cases, and semi-structured requirement documents are used in the software development process. In order to maintain links between these heterogeneous documents, we need to have a structured way of representing them. We represent models (generated from the artifacts) in XML. For example,

TEFSE/GCT'07, March 22-23, 2007, Lexington, KY, USA.

source code is represented in srcML[19], an efficient lightweight markup of the source code. Class models are represented in classML, (a programmer-centric view). We do not use XMI since many different versions exist and the notation is not user friendly. Use-case models can be represented in a similar manner.

2.1. A Fine-grained Differencing Example

The central part of evolving links depends on the fine-grained differencing of models that lie on the endpoints of those links. This allows us to determine changes in the syntactic structures of models and thereby determine which traceability links are suspect. To illustrate our point, we use an example from fine-grained differencing of source code namely, srcDiff [8, 20]- which is also represented in XML. The src2srcDiff translator takes two versions of source code and generates a srcDiff document containing old, new, and common parts between these two document versions. Generating srcML is an essential intermediate step.

```
<!-- Function body -->
<diff:common>...
<function>
  <type><name>int</name></type>
  <name>
    <name>Mailbox</name>::
    <diff:old><name>Validate_Password</name></diff:old>
    <diff:new><name>Get_and_Validate_Password</name>
    </diff:new>
  </name>
  <parameter_list>()</parameter_list>
  <block>...</block>
 </function>...
<!-- At function call location -->
<call><name>
  <diff:old>Validate_Password</diff:old>
  <diff:new>Get_and_Validate_Password</diff:new>
</name><argument_list>()</argument_list></call>...
</diff:common>
<!-- In the header file -->
<diff:common>...
<function_decl><type><name>int</name></type>
  <name>
    <diff:old>Validate_Password</diff:old>
    <diff:new>Get_and_Validate_Password</diff:new>
  </name>
  <parameter_list>()</parameter_list>;
</function_decl>...
</diff:common>...
```

Figure 1. An excerpt from a srcDiff document showing a change in the name of a method.

An example from a mail system written in C++ shows the srcDiff generated when we refactor the code by changing a method name. See Figure 1. The function Validate_Password is changed to Get_and_Validate_Password. srcDiff uses its own *diff* namespace. In this particular example we do not show the `<diff:common>` area. This change affects three points in the source code model: the function body, function call site and the header file.

The differencing facility for source code namely srcDiff [8, 20], is used to generate fine-grained differences between two versions of code. Our approach is not limited to srcDiff. Any fine-grained differencing method can be used on the artifacts. If links between a use case and a UML model such as classML are needed, our approach would need the fine-grained differencing for the use case (useDiff) and for the UML model (classDiff). The process is the same in either case.

2.2. Link Model

This section discusses the components and semantics of the link model used in our approach. Our goal is to define traceability links with clear and simple semantics. Figure 2 describes the proposed link model. We borrow concepts from the XML Linking Language, XLink. We do not follow the XLink implementation scheme. We use our own tool to process links.

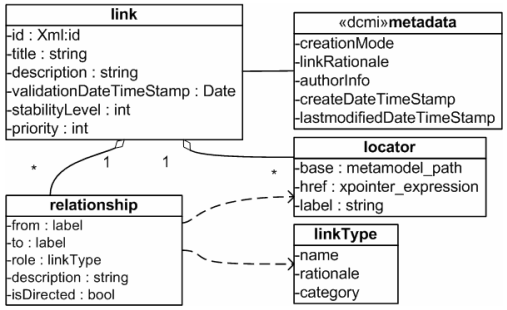

Figure 2. Proposed link model

We describe this simplistic model below. A traceability link is uniquely identified by its id represented in W3C's xml:id recommendation. The link title is a short description about the link as a whole. We call this the global description. The link description is a longer description about the link. The stability level of a link gives us an indication about the validity of a link. There could be many levels of stability starting with just two stable or unstable, each mapped to an integer value of 0 and 1 respectively. The priority attribute of a link indicates the importance of the link mapped to an integer value. Some links are more critical than others. Examples of priority levels are High, Low, or Normal. The levels of stability and priority are determined based on the software system and are extensible to include new levels. The validation stamp tells us the date and time the link was checked for stability. It does not indicate whether the link is stable or not.

The two crucial classes in our link model are the *locators* and *relationships*. A link has several locators associated with it. A locator consists of a base, href and a label. The base gives the path to a document within a model, href is an XPointer expression. Using XPath we can address paths within elements of our structured metamodels such as srcML. The label within a locator is used in the relationships to identify directionality.

A relationship consists of a source and a sink denoted by the *from* and *to* attributes. This corresponds to the arc feature within the XLink standard. The *description* gives a short explanation about the relationship. This description is referred to as the local description of a link in contrast to the global one mentioned above. The relationship attribute role determines the type of the link. For example, the role *implements* could exist between a design model and a source model. The *isDirected* attribute is used to determine if a link is directed or undirected. A '0' value indicates that the link is undirected. If this is the case, the *from* and *to* attributes do not have their usual meaning. They only refer to the two endpoints with no implication of directionality.

The creation method characteristic of a link is represented in the link metadata. The granularity characteristic of a link is defined in the href of a locator. The directionality characteristic of a link is defined in the relationship class. The multiplicity characteristic is

defined using both the locator and relationship class. Directionality is determined by the *from* and *to* attributes of a relationship whereas multiplicity (n-ary or binary) is determined by the number of locators defined and used in the relationship. The type characteristic of a link is represented in the role attribute of a relationship.

The linkType class describes link semantics. The attribute *name* within the linkType class is used in the *role* attribute of a relationship. The rationale stores justifications for the link. A link can have many types applied to it when appropriate given by the multiplicity of relationship classes with respect to a link in Figure 2. The set of link types is extensible based on business needs.

2.3. Link Storage and Representation

The storage of links is done external to the models in a set of files that comprise the link base. This decouples the semantics of the models and our link model thereby preserving the original models. The models are not dependent on the links. Storing the links externally also allows for different users to generate different views of the same link.

```
<links>
 <link id="myexample1" stability="1" priority="1"
     validation_datetime="1/14/2007 12:43:20EDT">
   <title>source and design method must match</title>
   <desc>A longer description may go here...</desc>
   <metadata>myexample1.dcmi.xml</metadata>
   <!-- Defining the locators -->
   <loc base="MailSys.classML.xml"
       href="//class[@name='Mailbox']/
       method[@name='Get_and_Validate_Password']"
       label="src" />
   <loc base="Mailbox.cpp.srcML.xml"
       href="//function
       [name='Mailbox::Get_and_Validate_Password']"
       label="dest" />
   <loc base="Mailbox.h.srcML.xml"
       href="//function_decl
       [name='Get_and_Validate_Password']"
       label="dest" />
   <!-- Defining the relationships -->
   <arc from="src" to="dest" role="must-agree"
       desc="These names must match" isDirected='1'/>
 </link>......
</links>
```

Figure 3. A link between classML and srcML

A traceability link is an n-tuple instantiating the link model presented in the previous section. Our links are represented in XML to support efficient/easy interoperability. To illustrate our approach, we use links between classML and srcML as examples from the same mail system we described earlier in Figure 1. See Figure 3 for an example of a link between classML and srcML.

This example is a n-ary directional link of type "must-agree" that links a classML document to two places in the source code, one where the function is declared in the header .h file and one to the function's name in the .cpp file. This link is stable (denoted by the value 1) and its priority is normal (denoted by the value 1). The XPath expressions described in the href attribute depend on the semantics of the srcML and classML models. The metadata of the link is given in a separate file named myexample1.dcmi.xml. In a similar manner, we can generate a link between a use case model and source code for a non-functional requirement. We can also generate a many-to-many or 1-to-many links using our link model. Our example points out the main components in our representation. Since source code represents the true functionality of a system at any point in time, we are mainly interested in fine grained links (both functional and non-functional) between design documents and source code.

3. EVOLVING TRACEABILITY LINKS

The process of maintaining traceability links in an efficient manner is an inherently difficult problem. This section answers the following question we posed in the introduction: How do we detect in a practical and scalable manner if a link is valid due to the independent evolution of system models? We consider fine grained differencing to be a step in the process of link evolution.

Dick [11] mentions four steps in implementing change management for traceability. First, identify which artifacts are affected by the change. Second, calculate the potential impact tree by processing the links. Third, prune and elaborate the impact tree to eliminate changes that don't propagate. Fourth, traverse the impact tree defining the change and finally apply the changes. We follow a similar approach which is presented below.

The architecture of our link evolution method is described in Figure 4. $Models_{v1}$ refers to a set of models in the system in a particular version. These models are structured based directly on the artifacts they represent. The link base consists of all the links and is stored external to the models. Links in the link base point to parts of the models in $Models_{v1}$ via XPath expressions (Step 1). Since models are constantly evolving independently (Step 2), we represent the evolved set of models as $Models_{v2}$. These two sets of models are then used by the model2modelDiff (Step 3), which is the fine-grained differencing step.

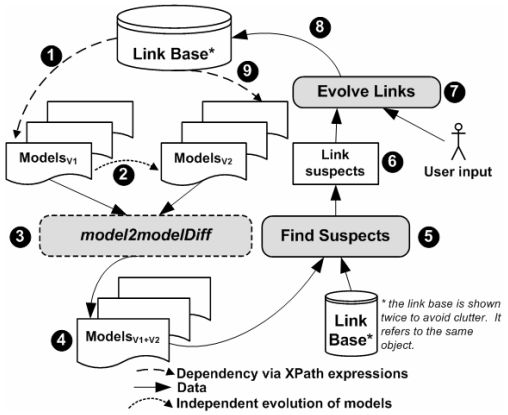

Figure 4. Architecture of the proposed link evolution method

The fine-grained differencing approach denoted by model2modelDiff generates a single document multiple version set of models with new, old and common parts between $Models_{v1}$ and $Models_{v2}$ (Step 4). The Find Suspects process (Step 5) reads this output along with the existing links and generates link suspects (Step 6). The Evolve Links process (Step 7) along with user input changes the link base (Step 8) which is reflected in the dependency shown by Step 9 in the diagram. User input might be required to make decisions about updating a link. There are cases where a link can be updated automatically without user input if a predetermined decision or policy exists.

Consider the following example that instantiates the general architecture described in Figure 4. If we are representing links between a UML model and a source model, we would run the src2srcDiff translator [20] on the original and evolved source

model and determine which parts of the source code changed. This information can then be used by Step 7 to change any suspect links from the source code model to the UML model.

Table 1 describes the process involved in generating a set of link suspects (Step 5 in Figure 4) in a scalable manner. The process of finding new and old sections in the single document multi-version fine-grained differencing output is done through traversing an XPath expression. For example, the XPath expression *//diff:new* gives a set of nodes that have been added anywhere in the document. Since *diff:old* marks deleted stuff in the new version, we need to check any XPaths in existing links that address this part. If we find such a link, it is marked as suspect. Finally we output all links that are found suspect. We can use the *diff:new* portion of $Models_{V1+V2}$, to help a user in making a decision about adding or deleting components in a link.

Table 1. Pseudocode for generating link suspects

Algorithm: Find Suspects *Input:* $Models_{V1+V2}$, Link base *Output:* A set of link suspects	
Step 1: Difference Selection	Use XPath to find changed locations in $Models_{V1+V2}$, denoted by <diff:new>, <diff:old> and <diff:common> sections
Step 2: Comparison	Compare the elements found in <diff:old> with the path in the link locators (denoted by href) of links in the link base.
Step 3: Suspect Selection	If there are link locators that address elements (using href) in a <diff:old> section, label this link as a suspect.
Step 4: Generation of Link Suspect Candidates	Output the list of links that are suspects along with diff:new sections showing what was added in the new version.

This approach is scalable since we do not go through the entire set of models. We go directly to the differencing information generated for us using the fine-grained differencing step. It has been shown in [8] that the size of and time to generate srcDiff are both reasonable. The space increase was less than 4 times the original source for srcDiff which results in the generation time to be in the order of seconds. The generation of srcML is also fast in the matter of a few minutes. Our experiences with evaluation of XPath expressions on an open source project with over 4,000 methods took less than 30 seconds. In its current state, src2srcDiff runs at approximately 6000 LOC per second and src2srcml runs at approximately 10,000 LOC per second. All these factors lead us to conclude that this approach is highly scalable.

We now address some questions that arise during link evolution. When do we run the system through the Find Suspects checker shown in Figure 4? The answer depends on the user. A user might want to run this check at predetermined points in time or after refactoring the design or code. Why does a link become a suspect? It could be that the location in model that the link refers to is no longer existent. What do we do when we determine such a link? We could suggest changes if feasible or we can let the user decide what to do. The user may wish to delete the link, modify the location the link points to, or add new locations to the link. We also take a tolerant approach to managing these links similar to xlinkit [24].

4. RELATED WORK

Several semi-automatic/automatic approaches [30] based on information retrieval [4, 10, 14, 15, 22, 28], probabilistic models [18], rules [31, 34], scenarios/test cases [12, 17] and events [5-7] are used to recover traceability links in software systems. However, there are only a handful of approaches available that address the crucial issue of managing link evolution since it is intrinsically non-trivial. We discuss these and related approaches below along with how our approach differs.

In a recent study, Murta et al. [23] developed the ArchTrace tool which takes a policy-centric approach to automatically updating traceability links between architecture and implementation. The links are updated after each change is committed to a repository. This work differs from our work in the following way. The links are coarse grained entities at the file level and maintained between source code files and components represented in xADL. In our approach due to fine grained differencing we are able to link to parts of a file. This reduces the amount of search space for a stakeholder wishing to understand a link.

Antoniol et al. [3, 4] build traceability links between two software releases of an object oriented system and point out differences between the releases. They use an intermediate representation AOL to represent design information recovered from code. Our approach handles a variety of different artifacts not just code. Another distinction is that our approach is based on source code models like srcML [19] and not an intermediate abstraction.

The traceability framework presented by Sherba et al. [29] has an evolution service that analyzes the changes to a set of relationships over time. It analyzes existing links in different versions without evolving the links themselves. von Knethen [32] presents a fine-grained conceptual trace model that determines types of artifacts to be traced in order to support impact analysis and requirements changes. Riebisch [27] introduce feature models that act as an intermediary in structuring traceability links between artifacts. Several hypermedia systems like the software concordance environment [25] and Chimera [2] also discuss maintaining and versioning links within a software development environment. Most of the systems are overly complex and do not address evolution of links in particular. Ivkovic et al. [16] use model synchronization techniques from model driven architecture to achieve traceability consistency. Models are represented using a graph metamodel. A transformation metamodel is then used to code model dependencies and equivalence relations are used to evaluate model synchronization. A change in a model is viewed as a combination of graph changes. This approach does not scale well.

Cleland-Huang et al. [6, 7] establish links using a event service that is maintained by requirements. Artifacts that want to link to a requirement need to subscribe to it using the event service. When a requirement changes, the linked artifact receives a notification message about the link semantics. The event handler continually searches for new events in the event list and notifies the link handler which in turn changes relationships between requirements as they occur. Our approach does not necessarily check for the validity of a link as soon as something changes. Testing for link suspects in our approach is done either by a user or at a fixed interval. The event based approach could work together with our approach if immediate checks are needed by certain systems.

A constraint based approach to traceability is taken by Reiss [26] and Nentwich et al. [13, 24]. Reiss [26] views software artifacts as imposing constraints on each other. Meta-constraints are rules that are defined on the set of artifacts. SQL queries are then

generated for constraints based on meta-constraints. An integration mechanism, based on constraints, keeps the different artifacts consistent during development. Abstractors generate required information from software artifacts, XMI is used as an intermediate representation (abstraction) for class diagrams. For incremental update, a file consisting of commands describing what should be updated is generated. This work detects changes in artifacts at the file level.

Nentwich et al. present xlinkit [24] that manages consistency of distributed and heterogeneous documents. It is based on XLink and is responsible for generating consistency links and optionally suggests repair actions. A constraint language, CLIX- based on first order logic and XPath, expresses relationships between elements in multiple documents. An application of xlinkit to check consistencies between UML models is given in [13]. We have not seen xlinkit been applied to evolution of links and believe this is due to its generality. They use XMI and JavaML as models of design and code respectively. Creating paths from XMI is tedious due to the nature of its representation. This approach does not scale well. Our work is closely related to the above two approaches [24, 26].

Zisman et al. [33] build consistency rules between distributed documents using XPointer and present a set of consistency rules. Alves-Foss et al. [1] describe traceability between design (XMI) and code (JavaML) using an XLink link base. The links are rendered in HTML using XSL and can be traversed via hyperlinks. Conte de Leon et al. [9] describe TraceML, a XLink based language to specify relationships between artifacts.

Our approach generates traceability links directly between structured meta-models (such as srcML and classML) and not between intermediate abstractions. Another distinction is that we support fine-grained traceability links. The links presented in [24, 26] are between these abstractions and not directly between the artifacts. Their abstractions are not easily converted back to the main artifact. In our case, we can generate source code using the *srcML2src* translator [19] and not loose any information.

5. CONCLUSIONS AND FUTURE WORK

An approach to evolve traceability links using fine-grained differencing of artifacts is outlined. Our goal is to create a practical and simple to use traceability tool to meet the needs of a stakeholder. To support our approach, we develop a simple link model and a representation to store links. We provide an answer to the two questions we posed in the introduction. The evolution of fine and coarse grained traceability links is done via fine-grained differencing. The process of detecting suspect traceability links in a scalable manner is also presented. We address both space and time issues to translate models into XML with evidence of scalability.

We are in the process of developing a traceability tool that focuses on link evolution and implements the approach presented in this paper. A large case study is being developed on an open source system which will serve as our test bed. Our assumption is that we start with a base set of existing links. Our plan is to use one of the information retrieval techniques to generate links among various software artifacts (such as use cases, UML class diagrams and code) and then use our fine-grained differencing approach to maintain them over subsequent releases. Complementary to this, we are also building our own XQuery based language to generate our initial set of links between artifacts. Getting our hands on good design documentation is virtually impossible since it does not exist in most cases. We would probably need to generate design documentation as well from existing descriptions of the system. The next step is to compare the evolved links with the actual links recovered at a later release. This will give us an idea of how well the evolving mechanism works. This test bed will also contribute to the measurement and benchmarks challenges.

6. REFERENCES

[1] Alves-Foss, J., deLeon, D. C., and Oman, P., "Experiments in the Use of XML to Enhance Traceability between Object-Oriented Design Specifications and Source Code", in Proceedings of 35th Annual Hawaii International Conference on System Sciences (HICSS'02), 2002, pp. 276-284.

[2] Anderson, M. K., Taylor, N. R., and Whitehead, J. E., "Chimera: Hypermedia for Heterogeneous Software Development Environments", *ACM Transactions on Information Systems*, vol. 18, no. 3, July 2000 2000, pp. 211-245.

[3] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A., "Maintaining Traceability Links During Object-Oriented Software Evolution", *Software - Practice and Experience*, vol. 31, no. 4, 2001, pp. 331-355.

[4] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering*, vol. 28, no. 10, October 2002, pp. 970-983.

[5] Cleland-Huang, J., Chang, C. K., Sethi, G., Javvaji, K., Hu, H., and Xia, J., "Automating Speculative Queries through Event-Based Requirements Traceability", in Proceedings of IEEE Joint International Conference on Requirements Engineering (RE), Sept 2002, pp. 289-296.

[6] Cleland-Huang, J. and Chang, K. C., "Supporting Event Based Traceability through High-Level Recognition of Change Events", in Proceedings of 26th Annual International Computer Software and Applications Conference, Oxford, England, Aug 2002, pp. 595-600.

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

[8] Collard, M., Kagdi, H., and Maletic, J. I., "Factoring Differences for Iterative Change Management", in Proceedings of 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), Philadelphia, Pennsylvania USA, September 27-29 2006, pp. 217-226.

[9] DeLeon, D. C. and Alves-Foss, J., "Experiments on Processing and Linking Semantically Augmented Requirements Specifications", in Proceedings of 37th Hawaii International Conference on System Sciences (HICSS'04), 2004, pp. 90279b.

[10] DeLucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Enhancing an Artefact Management System with Traceability Recovery Features", in Proceedings of 20th IEEE International Conference on Software Maintenance, Chicago, Illinois, September 2004, pp. 306-315.

[11] Dick, J., "Design Traceability", in *IEEE Software*, 2005, pp. 14-16.

[12] Egyed, A., "A Scenario-Driven Approach to Trace Dependency Analysis", *IEEE Transactions on Software Engineering*, vol. 29, no. 2, 2004, pp. 116-132.

[13] Gryce, C., Finkelstein, A., and Nentwich, C., "Lightweight Checking for UML Based Software Development", in Proceedings of 2002 Workshop on Consistency Problems in UML-Based Software Development, 2002, pp. 124-132.

[14] Hayes, J. H., Dekhtyar, A., and Osborne, J., "Improving Requirements Tracing via Information Retrieval", in Proceedings of 11th IEEE International Requirements Engineering Conference (RE), Washington, D.C, USA, September 2003, pp. 138-147.

[15] Hayes, J. H., Dekhtyar, A., and Sundaram, S., "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods", *IEEE Transactions on Software Engineering*, vol. 32, no. 1, January 2006, pp. 4-19.

[16] Ivkovic, I. and Kontogiannis, K., "Tracing Evolution Changes of Software Artifacts through Model Synchronization", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Sept 2004, pp. 252-261.

[17] Leite, J. and Breitman, K., "Experiences Using Scenarios to Enhance Traceability", in Proceedings of 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '03), 2003.

[18] Lin, J., Lin, C. C., Cleland-Huang, J., Settimi, R., Amaya, J., Bedford, G., Berenbach, B., Khadra, O. B., Duan, C., and Zou, X., "Poirot: A Distributed Tool Supporting Enterprise-Wide Traceability", in Proceedings of IEEE International Conference on Requirements Engineering (RE'06) - Tool Demo, Sept 2006.

[19] Maletic, J. I., Collard, M., and Marcus, A., "Source Code Files as Structured Documents", in Proceedings of Proceedings of the 10th IEEE International Workshop on Program Comprehension (IWPC 2002), Paris, France, June 26-29 2002, pp. 289-292.

[20] Maletic, J. I. and Collard, M. L., "Supporting Source Code Difference Analysis", in Proceedings of 20th International Conference on Software Maintenance (ICSM), Chicago, Illinois, September 11-17 2004, pp. 210-219.

[21] Maletic, J. I., Collard, M. L., and Simoes, B., "An XML-Based Approach to Support the Evolution of Model-to-Model Traceability Links", in Proceedings of 3rd ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'05), Long Beach CA, Nov. 8 2005, pp. 67-72.

[22] Marcus, A. and Maletic, J. I., "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing", in Proceedings of 25th IEEE/ACM International Conference on Software Engineering (ICSE'03), Portland, OR, May 3-10 2003, pp. 125-137.

[23] Murta, L. G. P., Andre, v. d. H., and Werner, C. M. L., "ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links", in Proceedings of 21st IEEE International Conference on Automated Software Engineering (ASE'06), 2006, pp. 135 - 144.

[24] Nentwich, C., Capra, L., Emmerich, W., and Finkelstein, A., "xlinkit: A Consistency Checking and Smart Link Generation Service", in Proceedings of ACM Transactions on Internet Technology, 2002, pp. 151-185.

[25] Nguyen, N. T. and Munson, E., "The Software Concordance: A new Software Document Management Environment", in Proceedings of 21st annual international conference on Documentation, ACM Special Interest Group for Design of Communications SigDoc'03, San Francisco, California, USA., October 12-15 2003, pp. 198-205.

[26] Reiss, P. S., "Incremental Maintenance of Software Artifacts", *IEEE Transactions on Software Engineering*, vol. 32, no. 9, 2006, pp. 682-697.

[27] Riebisch, M., "Supporting Evolutionary Development by Feature Models and Traceability Links", in Proceedings of 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04), Brno, Czech Republic, 2004, pp. 370-377.

[28] Settimi, R., Cleland-Huang, J., Khadra, O. B., Mody, J., Lukasik, W., and DePalma, C., "Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts", in Proceedings of 7th International Workshop on Principles of Software Evolution (IWPSE) Kyoto, Japan, Sept 6-7 2004, pp. 49-54.

[29] Sherba, S., Anderson, A., and Faisal, M., "A Framework for Mapping Traceability Relationships", in Proceedings of 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '03), October 2003.

[30] Spanoudakis, G. and Zisman, A., "Software Traceability: A Roadmap", in *Handbook of Software Engineering and Knowledge Engineering*, Chang, S. K., Ed. World Scientific Publishing Co, 2005, pp. 395-428.

[31] Spanoudakis, G., Zisman, A., Perez-Minana, E., and Krause, P., "Rule-based generation of requirements traceability relations", *The Journal of Systems and Software*, vol. 72, no. 2004, 2004, pp. 105-127.

[32] von Knethen, A., "Change-Oriented Requirements Traceability. Support for Evolution of Embedded Systems", in Proceedings of International Conference on Software Maintenance (ICSM 02), 2002.

[33] Zisman, A., Emmerich, W., and Finkelstein, A., "Using XML to Build Consistency Rules for Distributed Specifications", in Proceedings of International Workshop on Software Specifications & Design (IWSSD), 2000, pp. 141-148.

[34] Zisman, A., Spanoudakis, G., Perez-Minana, E., and Krause, P., "Tracing Software Requirements Artifacts", in Proceedings of 2003 International Conference on Software Engineering Research and Practice (SERP'03), Las Vegas, Nevada, USA, 2003, pp. 448-455.