An XML Based Approach to Support the Evolution of Model-to-Model Traceability Links

Jonathan I. Maletic, Michael L. Collard, Bonita Simoes

Department of Computer Science
Kent State University
Kent Ohio 44242

{jmaletic, collard, bsimoes}@cs.kent.edu

ABSTRACT

The paper summarizes the authors' current research on supporting model-to-model traceability. The authors present a graph theoretic definition of what they mean by models and traceability links. This definition is realized by the use of XML technologies to represent the models and traceability links. Practical means to represent different types of system models (e.g., source code and design) using XML are discussed. Traceability links are also implemented using XML technologies in an efficient and scalable manner. The evolution of the system, along with the traceability links, is supported by a fine-grained versioning technique. This allows for versioning and differencing of specific elements of the models versus just lines or whole files.

Categories and Subject Descriptors

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

General Terms

Design, Documentation

Keywords

Traceability, software evolution, maintenance.

1. INTRODUCTION

A software system is comprised of numerous artifacts. For the most part, these artifacts are models of the system at some level of abstraction (e.g., requirements models, specifications, design models etc.). We also view the source code as a model of the system (i.e., a concrete model). Some of the initial relationships between two models are realized via instantiation. For example, an abstract-design model is realized as source code by a programmer. Other types of relationships exist between the requirements model and the design and source models. Parts of a design are relevant to some non-functional requirement or cross-cutting concern. Saving and representing these relationships between models allows one to understand what parts of one model related to another model. The ability to trace back and forth between models, in the context of these relationships, is commonly viewed as a critical feature for the evolution and maintenance of large scale high quality software systems [5, 21].

TEFSE 2005, November 8, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-59593-243-7/05/0011…$5.00.

Unfortunately these relationships between models, or traceability links, need to be evolved and maintained as the system evolves. Traceability links are costly to construct. Initial sets of links can be constructed automatically via forward-engineering tools. More likely in today's commercial software development setting, these types of tools were not used. Either traceability links can be (are) constructed manually or in conjunction with tools that assist in the recovery of links. Unfortunately, these recovery tools are far from perfect and require manual tuning and assessment of generated links [3, 25].

Given this we feel the only practical approach is to maintain and evolve the existing traceability links as the system evolves. So the goal of our work is to develop a robust and flexible representation of model-to-model traceability links that is evolvable along with the system models. Our approach uses an XML-based representation of both the models and the links. Link evolution is supported by a fine-grained differencing approach that allows us to determine exactly what syntactic structures changed within a model. Current differencing approaches work at the level of file or line. Our approach allows one to work at any syntactic level (statement, function, class diagram, use case, etc.).

In previous work [22] we presented methods for traceability link conformance using a hypertext model allowing complex linking as well as versioning of links. The work presented here uses that as a starting point and extends to practical implementation issues. Here we are specifically interested in source code to class model traceability. We are using an XML representation of the source-code model that we've developed called srcML [10, 24]. srcML embeds XML tags concerning syntactic information into the source-code document while maintaining all of the original source. Additionally we are developing an XML representation for class models called classML. Like srcML it is a user-centric representation that supports addressing and analysis of the model. This is opposed to XMI that is solely for UML model exchange. Our work on meta-differencing [9, 23] is then used to identify specific changes to determine what traceability links are involved.

The paper is organized as follows. First we present a set of definitions to clarify our terminology and meaning of traceability. We then discuss our representation of the models and links. Following that is a presentation of how we support evolution of the traceability links. Related work is covered in the Section 5. Section 6 concludes the paper and presents future work.

2. DEFINING TRACEABILITY

The following set of definitions attempts to formalize a number of the concepts regarding model-to-model traceability. We feel that a detailed description of our interpretation of the general traceability problem is required to compare our research with other related projects.

Definition: A *traceability link l* is an edge between two system models *M* and *N*, where $M \neq N$ and $l(M_j, N_k, D)$ where M_j and N_k are subsets of models *M* and *N* respectively. *D* determines the directionality of the link which could be single or bi-directional along with undirected. When directionality is present, ordering is done from left to right.

Traceability links are n-ary and are defined for linking multiple source anchor nodes to multiple target nodes. If directionality exists between multiple source or target nodes, it should be the same. We cannot have one link going from left to right and another pointing from right to left for the same link node.

The set of all traceability links of a system is $L = \{l_1, l_2, \ldots l_p\}$.

Definition: A *traceability link type*, *t* denotes the type of a link. We can represent it as $l(M_j, N_k, D) \times t$. A link type could be *causal*, *non-causal* or *navigational*. A *causal* link is always directional. *Non-causal* links and *navigational* links could be uni-directional or bidirectional. Traceability links should support different types. If three types for the same link are needed, then three links with different types are created. See Figure 1 for the basic types of links shown between a UML class model and a source code model.

Definition: A model *M* is defined as a finite set of nodes, $\{m_1, m_2, \ldots, m_k\}$ where each node corresponds to some distinct part of the model. Anchors can be defined on nodes. A node can participate in one or more anchors in a hypertext model.

These nodes represent anchors in the terminology of hypermedia. A *model* is a document(s), set of diagrams (e.g., class model, use case model, etc), or source-code program. A model is represented as a set of parts or components identified by the underlying definition of the model. At this time, we view relationships between two parts of the same model to be a component of the model and not a traceability link. A model is some sort of structured data. For example, source code could be constructed into a source model. Similarly, class design could be represented as a (structured) class model.

Definition: A subset of a model is denoted by M_i, where $M_i \subseteq M$. It consists of a subset of the nodes present in *M*.

Definition: A *traceability graph* is a directed graph, $G = (V, L)$ with nodes $V = |M| \cup |N|$, where *M* and *N* are system models. *L* denotes the set of traceability links between *M* and *N*. They are ordered pairs defined over subsets of *M* to subsets of *N*, i.e., $L \subseteq 2^M \times 2^N$. As such the maximum number of traceability links is $2^{|M|} * 2^{|N|}$ and not just $M*N$.

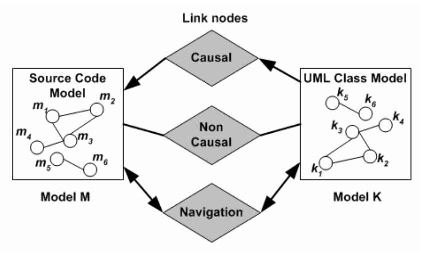

Figure 1 An example of different types of links between two models. On the left is a source-code model and on the right is a class model. Both models are connected graphs. The traceability links are also represented as a connected graph between the models

The conformance value of a link gives a confidence of how valid a given link is with regards to the conformance of the relationship between the two models described by the link. This *conformance rating* denoted as *CR* is a derived attribute of a link. It is dependent on the tool analyzing the links for conformance.

The next section discusses the traceability framework and its sub-systems.

3. REPRESENTATION

Our traceability representation is built on XML for both data representation and processing. On the data side XML representations, e.g., XML, XLink, XPointer, etc., are used for both the models and the traceability links. On the processing side XML tools and technologies, i.e., DOM, SAX, XSLT, etc., are used for creation, validation, and evolution of links. This section describes the traceability framework in terms of the definitions given in the previous section. Broad issues relating to our infrastructure will also be discussed. For a specific example, traceability links between a UML class model and source code is used.

3.1. Model Representation

Models are represented in XML with no restrictions as to the content, organization, or schema. This allows for full interoperability and flexibility of models in our approach including document-oriented models, e.g., DocBook, XHTML, etc. and data-oriented models, e.g., UML, ASTs, etc. Specifically, there are no restrictions (such as those made in XML Schema) on the use of certain XML features, e.g., mixed content.

The representation utilizes external links from-and-to the models. By using external links, no requirements are put on the schema of the models and as such the links and models are completely decoupled. Storing links externally allows for the straightforward storage of many-to-many relationships. It also allows for the creation of multiple link collections on the same models.

Traceability to source code is important for full utilization of the artifacts used in actual software development. Because the representation of source code is as plain text, an approach is needed in our infrastructure to provide an XML view of source code. Although many XML-based data-oriented representations exist, they do not provide full transparency to the original source code. For this reason a document-oriented XML representation of source code, srcML, (which we have developed) is used. In the srcML representation the source-code text is wrapped in XML elements. This allows the original source-code document to be addressed and transformed in XML with no loss of information including comments, white space, etc.

The infrastructure is not dependent on a srcML view of source code and any other model (including an AST-based model) can be used. However, the use of srcML demonstrates how traceability to source-code text can be achieved. In addition, it demonstrates how traceability can be achieved to non-XML documents whose original form must be preserved. The srcML representation currently supports C, C++ and Java.

3.2. Link Representation

As discussed in the previous subsection, the model representation is completely separate from the link representation. Since the system models are represented in XML, we leverage XPath to refer to addresses in the models forming a path between two or more models.

XPath (the XML Path Language) can select nodes from an XML document regardless of how they are ordered in XML hierarchies. This provides a standard, semantically rich way of expressing locations in the models. It provides pointer-like descriptions of locations, for example the XML element with id value of five has the expression **[@id='5']*. This also provides very intuitive physical description of the location, e.g., the first child of the second child of the root has the expression *1/2/1*, along with a very semantically-expressive description of the location, e.g., the condition of the first if-statement of a function has the expression *function/block/if/condition* in srcML. The exact form of XPath used depends on the semantics of the models involved and the method used to form the link.

An outcome of this lack of restrictions is that the model semantics remain inside the model. This allows for the use of links that are completely decoupled from the semantics of the models that they are linking. A full set of link types can be created and supported, and then applied to multiple applications or to multiple models.

The location addressing in the models is only part of the link representation. The meta-data of the link, e.g., type, timestamp, id, etc, must be stored. In addition the links must be collected. In order to store this information XLink, the XML Linking Language, is used. XLink specifies how to store the meta-data about the links. It also describes the use of linkbases which are external (to the document) collections of links.

One important piece of meta-data about a link is a timestamp which provides for the versioning of a link. This can be directly stored as meta-data on the link. A better solution would be to use a native XML database that provides for versioning. In this case, the timestamp is external to the link. This issue is still pending more investigation.

3.3. Scalability

Of great concern is the scalability of this framework. First we are concerned with the amount of storage individually needed for the models. The storage of the models is not an issue if they were originally stored in XML. However, if model conversion to XML is required then this can become an issue. Source code can be a particular problem since XML representations of source code can be hundreds of times larger than the original text. However, the document-oriented approach taken with srcML does not have this side effect. In practice we have observed an increase of only a single-digit multiple of the srcML over the original source code text. If this modest increase in space is still a problem the source code can remain as text and the srcML can be generated as needed. The srcML translator, *src2srcml*, can convert source-code in a text format to srcML in a stream-oriented (i.e., SAX) approach. In practice we have seen the translator work at 11 KLOC per second. The entire Linux kernel containing 161MB of text can be translated to a srcML representation in twenty minutes.

The storage of the links may create an issue depending on the number of links. At a minimum each link will contain an XPath expression with space needed for the meta-data about the link. An external linkbase using XLink is not designed for compactness. However the expressive nature of XPath provides some space efficiency since a single link can refer to many locations in a model.

Large numbers of links also provides a potential time efficiency problem. Applications such as validation may require traversal of large number of links. In addition, it may be required to follow a link in reverse. For example, for a given location in the source code it may be necessary to find which links point to it from the classML model. For a large number of links this may be very time consuming. However, our recent experience with the evaluation of XPath expressions on srcML has shown that this should not be a serious problem. We evaluated the equivalent of eight links to every method of an entire source-code project (over 4,000 methods), with the entire process taking under 30 seconds.

3.4. Applications

The creation, editing, and deletion of links require changes to the XML link representation. Because of the use of XML format, this becomes a form of XML editing with applications built using standard XML tools, e.g., XSLT, DOM, SAX, etc. They may be made programmatically, e.g., as the result of analysis, or they may be made by user action, e.g., selection using a traceability selection tool.

An application on traceability links can be applied at two levels. The lowest level works directly in the framework with no knowledge of the models, e.g., deletion of a link. These applications are portable to other models. At a higher level a tool can be written that uses knowledge of the models, e.g., validation.

4. EVOLUTION

The evolution of system models creates difficulties with respect to the maintenance of traceability links. While a small number of outdated links may be tolerated, a high rate of change in the models increases the percentage of outdated links. If the granularity of change is too large, the detection of outdated links will have too low a precision leading to the unnecessary elimination of valid links. The inability to deal effectively with model changes is one of the most important problems facing traceability.

We see the fundamental problem to be the following: How does one detect that the validity of a link is suspect due to evolution of the system model? This issue requires the examination of both the link and the change. Each may have different levels of granularity. The granularity of a link, in our representation, is determined by what types of XML subtrees the link can individually refer to. For source code this can be elements such as file, class, method, statement, or use/definition of an identifier. The granularity depends on the method used to create the link.

The granularity of change detection is at the level a difference can be detected. This depends on the change identification mechanism and may be at a physical (e.g., file and line) or syntactic level (e.g., class, statement). The granularity of change detection is also quite dependent on the efficiency of the algorithm. Textual differencing determines physical differences between text as in the commonly used utility *diff* which produces physical differences in terms of lines and columns. By applying the LCS (Longest Common Subsequence) [18] on the text in the lines of the file it is efficient and robust. It can be applied to source code in any language and in any state, and to any text format. However, it is line based and crosscuts syntactic structure. Semantic differencing detects changes to the computation of the entire program. Heuristics are used since the problem in undecidable and computationally expensive. One problem with semantic differencing is that differences can be detected even though a textual change has not been performed, e.g., a change to the type in the declaration of a variable.

In order to support traceability, the granularity of the links and changes must match. Textual links require textual change detection. Syntactic links require syntactic change detection. So for example change detection with a granularity of a method will unnecessarily detect changes for links that have a granularity of a statement in that method.

For the complete support of traceability during evolution fine-grained syntactic differencing is required. The primary application of syntactic differencing has been with merging [26] as was done in [17] where the LCS was applied to parse-tree sequences. However, this approach has difficulties with non-language constructs, e.g., comments and preprocessor directives [16]. Recently an approach has been used on AST's generated from source code [29], however this approach has time complexity problems.

Our work on this problem [9] has produced the concept of meta-differencing, which provides an infrastructure for analysis of fine-grained source code difference. The infrastructure is built on an XML representation of multiple versions of a source-code document. Utilizing XML tools allows the determination of fine-grained syntactic differences, e.g., changes to individual statements, conditions, etc. While the tool was originally developed specifically for source-code changes it can be applied to any XML document. So meta-differencing of class models represented in XMI or classML can also be done. The meta-differencing tool also allows one to analyze the differences through queries. For example, one can write a simple query to find all changes to loop conditionals or formal parameter lists.

With this we can support evolution of links in conjunction with evolution of the models. Consider a link from a class model to a source model. The link is from a particular relationship in the class model and is reflected in a variety of statements in the source. If the statements in the source code are changed, then the link is suspect and needs to be reevaluated and possibly modified.

This leads us to a number of different types of changes that could occur all at different levels of granularity. A *physical-change* is the detection of a change at the textual level. A *file-change* is the detection of a change at the file level and causes all links to that file to be suspect. This may be very imprecise and cause a large number of links to be suspect, even if a single statement is changed. A *line-change* is changes to individual lines, e.g., the output of the utility *diff*. While at a lower level of granularity, line changes cannot be easily compared to syntactic elements such as individual statements. Further analysis is needed to determine whether the line changes occur in a particular element.

A *syntactic-change* agrees in type with a syntactic link and is the detection of a change at the syntactic level. A *class-change* is a good match if the syntactic-level link is also at a class. However, the granularity difference is still too large if the link has a finer granularity and would cause all links to that class to be suspect. Lower levels, such as *method-change* or *statement-change* have the same issues, except the number of misidentified suspect links is smaller. A *sub statement-change* is the change to part of a statement, e.g., condition, type or name change. If the level of the link is lower-level, then the set of suspect links would be as minimal as possible. Other elements that are not in a typical AST, e.g., white space, comments, preprocessor statements, etc., can also serve as the source/destination of a link and have differences detected to them.

5. RELATED WORK

The need for a framework to maintain the various dimensions of software development as they evolve is discussed in [31]. Constraints are placed on the various dimensions to have the different dimensions of software consistent during development. An integration mechanism, based on constraints, keeps the different artifacts consistent. In more recent work [30] elaborates on the methods used. Class diagrams are represented in XMI and links to source code are set up. The work by [1] on tracing of requirements to Use Cases aims at designing a custom toolkit that can be used with a requirements-management tool such as DOORS [11].

Egyed [12, 13] recognizes the separation of software models and source code and has a tool called *TraceAnalyzer* that makes use of test cases to generate trace information during program execution. Palmer [27] introduces traceability and enumerates steps to achieve traceability in a large complex software system. He describes traceability management spanning across the software-development cycle in order to uniquely identify links.

Zisman et al. [36] use traceability rules for automatic generation and maintenance of traceability relations. XML is used to describe rules and artifacts. They define three types of traceability relations; *overlaps*, *realizes* and *requires*. Traceability relations are built if the rules are satisfied. They are only concerned with tracing different requirements artifacts to other artifacts. Spanoudakis and Zisman [34, 35] present a survey in managing inconsistencies in software models. Requirements traceability using contribution structures as used in [15] keeps track of who contributed to which part of the requirements. Leite et al. [20] describe a scenario-based framework as their traceability strategy. These scenarios are described in XML. TOOR [28] is a tool used to trace requirements in an object-oriented fashion and is based on a database-management system. Huang [8] describes an approach to create traceability links between non-functional requirements, design, and code using design patterns as mediators. Dynamic links are maintained as publish or subscribe relationships.

The idea of relating hypertext [32] to traceability can be traced back to 1989 [4]. RayTracer [14] is a tool that is able to maintain traceability links. It allows each user to create and maintain their links from data and design. TraceM [33] is a framework that automates traceability relationships. It converts implicit relationships that exist into explicit ones that are easier to comprehend and visualize. Method names from source code are used to map from design to code. Chimera [2] is an open hypermedia system supporting software-engineering tasks. Links are considered to be first-class objects. A view has a set of anchors in an object. A link is a set of anchors in views. Viewers display objects and the Chimera client manages links. It supports n-ary links but there is no directionality defined in the linkage. Versioning is not incorporated either.

Cimitile et *al*. [7] define a traceability relation that keeps track of links among object models along with information related to decisions as to why the link exists. RETO [19], is a requirements-engineering tool that supports traceability. It stores a traceability rules catalog that defines strong and weak traceability relationships. Rules are based on the type of conceptual model.

Numerous impact analysis [6] methods have been proposed in the literature however, they do not directly address the issue of traceability.

6. CONCLUSIONS & FUTURE WORK

The approach presented here has a number of advantages with regards to interoperability and flexibility. Using an XML representation for the software artifacts and models allows for linking models in any manner. Elements in one model can be linked to any element, at any level of granularity, in another model.

Translating models into an XML representation has proven to be relatively efficient. Most UML tools export to XMI and this can be used directly or translated into other more problem-specific representations. However, one must construct the translator. In the case of source code we have already done this and have a robust and useable tool.

Our approach to evolve the links along with the evolving models is to detect syntactic changes at the same level and type as the link. As changes are made, the smallest set of links that are subject to change can then be examined.

7. ACKNOWLEDGEMENTS

This work was supported in part by a grant from the National Science Foundation (C-CR 02-04175).

8. REFERENCES

[1] Alexander, I. SemiAutomatic Tracing of Requirement Versions to Use Cases Experiences & Challenges in Proceedings of 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '03) (Montreal, Canada, October 7, 2003).

[2] Anderson, M. K., Taylor, N. R., and Whitehead, J. E. Chimera: Hypermedia for Heterogeneous Software Development Environments. ACM Trans. on Information Systems, 18, 3 (2000), 211-245.

[3] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A. Identifying the Starting Impact Set of a Maintenance Request: a Case Study in Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'00) (Zurich, Switzerland, February 29 - March 3, 2000), 227-230.

[4] Balzer, R., Begeman, M., Garg, P., Schwartz, M., and Scheiderman, B. Hypertext and Software Engineering in Proceedings of Hypertext '89 Proceedings (1989), 395-396.

[5] Bianchi, A., Visaggio, G., and Fasolino, A. An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models in Proceedings of 8th International Workshop on Program Comprehension (IWPC'00) (June 10 - 11, Limerick, Ireland, 2000), 149.

[6] Bohner, A. S. and R., A. S. Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA, USA., 1996.

[7] Cimitile, A., Lanubile, F., and Visaggio, G. Traceability Based on Design Decisions in Proceedings of Conference on Software Maintenance (1992), 309-317.

[8] Cleland-Huang, J., Chang, C., and Christensen, M. Robust Requirements Traceability for Handling Evolutionary Change in Proceedings of IEEE Transactions on Software Engineering (September, 2003), 796-810.

[9] Collard, M. L. Meta-Differencing: An Infrastructure for Source Code Difference Analysis. Kent State University, Kent, Ohio USA, Ph.D. Dissertation Thesis, 2004.

[10] Collard, M. L., Kagdi, H. H., and Maletic, J. I. An XML-Based Lightweight C++ Fact Extractor in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03) (Portland, OR, May 10-11, 2003), 134-143.

[11] DOORS, Date Accessed: 03/28/2004, http://www.telelogic.com/, 2003.

[12] Egyed, A. Trace Observer: A Reengineering Approach to View Integration. Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781 USCCSE-99-517, 1999.

[13] Egyed, A. A Scenario-Driven Approach to Traceability in Proceedings of 23rd International Conference on Software Engineering (ICSE) (Toronto, Canada, May 2001, 2001), 123-132.

[14] Gardner, F. RayTracer: Traceability for Software Engineering in Proceedings of Third Symposium on Assessment of Quality Software Development Tools (June 7-9 1994, 1994), 224-232.

[15] Gotel, O. and Finkelstein, A. Extended Requirements Traceability: Results of an Industrial Case Study in Proceedings of IEEE International Symposium on Requirements Engineering (1997), 169-178.

[16] Hunt, J. J. Fast Semi-Semantic Differencing and Merging. Web page, Date Accessed: 02/01/2004, http://wwwswt.fzi.de/cocoon/mount/swt/mitarbeiter/jjh/, 2004.

[17] Hunt, J. J. and Tichy, W. F. Extensible Language-Aware Merging in Proceedings of IEEE International Conference on Software Maintenance (ICSM'02) (Montreal, Canada, October 3-6, 2002), 511-520.

[18] Hunt, J. W. and Szymanski, T. G. A Fast Algorithm for Computing Longest Common Subsequences. CACM, 20, 5 (May 1977), 350 - 353.

[19] Insfrán, E. A Requirements Engineering Approach for Object-Oriented Conceptual Modeling. Polytechnic University of Valencia, Spain, 2003.

[20] 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).

[21] Lindvall, M. and Sandahl, K. Practical Implications of Traceability. Software Practice and Experience, 26, 10 (1996), 1161-1180.

[22] Maletic, J., Munson, E., Marcus, A., and Nguyen, T. Using a Hypertext Model for Traceability Link Conformance Analysis in Proceedings of Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '03) (Montreal, Canada, October 7th, 2003, 2003), 47-54.

[23] Maletic, J. I. and Collard, M. L. Supporting Source Code Difference Analysis in Proceedings of IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September 11-17, 2004), 210-219.

[24] Maletic, J. I., Collard, M. L., and Marcus, A. Source Code Files as Structured Documents in Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02) (Paris, France, June 27-29, 2002), 289-292.

[25] 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), 125-137.

[26] Mens, T. A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering, 28, 5 (May 2002), 449 - 462.

[27] Palmer, D. J., "Traceability", in Software Engineering, Dorfman, M. and Thayer, R. H., Eds., Wiley-IEEE Computer Society Press, Los Alamitos, California, 1996, pp. 266-276.

[28] Pinheiro, A. C. F. and Goguen, A. J. An Object-Oriented Tool for Tracing Requirements. IEEE Software (1996), 52-64.

[29] Raghavan, S., Rohana, R., Podgurski, A., and Augustine, V. Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September 11 - 14, 2004), 188-197.

[30] Reiss, P. S., Kennedy, M. C., Wooldridge, T., and Krishnamurthi, S. CLIMB: An Environment for Constrained Evolution in Proceedings of 25th international conference on Software engineering, ICSE (2003), 818-819.

[31] Reiss, S. P. Constraining Software Evolution in Proceedings of International Conference on Software Maintenance (ICSM'02) (Montreal, Quebec, Canada, October 03 - 06, 2002), 162-171.

[32] Scacchi, W., "Hypertext for Software Engineering", in Encyclopedia of Software Engineering, Marciniak, J., Ed. John Wiley and Sons, Inc., New York, 2002.

[33] 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, 2003).

[34] Spanoudakis, G. Plausible and adaptive requirement traceability structures in Proceedings of 14th International Conference on Software engineering and Knowledge Engineering (2002), 135-142.

[35] 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.

[36] 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), 448-455.