Concordance, Conformance, Versions, and Traceability

Ethan V. Munson
Dept. of EECS
University of Wisconsin-Milwaukee
Milwaukee, WI 53211, USA
munson@uwm.edu

Tien N. Nguyen
Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011, USA
tien@iastate.edu

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement;
D.2.9 [**Software Engineering**]: Management

General Terms

Management

Keywords

Software Traceability, Integrated Development Environment, Software Conformance, Software Configuration Management, Version Control

1. INTRODUCTION

Since its inception in 1999, the Software Concordance Project has studied problems in software traceability [5]. Our research has been based on four principles:

1. That effective traceability analysis must address the full range of documents produced by the software development process;
2. That, for the foreseeable future, a large proportion of software documents will be written in natural languages, such as informal text or various types of diagrams, and that fully automated analysis of these documents will not be tractable;
3. That traceability practices will only improve when developers have tools that substantially reduce the effort required to maintain traceability information; and
4. That structured document and hypermedia technology is a good basis for traceability support in integrated development environments.

In this position paper, we summarize the research of the Software Concordance project. The project first produced an editor for Java programs and XML documents that is unusual in supporting full hypermedia features for the complete spectrum of software documents while still providing integrated program analysis [8]. With this editor, it is possible to represent traceability relationships explicitly as hyperlinks between elements of software documents.

We became interested in how the network of links connecting a software project's documents could be analyzed in order to help developers answer questions about traceability. We named this process *conformance analysis* and developed a formalism for it [7].

An important issue that was not addressed by either the Software Concordance editor or the conformance analysis model was the problem of versioning. Software projects are constantly evolving and many traceability problems are difficult to model unless versions are taken into account. As a result, we have spent considerable effort developing a new approach to versioning and software configuration management (SCM) that is structural in nature [10].

The remainder of this article is divided into four sections. The next three sections describe our research contributions and explain what we see as important about them. Section 2 looks at the Software Concordance editor, while Section 3 introduces the idea of conformance analysis and our formal model for it. Section 4 discusses structural versioning in Molhado. Finally, Section 5 attempts to integrate these experiences and suggest what future research should follow.

2. THE SOFTWARE CONCORDANCE EDITOR

The main goal of our work on the Software Concordance Editor was to show that it is possible to build an editing environment that provides good support for the full range of software documents, while still providing the program analysis (parsing, type checking, etc.) that developers expect from a modern IDE. Prior research had either placed limitations on support for documents other than source code [4] or did not support program analysis [11].

The Software Concordance editor [8] (SCE) is a tool for browsing and editing Java program source code and XML documents. All documents are represented as trees, with Java programs being represented by an abstract syntax tree (AST) and XML documents being represented by a standard tree of elements. All documents are editable, though full-featured direct manipulation editing is not currently supported in order to reduce the project's engineering effort.

The SCE allows hyperlinks to be defined between any set of document elements, including nodes in Java ASTs. Hyperlinks may either be embedded in the documents themselves, as in HTML, or they may stored independently from the documents, as in XLink [13]. Independent hyperlinks have variable arity and do not have to be directional.

The SCE also supports a variety of media and formatting options. It is possible to place images, audio clips, and formatted text as inline documentation inside Java source code. It also supports editing of images and UML diagrams.

We believe that the SCE is the first editor to successfully integrate such a variety of software documents and to support inline hyper/multimedia documentation in source code while still supporting program analysis. In order to achieve this goal, we had to make a difficult decision to abandon using a simple text stream as the representation for program source code. The SCE represents a Java program as an AST, not as a stream of characters. This yields a structure to which hyperlinks and multimedia documentation can be bound directly, without having to use brittle comment conventions to keep this information from disrupting program analysis. We realize that many programmers will loathe the idea of having the text stream become a secondary representation for source code. We challenge those programmers to solve the same problems without following our path.

If developers have an editing environment like the SCE, they will be able to define hyperlinks between all of their software documents. Some of these hyperlinks can be specifically designed to support traceability activities. It is also possible that automated or semi-automated tools might be used to identify traceability links, using techniques like those of Marcus and Maletic [3]. The central point is that, in an environment similar to the SCE, links can be tightly integrated with software documents and can be persistent.

3. CONFORMANCE ANALYSIS

So, in such an environment, developers will have a web of software documents and traceability links that connect them. It is possible to envision an idealized moment when all of these documents are in perfect agreement, or *conformance*, with each other. At this impossible, ideal point in time, design conforms to requirements, implementation conforms to design, and so on.

But systems are rarely static. Either they evolve as products or their environment changes. This evolution will be reflected in changes to some of the software documents. After some number of changes are made, developers will decide that it is time to ensure the system's correctness. At this point, it becomes intriguing to ask whether the web of traceability links can be exploited to improve the effectiveness or efficiency of this process. We believe that it is possible to build tools that direct the attention of developers to sets of documents that may not be in agreement. We call this process *conformance analysis* [7]. The basic idea is that certain hyperlinks are relevant to traceability because they connect sets of document elements that should be in conformance with each other. In general, if none of the elements connected by such a link has changed, then those elements should still be in conformance. Otherwise, it would be useful to have a developer check them again for conformance.

We have described a formal model for conformance analysis, based on Gunter's model of the build process [1]. There are two main ideas in the conformance analysis model: abstractions and a conformance analysis process.

The first idea is the concept of an *abstraction*, which is simply a way of viewing the web of conformance links. There is always a *standard abstraction*, which is simply that view that the developers hold in their heads, but is not amenable to formal specification and automated analysis. In this abstraction, a requirement says what the system must do, a design element describes a way of doing it, and the link between them records the fact that the design resulted from the requirement. Then, a conformance analysis system will define one or more other abstractions that happen to be useful for conformance analysis.

We have studied two abstractions for conformance analysis. The *timestamp abstraction* is derived from the way that *make* use timestamps for the build process. In the timestamp abstraction, document elements and links have modification timestamps that are updated with each editing change. Links have an additional validation timestamp which records the last time when a developer looked at all elements that the link points to and determined that they were in conformance. We have proposed a heuristic rating for possible conformance problems using the timestamp abstraction and have implemented support for it in the SCE [6]. Separately, we have proposed, with Maletic and Marcus, to use Latent Semantic Analysis as another abstraction [2].

The conformance analysis process is similar to the build process. A set of document elements with no incoming link edges are identified as a *marking*. The marking is gradually expanded by checking neighboring nodes in the Web for conformance and making changes in the documents until they do indeed conform. Nguyen has presented proofs of soundness and termination for conformance analysis [6], provided all conformance links are directional. Non-directional links remain troublesome for the model.

Conformance analysis has an important difference from the build process. In the build process, developers produce only a set of starting elements of the build process (e.g. header and implementation source code files). All other products are produced automatically by tools like compilers. If the build process fails, it is considered to be because of an error in one of the starting elements. In contrast, in conformance analysis, it is likely that all elements will be produced and updated directly by developers. When a conformance problem is found, any element connected to the link may have to be changed in order to make a correction.

4. STRUCTURAL VERSIONING

The central idea of our approach to version control for software systems is the structure-oriented product versioning model in Molhado [10], the SCM system for the Software Concordance environment. In contrast to existing text file-oriented version control systems, Molhado has the ability to manage the evolution of a software system in terms of logical objects, compositions, structures, and the logical connections among them.

In Molhado, each software artifact is considered to be structured and is represented as either an attributed tree or directed graph. Each node in the tree or directed graph has a unique identifier and can be associated with one or multiple attributes in any data type to represent properties of software artifacts. The fine-grained structure versioning algorithm in Molhado was developed to provide the fine-grained version control services for artifacts. For example, any structural unit in a program source code such as classes, methods, statements, expressions, etc or any XML element in an XML document can be versioned.

In addition, having unique identifiers for nodes makes it easier to build structural differencing tools not only for source code but also for design diagrams and multimedia documentations. The structural difference tools were built to allow developers to compare two arbitrary versions of a project's directory structure, of a program, of a design diagram, and of a structural unit in a structured document. The structural differencing tools in SCE are very efficient since they do not involve tree or directed graph differencing algorithms. Changes to the structure of software artifacts are recorded via the use of structured editors in SCE.

Because Molhado uses the product versioning model, consistent configurations that correctly relate software artifacts from all phases of a software process across all models are easily maintained. Product versioning in Molhado simplifies the configuration management tasks for software developers. For example, they do not have select versions of files to be included in a version of a composite, structured object, or a project. Since configurations are maintained among software objects, rather among files, developers must not worry about the mapping from a version of a composite object to versions of files.

Based on Molhado, a versioned hypermedia model is constructed to manage the evolution of fine-grained traceability links among software artifacts and the versions of complex hypermedia structures in a software system. Hyperlinks are explicitly represented as first-class entities with variable arity and are managed separately from document content, thus facilitating systematic analysis, information retrieval, browsing, navigation, and visualization of traceability link networks.

To support composition and aggregation for linking structures, two additional concepts are introduced: *linkbase* and *hypertext network*. A linkbase is a container for hypertext networks and/or other linkbases. A hypertext network, which represents a traceability link network, can belong to only one linkbase. The relation between a linkbase and a hypertext network is the same as the relation between a directory and a file in a file system. A hypertext network contains links and anchors. A link, representing for a traceability relationship, is n-ary and connects a set of its anchors together. Anchors denote regions of interest within an artifact and form the endpoints for links. An anchor can belong to multiple links. A link or an anchor can also belong to multiple hypertext networks. An anchor does not belong to an artifact. It *refers* to a structural unit within an artifact or to an entire artifact. This separation between anchors and structural units allows for the separation between hypertext networks and document contents. This approach is called *open hyperbase* versioned hypermedia [12] where the hypertext structures are managed separately from documents. Links and anchors can be associated with any attribute-value pairs.

Molhado not only supports SCM for traceability link networks, but also provides version control for individual traceability links via its structure versioning algorithm [9]. In brief, the evolution of an individual document node, an individual traceability link, or a traceability link network is recorded over time in a fine-grained manner. This feature is very important for software traceability since a traceability link connects fragments of artifacts together. As far as we know, this is the first versioned hypermedia model that achieves this feature.

5. TOOLS FOR TRACEABILITY

Having described the three main contributions of the Software Concordance project, it now makes sense to summarize them and look to the future.

5.1 Discussion

In keeping with the research principles presented in Section 1, research on the Software Concordance project has developed a set of tools that have the potential to help developers record, understand and maintain traceability information. The Software Concordance editor permits developers encode traceability information using hypertext links. The Molhado versioning framework keeps a version history of the documents and the links between them, using novel versioning techniques that directly track structural changes to a project and to the internal structure of individual documents. Concordance analysis offers the possibility of reducing developer effort to maintain traceability in the face of an evolving project.

As with any research effort, our tools have both strengths and limitations.

We have always been surprised to see the Software Concordance editor receives a rather lukewarm reception from developers. As researchers interested in documents, we think it is natural to want source code to support features found in conventional text documents, like hyperlinks and embedded objects from other media. The truth seems to be that developers like to think of code as a separate entity with a very simple representation. While, in the abstract, developers think that hyperlinks and inline diagrams as documentation would be useful, in practice these features have little added value for them. Some of this derives from developers natural lack of interest in providing documentation and it strongly suggests the need for tools to automatically discover traceability links. Still, developers must be able to create and maintain traceability links manually and the Software Concordance editor provides the tools to do this.

The Software Concordance editor is also not yet of production quality. It does not support direct manipulation editing and certain code modification are not permitted because of the way the parser and editor interact. But these problems are not fundamental and can be solved with additional engineering effort.

We think that a successful conformance analysis system could convince developers to use a system like the Software Concordance. Currently, traceability analysis is considered to be painfully tedious. Conformance analysis holds the promise of substantially reducing the effort in maintaining good traceability, but many challenges remain.

The timestamp abstraction that we have proposed is simple to implement and will work for small projects as well as large ones. But, to date, we have not validated its usefulness with a substantial case study and our proposed heuristics are not based on empirical evidence. We hope to apply the technique to real projects to address these problems.

We would also like to see research on new abstractions for conformance analysis. We have plans to explore the use of measures derived from Latent Semantic Analysis as one example. Other measures derived from program analysis or information retrieval techniques are also intriguing. The idea of combining multiple abstractions is also very appealing, because the use of complementary abstractions could strengthen analysis substantially.

In the area of version control, two important areas of concern are collaboration support and performance. Molhado currently lacks adequate support for collaboration for two key reasons. The first reason is that, while Fluid has powerful structural differencing tools, it currently cannot merge two versions into one. Building support for version merging is one of our next research tasks. The second lack is in user interfaces. We need to investigate what kind of user interfaces will be easily understood and used by developers of large projects. Our intuition is that developers will need to be given new versioning concepts that simplify the underlying complexity of the system. We area also very interested in how to support project teams that are distributed over great distances and many time zones. We think that this will require a combination of good rules of practice and user interfaces that support those rules well.

For structural version control to be useful, it must also have acceptable performance. The theoretical basis for the Fluid persistence model has good scalability characteristics, but the current implementation has yet to be rigorously tested. Nguyen has generated initial performance results, showing that storage requirements are substantial. Memory and disk requirements grow by a factor of about six over simple, ASCII-based representations of XML or Java documents. This appears to be a linear scaling factor, which seems acceptable. In contrast, we know of some runtime bottlenecks (e.g. in writing version files) that are a real problem and will have to be resolved by creative engineering. Systematic performance analysis of the system remains to be done, especially for truly large projects.

5.2 Future Directions

The previous subsection highlighted a number of areas that call for further study. We hope to pursue these areas of research in the near future. To summarize them:

- The Software Concordance editor is already largely sufficient for future research tasks. It is not a production tool and would need considerable engineering effort to reach that standard.

- Conformance analysis using the timestamp abstraction has been implemented, but the implementation has yet to be tested on a realistic example.

- The timestamp abstraction appears useful, but it would be desirable to have other abstractions with richer semantics as complements or replacements for it.

- Version control with Molhado currently lacks collaboration support and has some known performance issues.

But looking beyond these specific concerns derived from the current state of our research, where should traceability research be going?

First, we believe that traceability researchers should seek to balance formally-based techniques with less formal tools and processes designed around developer practices. Many software documents have complex natural semantics that are highly resistant to automated analysis, such as requirements for good aesthetics in user interfaces. Researchers should accept this and use formal techniques as part of a suite of methods for improving traceability practice. Thus, we seek to have a formal foundation for conformance analysis, and at the same time, are adamant that developers must be able to manually define hypertext links for traceability.

Second, we must find good incremental paths from current practice to improved practice. In that spirit, we hope that a combination of good structural version control, integration with other software documents and the ability to explicitly represent traceability relationships will convince developers to move away from simple text stream representations of program source code. Our commitment to supporting integrated program analysis is derived directly from our desire to find an incremental path to better practice, even though program analysis support presents many difficult engineering problems.

Third, software development practice is maturing. It was not that long ago that the use of even simple version control was rare. Today, both project management and project design are becoming more rigorous, even if there are still many chaotic projects. A trend toward better traceability practices could well be on the horizon. So, it behooves researchers to attack the problem with vigor and creativity.

6. ACKNOWLEDGMENTS

Satish Gupta's work was critical to the development of the first version of the Software Concordance editor. Cheng Thao has been tireless in seeking to improve every aspect of our tools and in seeking new directions for Molhado. John Boyland's assistance in working with Fluid has been central to the success of Molhado.

7. REFERENCES

[1] Carl A. Gunter. Abstracting dependencies between software configuration items. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(1):94–131, 2000.

[2] Jonathan I. Maletic, Ethan V. Munson, Andrian Marcus, and Tien N. Nguyen. Using a hypertext model for traceability link conformance analysis. In *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), an workshop associated with the 18th IEEE International Conference on Automated Software Engineering (ASE)*, 2003.

[3] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE'03, Proceedings of the Eighth International Conference on Software Engineering*, pages 125–137, 2003.

[4] Ethan V. Munson. *Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment.* PhD thesis, University of Califonia – Berkeley, 1994.

[5] Ethan V. Munson. The Software Concordance: Bringing hypermedia to the software development process. In *SBMIDIA '99 Anais, V Simpósio Brasileiro de Sistemas Multimídia e Hipermídia, Goiânia, Brazil*, June 1999.

[6] Tien N. Nguyen. *Object-oriented Software Configuration Management.* PhD thesis, University of Wisconsin – Milwaukee, 2005.

[7] Tien N. Nguyen and Ethan V. Munson. A model for conformance analysis of software documents. In *Proceedings of the International Workshop on*

Principles of Software Evolution (IWPSE), an associated workshop of FSE'03. IEEE Computer Society Press, 2003.

[8] Tien N. Nguyen and Ethan V. Munson. The Software Concordance: A New Software Document Management Environment. In *Proceedings of the ACM SIGDOC Conference on Computer Documentation*. ACM Press, 2003.

[9] Tien N. Nguyen, Ethan V. Munson, John T. Boyland, and Cheng Thao. The Molhado Hypertext Versioning System. In *Proceedings of the Fifteenth Conference on Hypertext and Hypermedia*. ACM Press, 2004.

[10] Tien N. Nguyen, Ethan V. Munson, John T. Boyland, and Cheng Thao. An Infrastructure for Development of Object-Oriented, Multi-level Configuration Management Services. In *Proceedings of 27th International Conference on Software Engineering (ICSE 2005)*. ACM Press, 2005.

[11] Kasper Østerbye. Literate SmallTalk using hypertext. *IEEE Transaction on Software Engineering*, 21(2):138–145, Feb 1995.

[12] E. James Whitehead, Jr. *An Analysis of the Hypertext Versioning Domain*. PhD thesis, University of California – Irvine, 2000.

[13] W3C XML Linking. http://www.w3c.org/XML/Linking.