Representations, Tools, and Techniques for the Complete Integration of Software Development Documents

Ethan V. Munson

July 22, 1998

Note: This document is an abridged version of the original proposal for NSF CAREER Grant CCR-9734102.

Proposal Summary

The proposed research will identify the document representations, software tools, and user interface services needed for the complete integration of all documents produced by the software development process. This work will provide models and tools that make the logical relationships between software documents explicit, thereby allowing them to be queried and analyzed. The proposed research will also bring program source code out of the dated world of eight-bit character streams by allowing multimedia documentation to be embedded within source code documents. This research is significant because it will break down the barriers that currently exist between program source code and the natural language documents that motivate, evaluate, and explain it. These barriers hinder software development by making it difficult to determine whether related documents conform to each other and by restricting in-line documentation of program source code to comments in typewriter-style text.

To meet the aims of this proposal, the investigator will address the following research issues:

- The design of document representations that support the complete interoperability of software documents, regardless of their medium, so that any software document (including source code) can include material in any medium or any relevant fragment of any other software document.
- The design of representations for the logical relationships between software documents that help developers determine whether the contents of related documents are in conformance.
- The design of analytic, visualization and user interface tools that help developers maintain and understand the relationships between their software documents and the extent to which these documents conform to each other.

The success of the research will be evaluated through the implementation of these designs in a development environment for Java programs and all their associated documents. This environment will use World-Wide Web technology to connect the documents but will supplement the standard document and link structure with fine-grained revision control, specialized

link semantics, and tools for analyzing and visualizing the entire document set. The environment will used in its own development and will also be deployed in undergraduate and graduate software engineering courses. Opportunities for deployment in industrial settings will be actively sought.

The proposal's education plan describes a set of activities which will increase enrollment and success of students from traditionally under-represented groups and will improve the university's computer science program at both the undergraduate and graduate levels. The plan includes an innovative curriculum for the GEST Summer Program for middle-school and high-school students based on the World-Wide Web, outreach activities, and new course offerings in software engineering. It integrates recent research results into graduate coursework, as well as using the environment developed by the proposed research at both the undergraduate and graduate levels. In addition, the education plan proposes to enhance the department's graduate program with a complete curriculum on software and systems.

Career Development Plan

I. Research Plan

1 Introduction

The proposed research will identify the document representations, software tools, and user interface services needed for the complete integration of all documents produced by the software development process. This work will provide models and tools that make the logical relationships between software documents explicit, thereby allowing them to be queried and analyzed. The proposed research will also bring program source code out of the dated world of eight-bit character streams by allowing multimedia documentation to be embedded within source code documents. This research is significant because it will break down the barriers that currently exist between program source code and the natural language documents that motivate, evaluate, and explain it. These barriers hinder software development by making it difficult to determine whether related documents conform to each other and by restricting in-line documentation of program source code to comments in typewriter-style text.

The results of this research will be applicable to any software development project that uses a formal process and produces a range of software documents. The growing adoption of software quality standards (such as ISO/FDIS 9000-3) means that the importance of documents in software development will only increase. It is of critical importance that improved tools for managing these documents and the logical relationships between them be developed and adopted. Such tools will also be applicable in any environment where collections of interrelated documents must be maintained and reasoned about, such as large engineering or business projects. Software development is an obvious starting point, but the problem is of general importance.

Programming environments have made great strides in recent years, so that software developers now expect close coordination between tools for editing, compiling, and debugging source code. However, programming environments do not interoperate with the office software suites that are commonly used to produce requirements and design documents, testing and bug reports, and user manuals. Recent research is closing the gap between requirements, design, and source documents through the use of formal specifications, but it remains clear that less-formal documents will be important parts of the software development process for the foreseeable future. Thus, it is important to find ways to better integrate these documents and to help software developers understand and maintain the relationships between them.

To meet the aims of this proposal, the investigator will address the following research issues:

- The design of document representations that support the complete interoperability of software documents, regardless of their medium, so that any software document (including source code) can include material in any medium or any relevant fragment of any other software document.
- The design of representations for the logical relationships between software documents that help developers determine whether the contents of related documents are in conformance.
- The design of analytic, visualization and user interface tools that help developers maintain and understand the relationships between their software documents and the extent to which these documents conform to each other.

This research builds on the principal investigator's prior work on document presentation services [28, 29, 30] and software development environments [10]. The work on document presentation services has produced a portable style sheet system based on a simple and powerful specification language [28] and a model of media that is used to reconfigure this system for applications supporting different media [29]. The principal investigator was one of the architects of the Ensemble integrated software development environment. Ensemble supports both multimedia and program source code documents and has served as a testbed for incremental program analysis techniques [20, 39] and for document presentation services [11, 15, 25, 31]. Ensemble uses a model-view-controller architecture [30, Chapter 3] that allows an arbitrary number of views of the same document and allows non-source-code documents to include other documents, regardless of type. The proposed research will extend this work so that program source code interoperates freely with documents in any medium and the relationships between these documents can be explicitly specified, analyzed, and queried.

Evaluation of the project will be based on testing with end-user populations (primarily students), runtime performance, and technical correctness. The grant project will develop a prototype environment for developing software documents, specifically Java programs and multimedia documents represented using XML [4, 5]. This environment will be tool-based, rather than monolithic, which should ease deployment and testing and will make extensive use of World-Wide Web technology. The primary test population will be students in undergraduate and graduate software engineering courses, who will use the tools in group development projects that require good documentation practices. The tools will also be tested on themselves, as the project will use them in its own development efforts. Finally, the grant project will strive to produce tools that are robust and powerful enough to justify experimental deployment in local industry.

2 Specific Objectives

The proposed research aims to integrate the diverse set of documents produced in developing software. This broad goal requires that the research produce new document representations that improve the interoperability of software documents and make the relationships between the ideas in these documents explicit. Using these new representations, the research must also develop tools and services that make it possible to maintain and understand the relationships between the documents and insure that the contents of the documents are in conformance with each other. These research goals give rise to a number of specific objectives.

The achievement of complete interoperability of software development documents implies that any document can include fragments of any other document, no matter what types of documents are involved. Program source code is the obvious stumbling block, since editors and environments still represent programs as simple text streams. What is needed is a new representation for program source code that allows

- Inclusion of fragments of other documents.
- Active inclusions, which change when the source document changes, and
- The binding of inclusions and multimedia documentation to sections of source code without interfering with the integrated program analysis services (lexing, parsing, static semantic analysis) of the editor or environment.

Given such a representation, new program presentation tools will be required that give programmers flexible control over the way that they view their source code so that the embedded documentation doesn't distract from the development task.

Interoperability between formal and informal documents also requires a uniform model of revision control. While the value of revision control tools for source code is well understood, such tools are not generally available for informal documents, especially non-text documents.

Software development documents have many logical relationships, which may be described by cross-references in the documents' content, but just as often are only implicit. Furthermore, these documents change over time and their logical relationships can also change as a result. The proposed research will develop representations, based on earlier research on hypertext [13, 14, 18], that allow developers to make these relationships (or links) explicit. This goal gives rise to the following objectives:

- A taxonomy of the types of logical relationships between software documents.
- A representation allowing bi-directional, typed links between particular versions of software documents that is compatible with World-Wide Web technology or an enhancement thereof. This representation must be amenable to efficient querying and analysis.

Once it is possible to represent these logical relationships, the proposed research will investigate the tools and services needed to analyze, visualize, and maintain them. Some of the subtasks that this research will accomplish are:

- User interface services that ease the definition of document relationships, so that software developers do not hesitate to specify them.
- Tools which analyze collections of software documents and their relationships for possible conformance problems.
- Visualization tools that help developers understand the level of conformance in a system and locate likely problem areas.

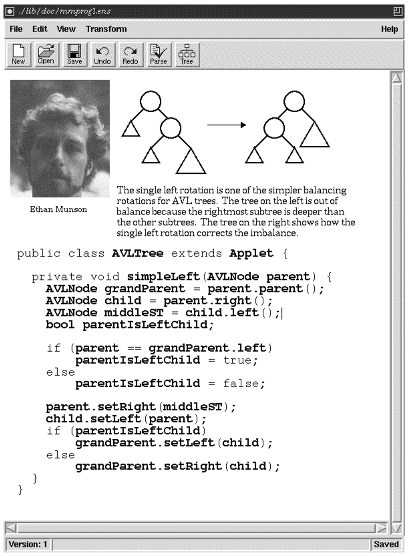

Figure 1: This screen dump of the Ensemble system illustrates how embedded multimedia documentation could be used to clarify complex algorithms, such as the rotations used to balance AVL trees. A section of Java source code is documented by a 2D graphic showing the rotation along with an explanatory paragraph. While Ensemble has in the past had support for video, this screen dump simulates an explanatory video clip with an included image.

- User interface services that aid in the correction of conflicts between documents.

The research will be evaluated by deploying its tools in undergraduate and graduate software engineering projects and by using the tools in their own development. Other deployment opportunities (in local industry, for example) will be actively sought.

3 Background

3.1 Software documents

The many documents produced by the software development process can be broadly divided into two categories: *formal* and *informal*.

Formal documents include program source code and formal specifications. Their common characteristic is that their syntactic and semantic structure can be determined by analysis of a text stream (with the obvious exception of visual programs). Formal documents are written using ASCII text editors or specialized environments. Even the advanced environments (for example, Ada-Assured [12]) are restricted to text, albeit with font and color variations, and do not support documentation in other media or connections with other software documents. The limitation to textual documentation can prevent programmers from expressing important ideas about their code that are better expressed in other media. Figure 3.1 shows an example where an embedded graphic figure clarifies an algorithm. The fact that programmers cannot link their source code to its supporting documents is just as serious a limitation, since the code is often a direct expression of ideas in those other documents.

All other software documents are informal. Any syntactic or semantic structure they have is either specified directly by the user, obtained from a shared template or form, or is implicit in the natural language content of the document. Examples include requirements documents, design documents, testing and bug reports, and user documentation. Informal documents are commonly produced using commercial office software suites, such as Microsoft Office [23]. MS Office provides extensive interoperability between the different types of documents it supports: any document can include active fragments of other documents. Furthermore, MS Office documents can import a wide variety of multimedia objects. However, these inclusions are not marked with information about their semantics.

3.2 Interoperability

In practice, formal and informal documents do not interoperate. The central problem is that, in formal documents, the text stream is used both for analysis and for presentation [37]. The lexical analysis phase of program analysis requires that the text stream adhere to the language specification, which allows only textual comments. Thus, it is not possible to embed objects composed of arbitrary byte streams (such as compressed images) inside program source code. It is possible to conceive of program editors that search for special comments pointing to pieces of non-text documentation held externally, but there are no examples of such an editor. Such an editor would require a special formatting model to correctly display the non-text documentation.

This is not to say that program presentation has been ignored. In fact, there is a large literature on program pretty-printing. Early work focused on pretty-printing standards for particular languages (see Baecker and Marcus for a summary [1, p. 18]) and on line-breaking algorithms for program statements [17, 24, 32, 33, 40]. More recent work has emphasized specification languages for pretty-printing, such as PPML [16]. All of this work has focused entirely on program source code.

The PI's dissertation research [30] investigated a more general approach to presentation. This research showed there exists a core set of presentation services which can be shared by independent modules for different media within a larger system. This work produced a portable style sheet system, Proteus [11, 28] that can be configured to support different media (text, graphics, video) and is also suitable for program source code. Configuration of Proteus for a new medium (or another application) is performed by writing a specification of the medium's primitive types, dimensions, and presentation attributes [29]. Proteus is also designed to support multiple simultaneous views of the same document in different styles, a mechanism that can be exploited for the production of novel user interfaces without requiring separate formatting services for each view. Proteus is portable and has been used by two systems: the Ensemble software development and multimedia document environment [10] and Multiple Presentation Mosaic [19], a multiple-view browser for the World-Wide Web.[1]

3.3 Document Relationships and Conformance

There are many types of relationships between software development documents. Without claiming to present a complete taxonomy, these are some examples:

[1]Examples of the output of Multiple Presentation Mosaic can be seen on the principle investigator's Web page at `http://www.cs.uwm.edu/faculty/munson`. These demonstrate some of the uses of multiple-view technology.

- The requirements *motivate* the design.
- The design *requires* the implementation.
- A test report *evaluates* the implementation.
- A bug report *complains about* a mismatch between the requirements and the implementation.
- A change to the implementation *responds to* a bug report.
- The user manual *documents* the design and implementation.

In general, these relationships are persistent, lasting days, weeks, or years, but they are not necessarily permanent. Because the documents in a system are dynamic and can be created, altered, and removed, the set of active relationships in a system is also likely to change over time.

Let us consider an imaginary software system whose documents are in perfect harmony with each other. We might say that its documents are *conformant*, because they conform to each other. If we then alter a requirement, such as the number of users to be supported, but make no other change, it becomes possible that the system does not meet its requirements. We might then say that the system's documents are *non-conformant*, because the system's design does not conform to its requirements.

Barring major advances in natural language processing research, completely automatic testing for conformance between software documents will not be possible. However, if the relationships between software documents were explicitly recorded, it might be possible to automate *detection* of *possible* non-conformance. Such automated detection could be used to guide developers to potential problems.

It is important to note that similar relationships exist among source code and specification documents. Programming languages have commands like "include" or "require" that describe a dependency between pairs of files. The difference is that these relationships between formal documents can be found, without any ambiguity, by automated analyses. In fact, relationships like these are an important information source for re-engineering tools [26].

Each of the above document relationships carries with it an implied logical ordering of its documents. For example, testing and bug reports cannot be produced until an implementation is available, and while it is not necessarily the case that requirement documents will be written before designs, there is certainly a logical relationship between them that makes design depend on requirements. Ordered relationships like these have been used for many years to automate efficient compilation [6, 8]. However, these techniques have yet to be applied to informal software documents.

4 Methodology

4.1 Interoperability

The proposed research will develop representations that provide for interoperability among formal and informal documents. The proposed representations will:

- Allow any document to include any fragment of any other document. The included fragment may be from a particular version or may be active, changing as the source document changes.
- Allow the binding of bi-directional, typed relationships to any pair of document fragments.
- Support a uniform model of fine-grained revision control.
- Allow program analysis and compilation services to operate without major modification.

The approach taken by the proposed research will begin with a uniform tree-structured representation for informal documents, such as XML [4, 5], an emerging standard for World-Wide Web documents. Interoperability between tree-structured informal documents is well-understood and can be found in Ensemble [22] and the Grif/Thot system [34].

Then, as proposed in the investigator's earlier work on interoperability [27], a representation that intermixes sections of source code with other sections that contain informal material will be designed. The informal material may either be embedded documentation or may be an inclusion of part of another relevant document. This representation will maintain a clear separation between the source code and informal sections of the document so that program analysis need only traverse the source code sections. The representation will allow the persistent binding of the informal material to the relevant section of the source code, so that the connection between them continues from one editing session to another. Furthermore, it will provide for the definition of persistent selections [14] that will serve as end-points for document relationships.

The proposed representation for source code documents will require the development of a novel formatting model that properly integrates the formal and informal material. The central problem is that automatic pretty-printers operate not from lines of text, but rather from an abstract syntax tree [1, 16]. But in the proposed representation, the leaves of the abstract syntax tree will be intermixed with some other tree-based representation for the informal material. No existing formatter or editor must coordinate between two tree representations, so a new formatting model will be required.

The proposed research will also design a uniform, fine-grained revision control model for software documents, because revision control is a critical element of the software document environments envisioned by the investigator. Figure 2 illustrates how apparent cycles in the relationship graph of software documents are broken when the relationships are defined between versions of the documents.

This work will build on research by Magnusson, Asklund, and Minör [21] on version trees for tree-structured text documents. They showed that version trees are more flexible and semantically correct than the line-oriented revision systems widely used for program source code [35, 36] and are better suited for collaborative software development. Wagner extended this work to integrate version management with program editing and analysis operations in the Ensemble environment [38], but did not provide a persistent representation of the versions. The proposed research will apply version trees to multimedia documents and will provide the persistence missing from Wagner's work.

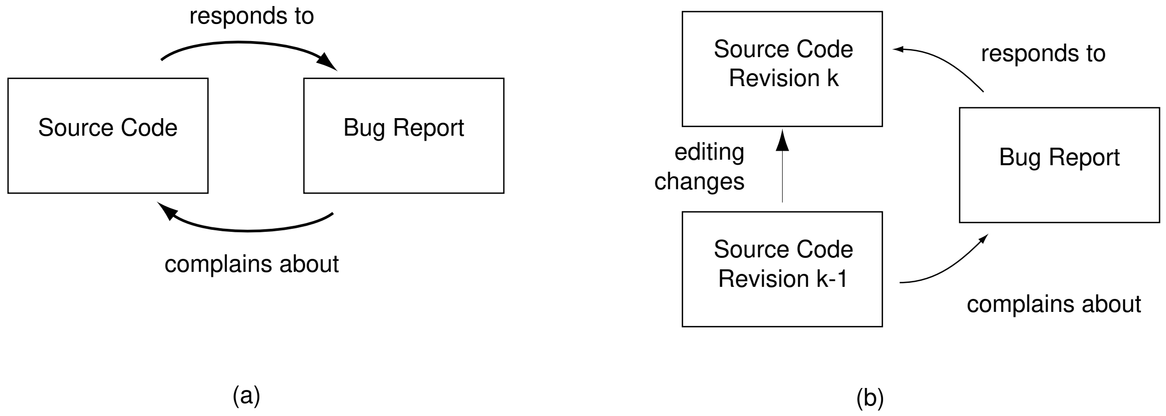

Figure 2: In (a), the relationships between a source code document and a bug report form a cycle, because the code is both cause of the bug report and the response to it. However, the addition of version information in (b) breaks this cycle by making clear that the bug report is complaining about a problem in the version $k-1$ of the source code, while version k of the source code has been edited and responds to the bug report.

4.2 Software Document Relationships

Section 3.3 listed several types of relationships that may exist between software documents. The list is not claimed to be complete, exhaustive, or minimal; it is simply a set of examples.

The proposed research will study relationships between software documents in order to define a taxonomy for them. The types of relationships identified by the taxonomy will then be used to mark links between software documents.

It might be argued that a taxonomy of relationships is unnecessary, that it would be sufficient to simply mark the existence of dependencies between documents without any information about the type of relationship. This argument is misguided because relationship types can convey important information about the semantics of the connection between two documents. Suppose that *complains about* and *comments on* are two relationship types. The fact that *A comments on B* is essentially neutral. In contrast, when *C complains about D*, it is clear that a problem exists. In most systems, complaints require a response, while comments do not.

Using the taxonomy of document relationships, the proposed research will design a representation for links between documents that makes these relationships explicit. This representation will have the following characteristics:

- Links will connect parts of document, rather than entire documents, so that relationships can be defined on a fine-grained basis.
- Since changes to document elements alter their relationships, links will connect specific versions of document elements. This will allow the network of relationships to reflect the dynamic nature of software documents.
- Since a developer may want to follow a link in either direction, the link representation will support traversal in both directions.

- At the same time, the link representation will be compatible with the World-Wide Web convention of embedding links in the source document (which normally makes them uni-directional).
- The representation must allow developers to state whether or not the documents in the relationship are conformant. The simple fact that requirements have changed does not *necessarily* mean that the design is inadequate. It simply means that a developer needs to inspect both documents to determine whether there is a problem.
- The representation must be readily accessible for querying and analysis.

4.3 Analysis, Visualization, and User Interface Services

Once the document and relationship representations have been defined, the proposed research will design and implement services that help developers maintain and understand the relationships between their software development documents. This work will rely on and enhance the experimental testbed described in the next section.

The proposed user interface services will allow the creation, inspection, and removal of links between documents. Developers will also be able to mark links with information about conformance of the documents that each link connects. Other interface services will allow the user to construct queries for types of relationships and levels of conformance. For example, a user may want to find all pairs of non-conforming documents having the *requires* relationship. The responses to these queries can either be reported to the user via a textual interface or by highlighting matching document sections.

In contrast to the user interface services, which will primarily provide information about local relationships, the analysis services will compute properties of the graph of relationships as a whole. Because this information may be complex and hard to understand, visualization tools will be designed and developed, building build on prior research on graphical presentations of large software systems using graphs [26] and compact visual summaries of source code files [2, 7].

The analysis and visualization services will allow developers to answer questions like the following:

- If requirement R is changed, which other documents may require changes?
- When requirement R changed, what changes to other documents had to be made?
- What features of the design have been complained about three or more times?
- How often are design and requirements changes made in response to bug reports?
- How often do bug reports complain about documents that test reports also complained about?

4.4 Evaluation

The concepts explored by the proposed research will be evaluated by implementation in an experimental testbed: an environment for Java programs and their associated informal documents. The environment will use a tool-based design, rather than trying to create a monolithic program, but will include a fairly large editor supporting all types of documents. Furthermore, the environment will be implemented in Java itself, so that over time, the researchers will be able to use and evaluate the tools they are creating.

The Java language will be used because it has a clean syntactic design that eases program analysis and because of its importance to the WWW. Focusing on a single programming language will simplify the system by allowing the use of special-purpose code to handle any tricky problems with incremental program analysis.

Informal documents will be represented using XML [5], an evolving standard for WWW documents. Unlike the HTML standard [3], which defines a single type of document, XML provides a mechanism for defining new types of documents. Furthermore, XML's design is intended to support bi-directional links, but this part of its design is not yet complete [4]. Unlike SGML [9], on which it is based, XML is designed for use in interactive systems.

The core of the environment will be an editor for Java programs and XML documents. The editor's features will include:

- Full interoperability between all document types,
- Integrated parsing and type-checking of Java programs,
- User interface features for managing document relationships,
- World-Wide Web compatibility, and
- Integration with the environment's revision control system.

All other tools and services, including revision control, relationship analysis, and relationship visualization, will be implemented as independent programs.

The experimental testbed will be used for group projects in undergraduate and graduate software engineering classes. Students using the tools will be surveyed about the tools usefulness and usability. More detailed evaluation information will be obtained through debriefings of selected students in these courses and through usability studies with small groups of users. Deployment of the experimental testbed in industrial settings is also planned and will use similar evaluation methods.

Other sources of evaluative information will be:

- Systematic use of mock-up situations to evaluate the design of document and link representations and the taxonomy of document relationships.
- The use of test suites to evaluate the correctness of the environment's tools.
- Performance testing of tools.

References

[1] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs.* Addison-Wesley, Reading, Massachusetts, 1990.

[2] Thomas Ball and Stephen G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, April 1996.

[3] T. Berners-Lee and D. Connolly. *Hypertext Markup Language — 2.0.* World Wide Web Consortium and MIT, June 1995. Internet Draft available from www.w3c.org.

[4] Tim Bray and Steve DeRose. Extensible markup language (xml): Part 1. linking. Available on the World Wide Web at http://www.w3.org/TR/WD-xml-link, April 1997.

[5] Tim Bray and C. M. Sperberg-McQueen. Extensible markup language (xml): Part 1. syntax. Available on the World Wide Web at http://www.w3.org/TR/WD-xml-lang, June 1997.

[6] Geoffrey Clemm and Leon Osterweil. A mechanism for environment integration. *ACM Transactions of Programming Languages and Systems*, 12(1):1–25, January 1990.

[7] Stephen G. Eick, Michael C. Nelson, and Jeffery D. Schmidt. Graphical analysis of computer log files. *CACM*, 37(12):50–56, December 1994.

[8] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9:255–265, 1979.

[9] Charles F. Goldfarb, editor. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML).* International Organization for Standardization, Geneva, Switzerland, 1986. International Standard ISO 8879.

[10] Susan L. Graham. Language and document support in software development environments. In *Proceedings of the Darpa '92 Software Technology Conference*, Los Angeles, April 1992.

[11] Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *Proceedings of the ACM SIGSOFT Fifth Symposium on Software Development Environments*, pages 130–138, Tyson's Corner, VA, December 1992. ACM Press.

[12] Inc. Grammatech. Ada-assured home page. Accessible at http://www.grammatech.com, 1997.

[13] Bernard J. Haan, Paul Kahn, Victor A. Riley, James H. Combs, and Norman K. Meyrowitz. IRIS hypermedia services. *CACM*, 35(1):36–51, January 1992.

[14] Frank Halasz and Mayer Schwartz. The dexter hypertext reference model. *CACM*, 37(2):30–39, February 1994.

[15] Michael A. Harrison and Vance Maverick. Presentation by tree transformation. In *IEEE COMPCON '97*, February 1997.

[16] INRIA: Centaur Project, Sophia-Antipolis, France. *The PPML Manual*, February 1994. For Version 1.3 of Centaur. Available by ftp from babar.inria.fr in directory `pub/croap/bertot`.

[17] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software—Practice & Experience*, 11(11):1119–1184, November 1982.

[18] John J. Leggett and John L Schnase. Viewing dexter with open eyes. *CACM*, 37(2):76–86, February 1994.

[19] Hong Liu. Multiple Presentation Mosaic. Master's thesis, University of Wisconsin-Milwaukee, May 1996.

[20] William Harry Maddox, III. *Incremental Static Semantic Analysis.* PhD thesis, University of California, Berkeley, January 1997.

[21] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-grained revision control for collaborative software development. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 33–41. ACM Press, December 1993.

[22] Vance Maverick. *Presentation by Tree Transformation.* PhD thesis, University of California, Berkeley, 1997.

[23] Microsoft, Inc., Redmond, Washington, USA. *Microsoft Office 4.2*, 1995.

[24] Martin Mikelsons. Prettyprinting in an interactive programming environment. Technical Report RC 8756, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, March 1981.

[25] Alok Mittal. SHILPĒ: A presentation system for ensemble. Master's thesis, University of California, Berkeley, California, December 1995.

[26] H. A. Müller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *Proceedings of the ACM SIGSOFT Fifth Symposium on Software Development Environments*, pages 88–98, Tyson's Corner, VA, December 1992. ACM Press.

[27] Ethan V. Munson. Interoperability of software documents. In *Workshop on Software Engineering and Human-Computer Interaction: Joint Research Issues*, pages 153–162, May 1994. Workshop pre-prints.

[28] Ethan V. Munson. A new presentation language for structured documents. *Electronic Publishing: Origination, Dissemination, and Design*, 8:125–138, September 1995. Originally presented at EP96, the Sixth International Conference on Electronic Publishing, Document Manipulation, and Typography, Palo Alto, CA, September 1996.

[29] Ethan V. Munson. Toward an operational theory of media. In *Proceedings of the Third International Workshop on Principles of Document Processing.* Springer-Verlag, Palo Alto, CA, September 1996. To be published as part of the Lecture Notes in Computer Science series.

[30] Ethan Vincent Munson. *Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment.* PhD dissertation, University of California, Berkeley, December 1994. Also available as UC Berkeley Computer Science Technical Report UCB/CSD-94-833.

[31] Kannan Muthukkaruppan. SPINE, a synthesizer for practical incremental evaluators. Master's thesis, University of California, Berkeley, CA, May 1994.

[32] Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.

[33] William W. Pugh and Steven J. Sinofsky. A new language-independent prettyprinting algorithm. Technical Report TR 87-808, Dept. of Computer Science, Cornell University, Ithaca, NY, January 1987.

[34] Vincent Quint and Irène Vatton. Combining hypertext and structure documents in grif. In D. Lucarella, editor, *Proceedings of ECHT '92*, pages 23–32, Milan, December 1992. ACM Press.

[35] M. J. Roekind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[36] W. F. Tichy. RCS — a system for revision control. *Software: Practice and Experience*, 15(7):637, July 1985.

[37] Michael L. Van De Vanter, Susan L. Graham, and Robert A. Ballance. Coherent user interfaces for language-based editing systems. *International Journal of Man-Machine Studies*, 37:431–466, 1992.

[38] Tim A. Wagner and Susan L. Graham. Integrating incremental analysis with version management. In *Proceedings of the Fifth European Software Engineering Conference*, 1995.

[39] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 31–43, June 1997.

[40] Richard C. Waters. XP: A Common Lisp pretty printing system. A.I. Memo 1102a, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, August 1989. Also appears in edited form as Chapter 27 of *Common Lisp: The Language, second ed.*