The Software Concordance: A new Software Document Management Environment[*]

Tien N. Nguyen
Dept. of EECS
Univ. of Wisconsin, Milwaukee
tien@cs.uwm.edu

Ethan V. Munson
Dept. of EECS
Univ. of Wisconsin, Milwaukee
munson@cs.uwm.edu

ABSTRACT

Because source code is only one of many types of documents that must be maintained as a software system evolves, modern software documentation could be improved by better interoperability between source code and other non-program software documents. The Software Concordance (SC) is a prototype integrated development environment (IDE) that uses a tree-based document representation for software documents, an integration between hypermedia and program analysis services, and inline multimedia documentation in program source code to improve software document management. It provides uniform editing, versioning, and hyperlinking services for Java source code, and multimedia documentation in XML. Unique features of the Software Concordance include support for hyperlinks and multimedia documentation in program source code that do not disrupt lexical and syntactic analysis and a single versioning model for both hyperlinks and the documents they connect.

This paper describes the XML-compatible, tree-based document representation of the Software Concordance environment. This paper also describes essential features of the Software Concordance environment that provide supports for developers and documenters in a software project.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments;
H.5.4 [**Information Interfaces and Presentation**]: Hypertext / Hypermedia

General Terms

Documentation, Management

Keywords

documentation, hypermedia, software engineering

[*]This research was supported by the University of Wisconsin-Milwaukee Dissertation Fellowship, U. S. Department of Defense and by NSF CAREER award CCR-9734102.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGDOC'03, October 12–15, 2003, San Francisco, California, USA.
Copyright 2003 ACM 1-58113-696-X/03/0010 ...$5.00.

1. INTRODUCTION

During the process of implementing and designing large-scale software systems, software engineers produce a large variety of heterogeneous software artifacts such as requirements documents, feasibility studies, design specifications, source code, module documentation, test plans, bug reports, and sales plans. One of the most challenging tasks is to manage this collection of software documents so that the information can be quickly accessed and consistently organized. However, most of existing integrated development environments (IDEs) do not interoperate well with the office software suites that are commonly used to produce documentation. In particular, formal documents including program source code and formal specifications are written and managed as simple text documents using specialized text editors, often in IDEs that include interactive program analysis. Non-program documents and multimedia documentation are produced using a variety of tools such as word processors, graphics editors, and specialized environments for project planning or graphical design languages such as UML [30]. In general, these tools do not interoperate well with the environments used to edit formal documents.

A typical scenario in the maintenance phase is that of performing a test case. A bug in a particular functionality is found and a bug report is created. To fix the bug, an engineer reads the bug report, test case and requirement specification. The engineer then finds the corresponding design document and source code, and after analyzing them, modifies, recompiles and tests the system. This process has inefficiencies because software documents and source code are maintained separately using different tools that interoperate poorly. The source code is maintained using text editors or IDEs that have interactive program analysis. Other software documents, unlike source code, do not have well-defined semantics, and are maintained using word processors and graphics editors. The ability to define hyperlinks: "bug report $\rightarrow$ test case $\leftrightarrow$ requirement specification $\leftrightarrow$ design document $\leftrightarrow$ source code", and navigate this rich network of documents would greatly improve the efficiency of this process. The last hyperlink should invoke an IDE and take the user to the class or the method that has the error. The hyperlinks across program source code and documentation are hardest to achieve because IDEs and word processors used to maintain documentation do not interoperate well.

The SC research attempts to achieve the interoperability between program source code and documentation by breaking down the barriers that currently exist between program source code and the natural language documents that motivate, evaluate or 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 SC project is also motivated by the observation that software engineers have difficulty maintaining semantic consistency among their documents. The relationships among software documents are often implicit and thus cannot be browsed, navigated, queried, or analyzed systematically. This is particularly true for formal documents, whose lexical and syntactic rules hinder the use of hyperlinks, the most natural means for representing relationships.

To address these problems, we seek to demonstrate that it is possible to create a new kind of IDE in which:

- All software documents, including source code, are XML-compatible and support hyperlinks and embedded multimedia elements as documentation. Some software documents (such as source code) will be suitable for specialized analyses (such as compilation), but these analyses will not hinder interoperability.

- A variety of document views will be available in order to support the variety of tasks that a programmer performs. For example, programmers may want to hide distracting multimedia documentation or may benefit from novel "fish-eye" presentations of source code.

- The environment will provide analyses and visualizations of the graph of hyperlinks among the software documents so that developers can answer important questions related to such issues as requirements and bug tracing.

- Programmers will continue to have high-quality, interactive program analysis built into their editors and will not be forced to use unnatural top-down editing techniques.

The Software Concordance (SC) is a prototype environment that uses a uniform, tree-based, XML-compatible document representation for both Java source code and multimedia documentation in XML (including UML diagrams). This document representation supports fine-grained (element-level) version control and fine-grained linking between document elements. Users can edit Java, XML documents (raw multimedia data is stored separately and not versioned), UML diagrams, and import from and export to standard XML format. They can attach multimedia documentation and hyperlinks to any part of a Java program and include multimedia material in other types of documents. The SC editor also supports synchronized multiple presentations of a document using style sheets, so that documents can be viewed in multiple ways, depending on the task at hand.

The next section discusses the related work on improving the interoperability among software documents. Section 3 describes our representation for software documents that allows for an integration between hypermedia and program analysis services, inline multimedia documentation within program source code, linking between any fragment of any software documents, and versioning for software documents and their relationships in a fine-grained manner. Section 4 describes how the SC environment provides a uniform fine-grained version control supports for documents and hypertext structures of documents. The essential features of the SC environment are discussed in section 5. Discussions, future work and conclusions appear in the final section.

2. RELATED WORK

Documentation has long played a key role in aiding program understanding. Software engineers rely on program documentation as an aid in understanding the functional nature, high-level design, and implementation details of complex applications. There are various commercial software documenting systems and tools, which vary from office software suites [1, 20] to specialized documenting environments [28, 30]. Although they have been very successful in providing excellent documenting tools, most of them are not focused on providing a complete software document management environment, where program source code and multimedia documentation interoperate well with each other in a cohesive way. Related research work on improving interoperability between program source code and documentation in software development environments can be divided into following categories: software document management environments, integrated programming environments, hypertext tools, versioned hypermedia tools, and program browser and visualizer.

Software document management environments maintain the relationships among documents from different phases of a software system's life cycle. Two environments have approaches that are strikingly different from ours. Neither system uses hyperlinks to represent relationships and their architectures are also quite different. Knuth's literate programming [17] envisioned a single document encompassing both the implementation and documentation of a system and used techniques from book typography to describe relationships within this document. In contrast, we picture a large collection of documents, linked into a literate whole by versioned hypermedia. The Desert system [29] improves software document interoperability by integrating a variety of tools through broadcast messages. Its FrameMaker-based editor integrates program editing with other editing tools for design diagrams and user interfaces. Desert lacks support for the evolution of document relationships and its tool-based approach tends to produce systems with less integration than we seek. The SODOS and DIF systems were closer to our approach. As in our approach, the Software Documentation Support (SODOS) system [15] was based on a uniform document graph model and used a relational database to store a pre-defined set of software document and relationship types. The Document Integration Facility (DIF) [11] supported many types of documents produced and used throughout the development process. DIF represented a hypertext network by storing all textual information in files and storing relationships in a relational database. DIF was designed to support traceability through keyword-based search and navigation mechanisms. SODOS or DIF both stored only a single version of a software project and so could not represent its evolution. They also lacked support program analysis or multimedia.

Programming environments are systems that integrate various program analysis such as lexing, parsing, compiling, and code generating into a program editing environment. This class of environments does not focus on supporting non-program documentation. Most of the editing services are text-based and do not support fine-grained hyperlinks. More advanced integrated programming environments include the Cornell Program Synthesizer(CPS) [33], Ensemble [37], Pan [5], and Mjølner [16]. CPS was one of the first programming environments and introduced the syntax-directed editing approach. The CPS and its commercial derivatives are text-based. Ensemble, and its predecessor Pan had a vision similar to that of the Software Concordance. Both systems sought to provide novel presentation and user interface features in order to support a broader array of software development tasks than just writing code. Ensemble added support for multimedia documentation and style sheets, but lacked a uniform document representation encompassing source code. Neither Ensemble or Pan could support fine-grained hypermedia links because their storage format for programs was plain ASCII source code. Mjølner is a very comprehensive, grammar-driven programming environment that supports program editing through both textual and window hierarchy views.

Several researchers have applied hypertext technology to improve software document management. DynamicDesign [6] is a hypertext Computer-Aided Software Engineering Environment for the C language. It defines a fixed set of information units and relationships. Computed Hypermedia Programming (ChyPro) [2] is a hypermedia-programming environment for SmallTalk-80. ChyPro also uses hypermedia techniques to improve a programming environment, which allows interleaving graphical and textual representations of SmallTalk code, documentation and data, and linking heterogeneous or distance pieces of code or documentation together. ChyPro had a serious scalability problem because both data and hypermedia information were stored in a centralized repository. Østerbye supports literate programming [26] by modeling a programming environment for SmallTalk as a hypertext system. His environment allows programmers to write code and documentation, and then link them together freely. The Chimera open hypermedia system [3] is designed for software development. Chimera addresses the interoperability problem by providing hypermedia services across multiple documents maintained by different applications. None of these hypertext-based approaches addresses the issue of document evolution and only Chimera's paradigm is compatible with interactive program analysis.

Versioned hypermedia systems [12, 13, 14, 39] offer an appealing approach to representing the evolution of software documents and their relationships. These systems keep track of the evolution of the objects connected via hypertext links and some also maintain versions of the links. However to date, IDEs based on versioned hypermedia have only provided simple versioning of objects, often with limitations, and never with versioning of links. Also, none support interactive program analysis. In both HyperWeb [10] and HyperCASE [8], the smallest versionable object was a file, which is too large to be a suitable basis for representing logical relationships among software documents. HyperWeb was based on RCS [36], while HyperCASE used a specialized database. HyperPro [27] was more fully developed and supported versioning of objects as small as procedures in Pascal programs.

Finally, program browsers and visualizers exploit program analyses to provide informative views that help developers understand their systems. These systems rely on batch analyses to generate their representations, providing interactive browsing but do not support interactive editing. CHIME [9] and Javadoc [32] insert HTML tags into source code by querying a database produced by existing static analysis tools. SHriMP [31] focuses on visualizing program source and related structured documents. Reverse engineering is a branch of software engineering that focuses on recreating high-level information (such as program documentation) from low-level artifacts (such as source code). Many research explored automated approaches to maintaining the connection between documentation and code by using the reverse engineering capabilities, information retrieval and latency semantic indexing techniques [18, 19].

The systems described here have valuable features, but none of them provide the kind of complete solution that we envision. Software document management environments have evolved from the rigid structures of SODOS to the flexible, hypermedia-based approaches in HyperPro, but they never support program analysis. Desert recognizes that source code is simply a special kind of document, but does not take the step of adopting a uniform document representation. Hypermedia-based IDEs lack support for the evolutionary aspects of software documents. The Software Concordance brings these approaches together to provide interoperability between program source code and documentation via unified editing and hypermedia services, flexible presentation, version control, and integrated program analysis in a single system.

3. UNIFORM DOCUMENT REPRESENTATION

In a collection of software documents produced during the software life cycle, non-program documentation such as requirement specifications, design documents, test cases, bug reports, user manuals, and transition procedures have some explicit or implicit structures. They can all be represented by structured documents [22]. In a broader sense, program source code is also a kind of structured document that has a tree structure of lexical units. Similarly, other formal specifications are also structured documents. Structured documents can be divided into two categories [22]: *formal* and *informal*. Formal documents include program source code and formal specifications. Informal documents include all other documents. A formal document is written using a formal language, and formal languages have precisely defined semantics. Therefore, formal documents can be understood and analyzed by tools such as compilers. An informal document is written using a natural language, and natural languages do not have precisely defined semantics. A structured document can be seen as a hierarchical structure of structural units. A *structural unit* is a fragment of a software document that is used to encode a concept such as a sentence, a paragraph, a section in software documentation or a statement, declaration, class in program source code, or a graphical item in a UML diagram.

To address the interoperability problem, the SC research developed a uniform document representation that encompasses both formal documents (Java programs) and informal documents (multimedia documentation in XML format). The uniform document representation is built based on the Fluid Internal Representation (Fluid IR) [7]. The Fluid infrastructure also includes a fine-grained version control engine to all system objects. The following sections describe the basic concepts of the Fluid IR model and how they are used to represent the Java program source code and multimedia documentation.

3.1 The Fluid IR and versioning model

The Fluid IR is based on two important notions: *nodes* and *slots*. A node is the basic unit of identity and is used to represent objects. A slot is a location that can store a value, possibly a reference to an IR node. A slot can exist in isolation but more typically slots are attached to nodes (using an *attribute*). An attribute is a mapping from nodes to slots. An attribute may have particular slots for some nodes and map all other nodes to a default slot. All the slots of an attribute hold values of the same data type. The Fluid IR data model can thus be regarded as an attribute table whose rows correspond to IR nodes and columns correspond to attributes. The cells of the table are slots. Once we add versioning, the table gets a third dimension: the version. Directed graphs are built from these primitives. A directed graph is defined via an attribute that holds the sequence of children for each node.

In Fluid, a *version* is a point in a tree-structured discrete time abstraction, rather than being a particular state of a system component. This is a form of *product versioning*, in which a uniform global version space is maintained across the entire the Fluid IR world. The version model is state-based, where each version can be associated with a name and meta-information. Versions and variants are not distinguished. The set of versions is arranged in a tree, called the *version tree*, with the root of the tree being the initial version of the Fluid IR world. The *current version* is the version designating the current state of the Fluid IR world. Any version may be made current and every time a versioned slot is assigned a (different) value, a new version of the system is produced, derived from and branching off of the current version.

Three additional Fluid IR entities are used extensively in our uniform document representation: *operators*, *trees*, and *syntax trees*. A tree is created by registering slots with unique names that store the *parent* and *child sequence* for an IR node. A syntax tree is a tree with an additional slot that stores an *operator* in each IR node. An operator identifies the type of its IR node and the set of operators defines the type system for a syntax tree. Thus, the operator determines the number and types of the children of an IR node.

This framework for representing typed, attributed trees and their versions provides the low level infrastructure on which we build our document and versioning services.

3.2 Document representation

The Fluid system is designed to support research on Java program analyses [7] and thus provides a suitable representation for Java program source code. Fluid's Java package defines a set of operators to represent the Java abstract syntax tree (AST), with each node in the AST being represented as an IR node. In addition to the parent and children slots required to define a tree, each AST node is associated with a slot whose value is a Java operator that defines the syntax for the structural unit represented by the AST node. For example, an addition expression "x + y" would be represented by an IR node that is associated with a Java operator slot named "AddExpression". Children nodes are left child and right child of the addition expression. They are both associated with the same operator named "NamedExpression," and each of them has a special slot whose value refers to its own identifier. The left child refers to the identifier "x", and the right child refers to the identifier "y".

XML documents are represented by syntax trees whose IR nodes have operators drawn from two new categories: *intermediate* and *text*. There is only one type of text operator, which is used to represent XML's character data (CDATA) construct. All IR nodes associated with the text operator have an additional slot that holds the CDATA string. XML elements are represented by IR nodes that have intermediate operators. For each type of element defined in an XML Document Type Definition (DTD), there will be a corresponding type of operator. IR nodes associated with an intermediate operator will have one slot that holds the element name and one additional slot for each attribute that can be defined for that operator. The current SC XML parser is non-validating, but this representation has been designed to support validating parsers as well. For example, after adding a design document DTD into the SC environment, users can operate with the design documents adhering to that DTD. In addition, a XML-based representation for UML diagrams [34] is also adapted into the SC environment to provide editing, hyperlinking, and versioning supports for UML diagrams.

3.3 Inline documentation representation

The SC environment enables any multimedia documentation, including images, audios, and graphics, to be embedded into program source code, allowing this documentation to be bound to any program structural unit. For example, a graphical picture can be edited and attached to a method in a program to illustrate the main algorithm of the method. It can also be put in a separate XML document and be referred to by a hyperlink. This feature is important since it provides the ability of self-documentation of program source code. Multimedia documentation in Java programs is supported by special slots for textual documentation, images, graphics and audios attached to IR nodes representing to Java AST nodes. The textual documentation slot holds the text string, while image, graphics and audio slots hold the names of image, graphics and audio files respectively. The multimedia files are stored separately. Traditional comments are supported in terms of textual documentation.

3.4 Hyperlink representation

The current SC system supports HTML-style hyperlinks via *href* and *anchor-id* slots. An anchor can be placed on any IR node by defining a value for the *anchor-id* slot. This identifier must be unique within the document. Links are created by defining a value for an *href* slot that is a URL. With this mechanism, hyperlinks are handled uniformly in source code, XML documents, and UML diagrams, which will allow developers to move seamlessly between program source code and its supporting documentation. Hyperlinks can be attached to and can point to any structural unit of any software document. This will permit developers to define very fine-grained links among the various types of software documents.

Importantly, this approach to binding multimedia annotations and hyperlinks to source code does not interfere with Fluid's services for program lexing, parsing, and type checking. This is possible because Fluid's analysis system ignores the slots that support multimedia and hyperlinks. So, programmers using the SC gain hypermedia services without sacrificing the integrated program analysis typical for modern development environments.

3.5 Storage representation

Java programs and XML documentation are stored according to the Fluid persistence model. In the Fluid persistence model, a *persistent entity* is the basic unit of information that can be stored and then loaded. The Fluid persistent storage model uses the *forward direct delta* technique to store differences between versions. The data files for versioned persistence are immutable and may be freely duplicated or distributed using any file sharing protocol.

The SC system provides facilities to import and export its internal documents (stored in the Fluid persistence representation) from and to external formats. There are several types of external formats that are currently supported such as plain Java source code, plain ASCII text file, multimedia documentation in XML, a UML diagram in XML format, and an XML-based Java representation similar to JavaML [4]. The last format must be used if embedded multimedia documentation is to be maintained. Multimedia files are stored separately and are referenced. Via the SC editor, documents from any version of the system can be exported to any of these types of files. This export and import feature is obviously important since it helps users work with tools outside the SC system.

4. VERSION CONTROL SUPPORT

This section describes how the low-level version infrastructure of Fluid is used to support fine-grained versioned hypermedia in the Software Concordance.

4.1 Fine-grained versioning for documents

A software configuration management system (SCM) [23] was developed to provide fine-grained version control and change management supports for software documents. Software Concordance documents (including program source code, multimedia documentation and UML diagrams in XML) are treated as *tree-structured components* in our SCM system and are versioned in a fine-grained manner. The details of the tree-based versioning scheme are illustrated in Figure 1. For example, document *A* has three versions: *a*, *b*, and *c*. Versions *b* and *c* are derived from version *a*, as shown in the version history part of Figure 1 a). The node structure of the document tree for the three versions is shown in Figure 1 a). Version *a* has five nodes numbered from 1 to 5. Version *b* has two differences from the version *a*: node 4 was deleted and the content of node 5 was changed. Version *c* has an inserted node (node 6) and node 3 was deleted.

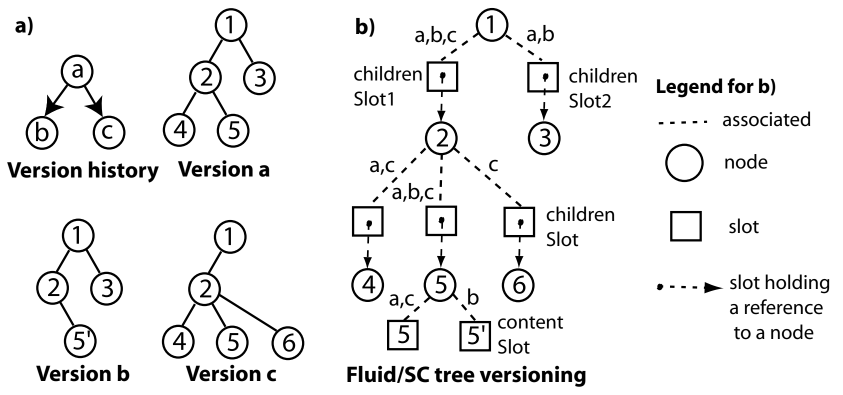


Figure 1: Document tree versioning

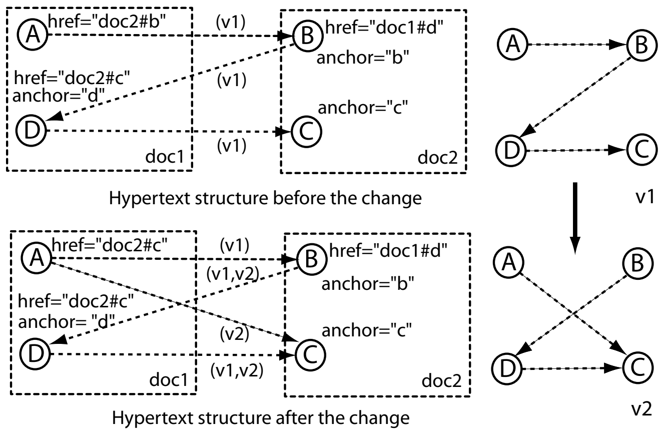


Figure 2: Versioning for Hypertext Structures

The SC representation of the three versions is shown in Figure 4 b). In our representation, the versions share a single tree. In the tree, each node is associated with slots that store the references to *parent* and to *child sequence* of the node. For simplicity, parent slots are not shown. The leave nodes also have content slots that store their contents. For clarity purpose, only the content slot of node *5* is shown. Depending on the current version, the tree knows how to retrieve the correct values of associated slots for a node. So, the association between node *1* and *childrenSlot1* is labeled "a,b,c" because it is active for all three versions, while the association between node *2* and the *childrenSlot* that refers to node *6* is only labeled "c" because node *6* is present only in version *c*. The content of node *5* is in the content slot *5* if the current version is either *a* or *c*, while it is in the content slot *5'* if the current version is set to *b*. Note that only difference between content slots *5* and *5'* is stored.

4.2 Versioning for hypertext structures

The term *hypertext structure* refers to the set of document nodes and hyperlinks between them. Our product versioning framework imposes a uniform global version space across an entire software project. For different versions, the hypertext structure might have different shapes.

Hypertext structure is versioned in the same manner as document trees. Suppose that we have two documents: *doc1* and *doc2*, each with two anchors and connected by three links as shown in Figure 2. At version *v1*, *A* links to *B*, *B* links to *D*, and *D* links to *C*. Suppose that the hypertext structure is changed and now node *A* points to node *C*, creating a new version *v2*. At a lower level, this means that the value of the *href* slot associated with node *A* is now "doc2#c". As a result of this change, the link between *A* and *B* is no longer present in version *v2*, and *C* can be reached directly from *A*. However, if a users decides to make version *v1* be the current version, the system will restore the attribute values to provide the correct hypertext structure for that version.

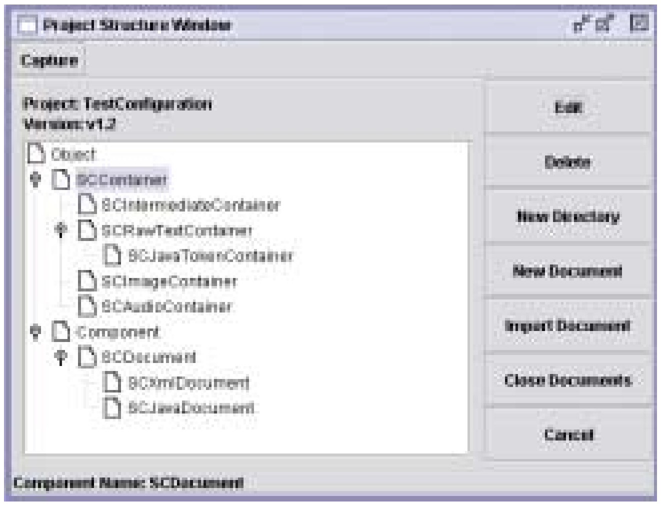
This scheme implies that no version selection rules are involved in structure versioning. The correct destination node for a link is implicitly determined by the version. This is an advantage of our system over the traditional composite-based versioned hypermedia systems [38], which have problems including difficulty with link and hypertext structure versioning, cognitive overhead in version creation and version selection rules, and navigation interfaces [25]. Because no version selection rules are involved in hypertext structure versioning, less overhead is imposed as users navigate through the versioned hypermedia space. Moreover, structure versioning is more efficient because all data items are versioned uniformly in a fine-grained manner. The environment also shows that the cognitive overhead of version creation can be reduced since users will operate at the level of an entire project rather than at the level of each individual component.

5. THE SC ENVIRONMENT

This section describes essential features of the Software Concordance environment. The SC editor is a uniform document editor, unifying a structured document editor for multimedia documentation in XML and a syntax-recognizing Java program editor. Users edit an XML document or a Java program in the same manner. Users can also edit their UML diagrams as in traditional UML editors. The SC environment also provides uniform, fine-grained version control supports for a software project's documents.

5.1 Project versioning

The configuration management system [23] of the SC environment provides version control services for both program source code and documentation. The CM system allows developers to create a new project (*project* is a hierarchical structure of software documents), to open an existing project, to display and modify project structure, to capture a new version, to branch off a new version from an existing version, to switch to work on different versions, to commit and save changes, and to discard changes. The version support is fine-grained, that is, it works at the structural unit level, rather than at the line level as in traditional version control systems such as CVS [21] or RCS [35].

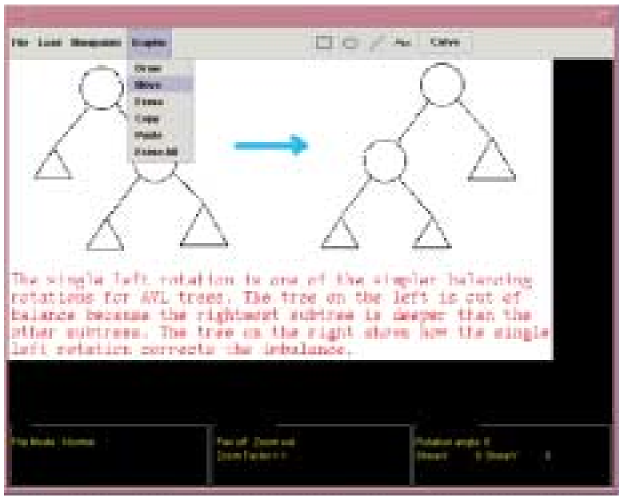


Project: TestConfiguration
Version: v1.2

```
Object
  SCContainer
    SCIntermediateContainer
    SCRawTextContainer
      SCJavaTokenContainer
    SCImageContainer
    SCAudioContainer
  Component
    SCDocument
      SCXmlDocument
      SCJavaDocument
```

Component Name: SCDocument

Figure 3: The project structure window



The single left rotation is one of the simpler balancing rotations for AVL trees. The tree on the left is out of balance because the rightmost subtree is deeper than the other subtrees. The tree on the right shows how the single left rotation corrects the imbalance.

Figure 5: Image and graphic editor

After users open a project and select a version via version tree (or a default version is chosen automatically), a project structure window is shown (Figure 3) to present the project hierarchy at the selected version. This window plays some important roles. Firstly, it serves as a navigation tool for the whole project. That is, users can choose to display and to work on any component appeared in the window. Secondly, it provides an overview for the project at the selected version. Thirdly, via this window, users can modify the project structure. Finally, when users switch their focus onto the window, the system will automatically set the current working version to the version displayed in the window. From the project structure window, users can create a directory; create and open documents; delete, rename and move components in the hierarchy; close all documents; capture a version; commit changes, and discard changes that they made to the current version.

5.2 The user interface

A screen shot from an editing session is shown in Figure 4. This session's rightmost window shows part of the implementation of an AVLTree class, specifically the simpleLeft method. The source code has been pretty-printed automatically using information from syntax analysis. The details of how the code is pretty-printed are controlled by a simple style sheet. Appearing above the method's source code is some multimedia documentation, including a diagram of the simple left rotation operation, a textual description (but *not* a comment), and buttons that control an audio clip of the implementor explaining details of the implementation. In addition, the developer is able to hide all these multimedia documentation in a program presentation.

On the left, the developer has another window open on the same source code. This window shows the entire class, but using a style sheet that hides the method bodies and multimedia documentation so that only the class interface is visible. This is simply another view of the same information shown in the rightmost window. The two views are different only because they are presented using different style sheets. The developer can edit the document through either window and any changes will appear in both.

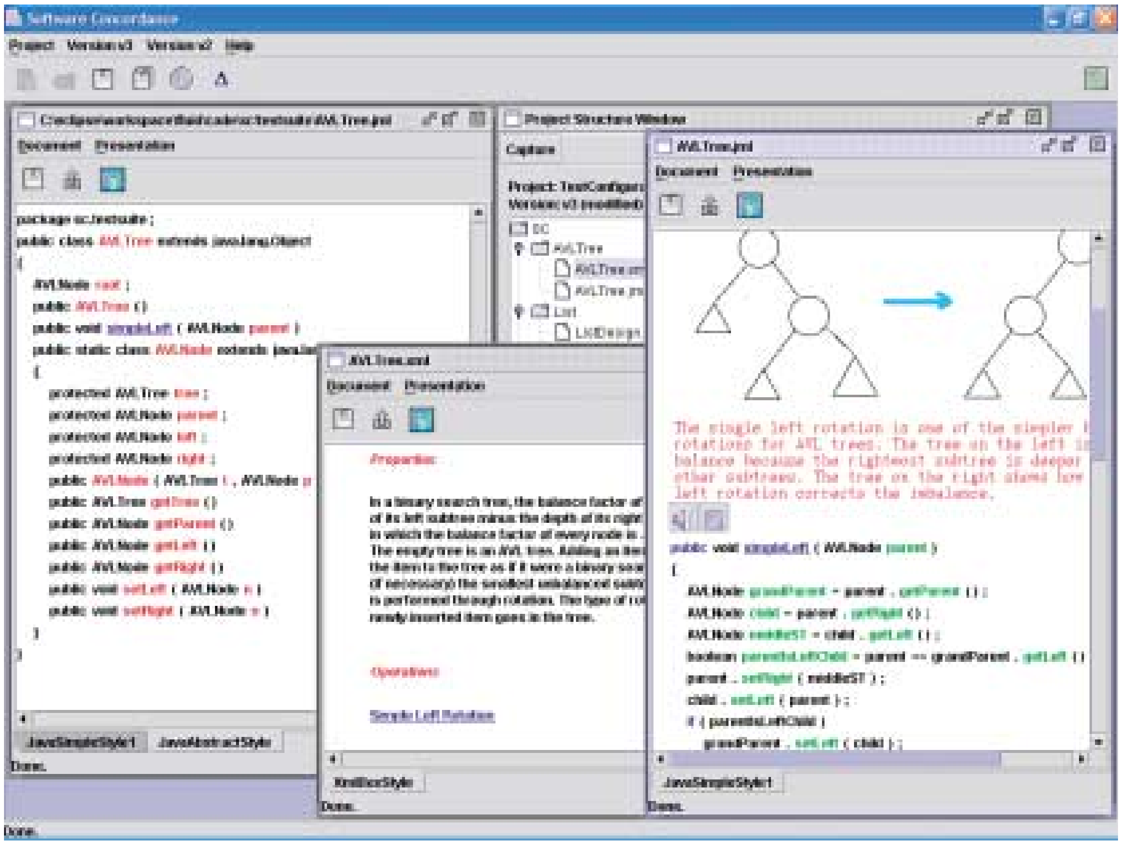
The center window shows a design document for the AVLTree class. This window was brought up because the developer clicked on the word "simpleLeft" in one of the other windows. This word happens to be a hyperlink that points to the "Simple Left Rotation" heading in the design document. This heading is itself a hyperlink that can be followed back to the simpleLeft method in the source code. The window on top shows the structure of the project at the developer's working version.

5.3 Editing in the SC environment

A user interacts with the system using menu, tool bar and contextual pop up menu. Suppose the user opens a document. To display a document, a default style sheet is selected by the system for the document. The presentation of the document is built based on the document tree and the style information. The user can choose to open a document with any style sheet.

To edit a document, the user moves the mouse and selects any structural unit of the document that needs to be edited. Then, via the commands in the pop up menu, he can choose to edit the content of that structural unit presented in the selected portion of the presentation, or to edit the documentation associated with that unit. The editor invokes the node editor. The *node editor* is a simple ASCII text editor. The SC system unparses the node and displays the resulting textual representation of the node to be edited. The user edits the text and commits the changes. The system incrementally parses the modified text, creates new nodes and attach them to the document tree. Depending on the type of the document that is being edited, the SC system invokes either an XML parser or a Java parser to incrementally parse the modified text. If there exists any errors in the modified text, error messages are displayed and the user can fix them. To delete a structural unit, the user invokes the deleting command in the same pop up menu.

To edit the image or graphic documentation associated with a structural unit, the user invokes an image and graphic editor. The image and graphic editor is shown in Figure 5. When the user finishes with his images, the Software Concordance editor adds the resulting images into the document presentation window. To associate audio documentation with a structural unit, the user invokes an audio selection dialog to choose an audio file. The user can create and edit a hyperlink or an anchor using steps similar to the ones needed to edit the documentation. The user can enter a URL or choose a file from a selection dialog. When some changes were made to the document, all presentations that currently displayed the document at the current version automatically update themselves to reflect the changes in the document and the screen layout is changed accordingly.



```
package sc.testsuite ;
public class AVLTree extends java.lang.Object
{
  AVLNode root ;
  public AVLTree ()
  public void singleLeft ( AVLNode parent )
  public static class AVLNode extends java.la
  {
    protected AVLTree tree ;
    protected AVLNode parent ;
    protected AVLNode left ;
    protected AVLNode right ;
    public AVLNode ( AVLTree t , AVLNode p
    public AVLTree getTree ()
    public AVLNode getParent ()
    public AVLNode getLeft ()
    public AVLNode getRight ()
    public void setLeft ( AVLNode n )
    public void setRight ( AVLNode n )
  }
}
```

Properties:

In a binary search tree, the balance factor of
of its left subtree minus the depth of its right
in which the balance factor of every node is
The empty tree is an AVL tree. Adding an item
the item to the tree as if it were a binary sear
(if necessary) the smallest unbalanced subtr
is performed through rotation. The type of ro
newly inserted item goes in the tree.

Operations:

Simple Left Rotation

The single left rotation is one of the simpler
rotations for AVL trees. The tree on the left is
balance because the rightmost subtree is deeper
other subtrees. The tree on the right shows how
left rotation corrects the imbalance.

```
public void singleLeft ( AVLNode parent )
{
  AVLNode grandParent = parent . getParent () ;
  AVLNode child = parent . getRight () ;
  AVLNode middleST = child . getLeft () ;
  boolean parentIsLeftChild = parent == grandParent . getLeft ()
  parent . setRight ( middleST ) ;
  child . setLeft ( parent ) ;
  if ( parentIsLeftChild )
    grandParent . setLeft ( child ) ;
```

Figure 4: Screen shot of the Software Concordance Editor

To edit a UML diagram, the user invokes the UML diagram editor (shown in Figure 6). It is a specialized graphic editor that allows the user to edit UML diagrams such as class diagrams, use case diagrams, sequence diagrams, activity diagrams, etc. The system can be extended to incorporate editing services for any multimedia documentation as well. For example, an editor for a new medium that follows the SC protocol can be easily integrated into the system.

6. DISCUSSION AND CONCLUSION

The Software Concordance research has built a software document management and development environment for Java programs and multimedia documentation in XML. The environment improves the interoperability between program source code and non-program software documents via its uniform document representation that allows for inline multimedia documentation within source code and for hyperlinks between any fragment of documents while still supporting for program analysis services such as lexing, parsing, and type checking. The environment also supports fine-grained version control for software documents via its configuration management system. A stylesheet-based presentation system allows users to have different document views for various tasks that they perform.

The SC environment is useful for both developers and non-programmer users as well. Documenters, without any understanding of the source code, could use the system to edit their documentation in XML without risking breaking links or missing making ones. Moreover, documenters could take advantages of self-documented programs from developers. With existing documenting tools, developers' comments have been underexplored. Therefore, this research also contributes to the collaborative documentation process. The documenters are able to import the existing documentation in XML format into the SC environment at any version and can also export program source code and multimedia documentation to plain Java or XML format, and then use any external tools to edit them. Therefore, that makes the SC environment XML-compatible and users do not need to edit every document inside the SC environment. In addition, Software Concordance could also be used as a structured document authoring tool for XML documents.

The SC environment is still at the research stage. We are in the process of evaluating the usability of the system with formal studies. Analyzing and visualizing tools are being developed to help developers to have better understanding of the graph of relationships among the software documents. The UML editor is being improved to support more types of UML diagrams. The services for consistency and change management are being developed based on our versioned hypermedia infrastructures [24].

7. REFERENCES

[1] Adobe FrameMaker. http://www.adobe.com/products/framemaker/.

[2] Maurice Amsellem. ChyPro: A hypermedia programming environment for SmallTalk-80. In *Proceedings of ECOOP'95*, August 1995.

[3] Kenneth M. Anderson, Richard N. Taylor, and E. James Whitehead, Jr. Chimera: hypermedia for heterogeneous software development

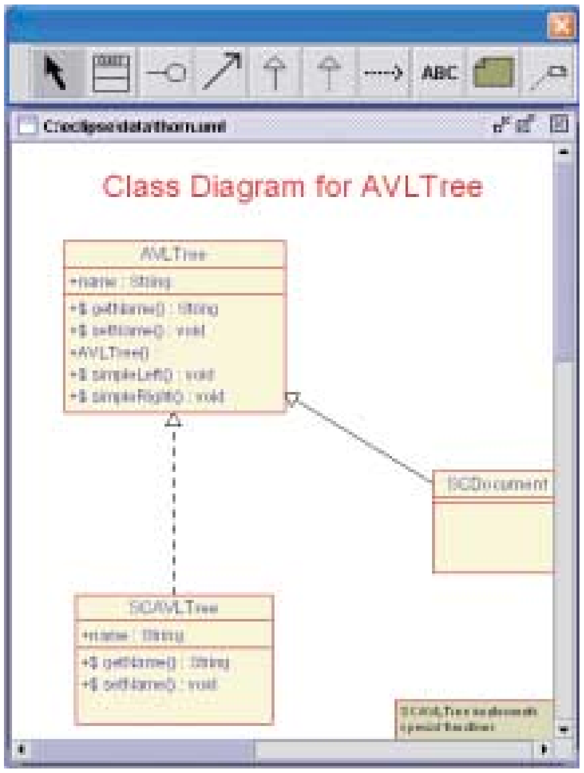


Figure 6: UML diagram editor

enviroments. *ACM Transactions on Information Systems (TOIS)*, 18(3):211–245, 2000.

[4] Greg Badros. JavaML: A markup language for java source code. In *WWW9: 9th International World Wide Web Conference*, pages 159–177, May 2000.

[5] Robert A. Ballance, Susan L. Graham, and Michael L. Van de Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, January 1992.

[6] James Bigelow and Victor Riley. Manipulating source code in DynamicDesign. In *Proceedings of the Hypertext'87*, pages 397–408, 1987.

[7] John Boyland, Aaron Greenhouse, and William L. Scherlis. The Fluid IR: An internal representation for a software engineering environment. In preparation. For information see `http://www.fluid.cs.cmu.edu`.

[8] Cybulski and Reed. A Hypertext Based Software Engineering Environment. *IEEE Software*, 9(2):62–68, March 1992.

[9] P. Devanbu, Y.-F. Chen, E. Gansner, H. Müller, and J. Martin. CHIME: customizable hyperlink insertion and maintenance engine for software engineering environments. In *Proceedings of the ICSE*, pages 473–482, 1999.

[10] James C. Ferrans, David W. Hurst, Michael A. Sennett, Burton M. Covnot, Wenguang Ji, Peter Kajka, and Wei Ouyang. HyperWeb: a framework for hypermedia-based environments. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 1–10. ACM Press, 1992.

[11] Pankaj K. Garg and Walt Scacchi. A hypertext system to manage software life-cycle documents. *IEEE Software*, 7(3):90–98, May 1990.

[12] Anja Haake and David Hicks. VerSE: towards hypertext versioning styles. In *Proceedings of the seventh ACM conference on Hypertext*, pages 224–234. ACM Press, 1996.

[13] Wendy Hall, Gary Hill, and Hugh Davis. The Microcosm link service. In *Proceedings of the fifth ACM conference on Hypertext*, pages 256–259. ACM Press, 1993.

[14] David L. Hicks, John J. Leggett, Peter J. Nrnberg, and John L. Schnase. A hypermedia version control framework. *ACM Transactions on Information Systems (TOIS)*, 16(2):127–160, 1998.

[15] Ellis Horowitz and Ronald Williamson. SODOS: a software documentation support environment—its use. *IEEE Transactions on Software Engineering*, SE-12(11):1076–1087, November 1986.

[16] J. L. Knudsen, M. Löfgren, O. Lehrmann-Madsen, and B. Magnusson. *Object Oriented Environments: The Mjølner Approach*. The Object-Oriented Series. Prentice Hall, 1993.

[17] Donald Knuth. *Literate Programming*, volume 27 of *Center for the Study of Language and Information — Lecture Notes*. CSLI Publications, January 2001.

[18] Alessandra Alaniz Macedo, Maria da Graca Campos Pimentel, and Jose Antonio Camacho-Guerrero. An infrastructure for open latent semantic linking. In *Proceedings of the thirteenth conference on Hypertext and hypermedia*, pages 107–116. ACM Press, 2002.

[19] 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.

[20] Microsoft Office. http://www.microsoft.com/.

[21] Tom Morse. CVS. *Linux Journal*, 1996(21es):3, 1996.

[22] 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.

[23] Tien N. Nguyen, Ethan V. Munson, and John T. Boyland. State-based product versioning in the Fluid/SC CM system. In preparation.

[24] Tien N. Nguyen, Satish Chandra Gupta, and Ethan V. Munson. Versioned hypermedia can improve software document management. In *Proceedings of the Thirteenth Conference on Hypertext and Hypermedia*. ACM Press, 2002.

[25] Kasper Østerbye. Structural and cognitive problems in providing version control for hypertext. In *ECHT '92, Proceedings of the ACM conference on Hypertext*, pages 33–42, 1992.

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

[27] Kasper Østerbye and Kurt Nørmark. An interaction engine for rich hypertext. In *ECHT '94, Proceedings of the 1994 ACM European conference on Hypermedia technology*, pages 167–176, 1994.

[28] Rational Rose. http://www.rational.com/products/rose/index.jsp.

[29] Steven P. Reiss. Simplifying data integration: The design of the Desert software development environment. In *Proceedings of the 18th International Conference on Software Engineering*, pages 398–407, 1996.

[30] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998.

[31] M.-A. D. Storey, C. Best, and J. Michaud. SHriMP Views: An interactive and customizable environment for software exploration. In *Proceedings of IWPC'2001*, May 2001.

[32] Javadoc tool home page. http://java.sun.com/j2se/javadoc/.

[33] Tim Teitelbaum and Thomas W. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, Sep 1981.

[34] Thorn UML editor. http://thorn.sphereuslabs.com/.

[35] Walter F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th international conference on Software engineering*, pages 58–67. IEEE Computer Society Press, 1982.

[36] Walter F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[37] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. PhD thesis, University of Califonia – Berkeley, 1998.

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

[39] Uffe K. Wiil and John J. Leggett. Hyperform: using extensibility to develop dynamic, open, and distributed hypertext systems. In *Proceedings of the ACM conference on Hypertext*, pages 251–261. ACM Press, 1992.