Model-based Version and Configuration Management for a Web Engineering Lifecycle

Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University
tien@iastate.edu

ABSTRACT

During a lifecycle of a large-scale Web application, Web developers produce a wide variety of inter-related Web objects. Following good Web engineering practice, developers often create them based on a Web application development method, which requires certain logical models for the development and maintenance process. Web development is dynamic, thus, those logical models as well as Web artifacts evolve over time. However, the task of managing their evolution is still very inefficient because design decisions in models are not directly accessible in existing file-based software configuration management repositories. Key limitations of existing Web version control tools include their inadequacy in representing semantics of design models and inability to manage the evolution of model-based objects and their logical connections to Web documents. This paper presents a framework that allows developers to manage versions and configurations of models and to capture changes to model-to-model relations among Web objects. Model-based objects, Web documents, and relations are directly represented and versioned in a structure-oriented manner.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Configuration management;
H.5.4 [**Information Interfaces and Presentation**]: Hypertext / Hypermedia

General Terms

Documentation, Management

Keywords

Web Engineering, Model-based Configuration Management, Versioned Hypermedia

1. INTRODUCTION

A lot of efforts and time have been spent to develop and to maintain Web-based applications in the daily basis. Web sites with a few dozens of static HTML pages can be developed using technologies and tools for *ad hoc* development. The dramatic growth of the WWW leads to large-scale Web applications, often distributed over several sites and containing a large number of highly dynamic Web objects. Such large-scale, dynamic Web-based applications with complex navigational patterns and sophisticated computational behaviors are no longer manageable with ad hoc methods.

WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

Web development is no longer considered as a simple document authoring task. In fact, it requires the same rigorous methodologies and tools as in the successful *software engineering* discipline. In response, the discipline of *Web engineering* has emerged, advocating a systematic approach to development of high quality Web-based systems [21]. Several Web and hypermedia application development methods have been proposed, such as OOHDM [27], RMM [16], and WebComposition [12], guiding Web developers in their design and implementation tasks. Those methods often introduce logical models and associated entities to address different aspects of the development and maintenance process. For example, in OOHDM [27], the conceptual model allows developers to express relations among data objects, while the navigation model and presentation model address navigational structure and information presentation in a Web application, respectively.

Entities in design models are connected to each other and related to Web documents at the implementation level (e.g. HTML pages or scripts). The logical connections among them are crucial for a successful development process. For example, the relation between a particular navigational pattern and its realization via user-interface elements in HTML documents is important. Another example is the connection between an entity in a conceptual data schema and its corresponding entity in a navigational schema. This connection helps to express navigation paths to Web pages containing a particular piece of information.

During a Web engineering lifecycle, not only the logical models are in a state of evolution, but also the logical connections among their entities are changed as well. For example, in one version, a particular navigational path is implemented as *regular hyperlinks* embedded within HTML pages; in a newer version, it is realized as a *computation link* encoded within a Java script. Keeping track of changes to design/implementation models and the connections among their Web objects would tremendously benefit Web developers in their development and maintenance tasks.

To achieve that, one could use a software configuration management system (SCM) to record changes to Web artifacts which reflect development histories. SCM is a sub-discipline of software engineering that is concerned with the management of changes to a software system and its artifacts. However, due to the nature of a Web-based application, existing SCM approaches for Web engineering are not well-suited. With existing SCM systems, design decisions are not directly accessible. They are embedded within file-based resources stored in version control repositories. Those SCM approaches are either too general or too specific, and fail to manage the evolution of logical models, associated objects, and model-to-model relations. Key limitations of existing Web configuration management tools include their inadequacy in efficiently representing semantics of design models and inability to model changes to

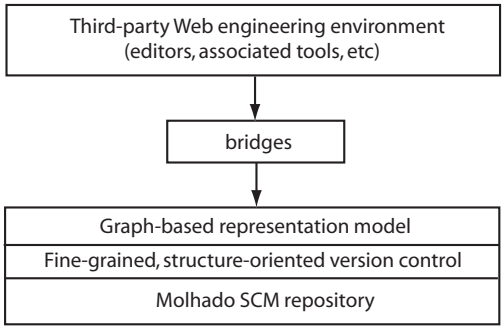


Figure 1: Framework

model-based Web objects and logical connections. Some SCM tools based on text (or binary) files are not well-suited for design models whose objects are often structured and have rich underlying semantics. They are focused on providing content change management mainly for Web documents making up a Web application at the implementation level. Those file-based Web SCM systems disregard the underlying syntactic and semantic structures of model-based Web objects. Changes between versions are handled at the text line level. Thus, it is largely up to Web developers to figure out changes at the model level.

In contrast, there are Web SCM tools that are designed toward specific types of Web objects (e.g. XML or HTML) with the knowledge about the languages being encoded within the tools. They work perfectly with markup files at the implementation level. However, the logical structures of model-based Web objects such as in complex design schemas might not be well reflected in tree-based structures of XML or markup documents. Managing changes at the tree-based structural level is apparently not suitable for model-based evolution. In brief, existing Web configuration management approaches make the development and maintenance process of Web applications inefficient and error-prone.

To address those issues, we have developed a framework that allows Web developers to manage versions and configurations of model-based Web objects, and to capture changes to model-to-model logical relations among them. The evolution of connections and dependencies among model-based design objects and Web documents at the implementation level can also be recorded.

2. OUR APPROACH

2.1 Background

The work that is presented in this paper is part of a research effort which aims at applying SCM techniques to Web and hypermedia applications. Our previous research result has produced *Molhado*, an object-oriented SCM repository, that is capable of managing versions of software artifacts at the object level [22]. The main target of Molhado is no longer a file in a file system. File is considered as just one type of object. All objects can be versioned and saved persistently in Molhado object-oriented SCM repository. Molhado allows developers to capture and retrieve changes to their software objects, without concerning about the concrete level of file storing and versioning in a file system.

We have also applied this object-oriented SCM technology to build the first version of an SCM-centered Web development environment, named *WebSCM* [23]. The first version of WebSCM has an extensible and pluggable architecture that allows for the integration of editors for any new document types whose internal representation is *XML-compatible*. WebSCM uses a Document Object Model (DOM) [8] parser to import tree-structured documents, converts the DOM trees into Molhado's document tree representation, and then manages them [23]. WebSCM is a structure-oriented environment that provides editors for several kinds of Web documents. It has the structured editors for XML, HTML, and a syntax-recognizing Java program editor (i.e. syntactical correctness of documents is always enforced). In brief, the first version of WebSCM provides a fine-grained, structure versioning services for only *tree-structured* Web documents in XML, HTML, and Java formats.

2.2 Framework Summary

The research results presented here are from our efforts to address the issues of managing the evolution of Web applications at the model level. As discussed earlier, the tree-based version and configuration management in Molhado and WebSCM was insufficient for managing the model-based evolution in a Web engineering lifecycle. Also, a third-party Web development environment might support a new design methodology and its associated model-based entities might have more complex structures. Therefore, we need to have a more generic representation model than XML document tree model used in the previous version of WebSCM.

A generic and domain-independent representation model is designed using a special data structure called *attributed, typed, nested*, and *directed graphs*. Via this model, it is able to capture logical structures of a wide variety of Web objects in logical models as well as Web documents. A novel structure-oriented versioning algorithm for that data structure has been developed to provide the fine-grained content change and version management for Web entities of logical models. More importantly, that representation allows us to take advantage of the storage and configuration management capabilities of the Molhado object-oriented repository, to store different versions of Web objects and linking structures. The model-to-model logical connections are managed via our versioned hypermedia infrastructure, in which linking structures are maintained separately from Web objects. Thus, it facilitates systematic analyzing and processing of those logical relations.

In a Web engineering/supporting environment that supports a particular design model or in an editor for a model-based Web object type, one could either directly use our representation for Web objects and linking structures, or build a *bridge* to act as a converter between the native representation model of that environment and our representation model (see Figure 1). The only requirement on the bridge is that it needs to call our graph library functions to update our repository whenever there are changes to Web objects in a third-party development environment. Currently, we require one bridge for each native Web object type.

We have developed a generic, structure-oriented differencing and merging algorithms for model-based Web objects and documents. They are extended to deal with both embedded and first-class *hypermedia structure*, which is defined as a collection of hyperlinks and connected Web objects. This is a novel contribution to hypertext versioning research since no attempt has been made to apply software merging techniques to versioned hypermedia documents.

Next section describes our structure-oriented, graph-based representation model. Section 4 explains how that representation model is implemented. A new fine-grained versioning mechanism is developed for this representation (Section 5). Section 6 presents the use of our graph-based representation model for hypermedia structures. The structure-oriented differencing and merging frameworks are presented in Section 7 and Section 8, respectively. The current status of our implementation is described in Section 9. Section 10 presents related work. Conclusions appear in the last section.

3. STRUCTURE-ORIENTED REPRESENTATION MODEL

This section describes a graph-based representation model for Web objects and hypermedia structures among them. Graphs are commonly known, well understood, have an established mathematical basis (graph theory), and encompass a huge number of concepts, methods and algorithms [19]. This makes them very interesting from a formal as well as a practical point of view. We use a special type of graphs, called *attributed, typed, nested,* and *directed graphs* to represent Web objects as well as hypermedia structures.

First of all, a directed graph can be defined as a tuple

$$G = \{N, E, source, sink\}$$

where N is a finite set of nodes (or vertices), E is a finite set of edges (or arcs), and $N \cap E = \emptyset$. $source$ and $sink$ are functions $source : E \rightarrow N$ and $sink : E \rightarrow N$ assigning exactly one source and target node to each edge. We allow multi-graphs where different edges can have exactly the same source and sink nodes. However, we do not allow hyper-graphs, which contain hyper-edges that have more than one source or target node.

Unlike document nodes in DOM or in many other XML-based document models, a node in our model has a unique identifier. A node has no values of its own. However, each node in a directed graph can be associated with multiple attribute-value pairs. That is, for each $n \in N$, there is an associated attribute table consisting of one or multiple attribute-value pairs (a_i, v_i) where a_i is an attribute name and v_i is an attribute value. An attribute name can be any *string* and must be uniquely identified. The domain of v_i can be any data type T, possibly the *reference* type. These typed attributes accommodate multiple properties associated with nodes.

In other graph-based representation models, a graph could be *labeled*, where nodes and edges are attached by labels of some types (often either string or integer). That is, $label = (nlabel : N \rightarrow NodeLabel, elabel : E \rightarrow EdgeLabel)$ is a pair of node-labeling and edge-labeling functions. Our model extends this labeling technique by allowing each edge in a directed graph to be associated with an attribute table in the same manner as a node.

Our model also allows a directed graph to be nested within another in order to support composition and aggregation among Web objects. Nesting is a natural way for humans to control the complexity of a system. In a nested graph, the overall complexity is reduced by allowing nodes to contain entire graphs themselves. Nested graphs are also referred to as *hierarchical graphs* [19]. This characteristic of a directed graph in our representation model is defined by a partial node mapping function: $nested : N \rightarrow N$, such that its corresponding relation $nested \subset N \times N$ is acyclic and loop-free. This constraint is needed to ensure that we have a nesting hierarchy and a proper composition mechanism, i.e., a node cannot be contained within itself. Using relation notation, $(n, m) \in nested$ denotes that n is directly nested in m.

The reason why attributed, typed, nested, and directed graphs are used to represent Web objects and hypermedia structures in our framework is manifold. Firstly, graphs are an intuitive, visually attractive, general and mathematically well-understood formalism. From the practical point of view, directed graphs are often used as an underlying representation of arbitrarily complex software artifacts and their interrelationships in traditional software engineering environments [19]. Also, graphs have already been used for describing and understanding a number of aspects of a software system such as program behavior, program control flow, structural and internal relations between parts of a system, etc. Applying to Web engineering, nodes of a graph can represent Web objects such as entities and relationships in RMM [16], classes and relations in OOHDM's conceptual schemas [27], user-interface components in a navigational model [27], methods and classes in a program and a script, and elements in XML or HTML documents, etc. The edges can be used to represent all kinds of relationships between these entities such as inheritance relations, dependencies, logical mappings, or navigational paths, etc. Directed graphs are sufficiently general to be used for a wide variety of Web objects, depending on the interpretation given to nodes and edges.

Secondly, a nesting mechanism is attached to the graphs to facilitate the composition and aggregation among Web objects. The nested graphs also enable an encapsulation and layering mechanism to reduce the complexity and to hide unimportant details of an artifact from others. Low-level dependencies between nodes can be abstracted to higher-level dependencies between the nodes in which they are nested. It is apparent that many forms of nesting occur in every phase of a Web application's lifecycle. For example, in design models, design schemas contain composite entities such as in OOHDM's conceptual schemas or abstract data view design schemas. At the implementation level, in scripts or programs, we can easily find nested methods, composite classes, packages, or nested elements in markup documents such as HTML.

Thirdly, the association of an attribute table to a node or an edge facilitates the modeling of complex Web artifacts and allows us to take advantage of underlying SCM and version control services for different data types provided by the Molhado repository [22]. Molhado is based on the attribute grammar technology and has a rigorous type system basis [22]. This association enables us to take advantage of that technology. Next section will explain how we implement this graph-based model in Molhado.

Finally, the popular DOM [8] and XML document tree models [17, 37] can be nicely encoded via this attributed, directed graph-based representation model since their trees form a sub-class of this type of graph. Thus, many types of implementation artifacts in a Web application such as HTML, XML, and other markup documents can be represented using our graph-based model.

4. STRUCTURE-ORIENTED VERSIONING

4.1 Data and Version Models

This section describes how the attributed, typed, nested, and directed graphs are implemented in Molhado's data model. First of all, we would like to summarize Molhado's data and version models. Details could be found in another document [22]. Figure 2 conceptually illustrates the main concepts in that data model: *node, slot,* and *attribute*. In our terminology, a node and an attribute are called *Intermediate Representation* (IR) node and attribute, respectively. An IR node is the basic unit of *identity*. An IR node has no values of its own — it has only its unique identity. A slot is a location that can store a value of any data type, possibly a reference to an IR node or a set of slots. A slot can exist in isolation but typically slots are attached to IR nodes, using an attribute. An attribute is a mapping from IR nodes to slots. It 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 data model can thus be regarded as *attribute tables* whose rows correspond to IR nodes and columns correspond to attributes. The cells of attribute tables are slots. Once we add versioning, the tables get a third dimension: the version (see Figure 2).

With version control added, there are three kinds of slots. A *constant slot* is immutable and can only be given a value once, when it is defined. A *simple slot* may be assigned even after it has been defined. The third kind of slot is the *versioned slot*, which may have

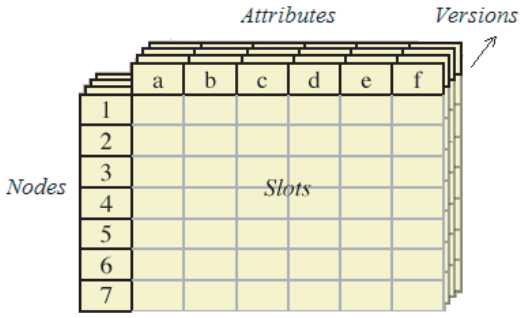


Figure 2: Data Model

different values in different versions (*slot revisions*). Any primitive data type of a slot can be versioned in Molhado. A slot may also exist in a *container*, an entity with identity and ordered slots. A container may be heterogeneous (a *record* in which each slot has its own type) or homogeneous (a *sequence* of slots of the same type). A container may be fixed or variable in size.

In Molhado's version model, a *version* is *global* across entire software system and is a point in a *tree-structured discrete time* abstraction. That is, the third dimension in the attribute table in Figure 2 is tree-structured and versions move discretely from one point to another. Molhado uses *product versioning* where a uniform global version space is maintained. The version model is state-based, where each version is a first class entity that represents a state of a system. A version can be associated with a name and meta-information such as date, time, authors, etc. The *current version* is the version designating the current state of a system. Any version may be made current. Every time a versioned slot is assigned a (different) value, we get a new version, branching off the current version. In brief, Molhado knows how to manage versions of slots in any data type, and how to store and correctly retrieve versioned slots that belong to a particular version point.

4.2 Mapping to the Data Model

Since a directed graph in our representation model is also based on attribute-value pairs, it is reasonably straightforward to realize it via Molhado's data model for versioning purpose. An attribute table is constructed for a directed graph as follows.

(a) Each graph node is represented by an IR node in the table. The associated attribute-value pairs of a graph node could be easily mapped into a row of the table. Attribute values are realized as slots associated with the corresponding IR node. The attributes in those attribute-value pairs are added into the set of IR attributes of that table. From now on, we simply refer to IR attributes as attributes.

(b) Each edge in the graph is also represented by a new IR node (i.e. a new row) in the attribute table. Let us call it an "edge" node. The associated attribute-value pairs of the edge are integrated into the attribute table as in (a). Furthermore, for each "edge" node, two additional attributes are defined: "sink" attribute defines the target node of the edge, and "source" attribute defines its source node.

(c) For each IR node that is used to represent a graph node, an additional "children" attribute defines a slot containing a reference to a sequence of outgoing edges of the node. An example of this process will be described later (see Figure 3).

4.3 Nesting Mechanism

To handle nested graphs, we make use of the composite component version control mechanism in Molhado [22]. In this mechanism, A Molhado *component* is an entity that represents a *logical object* in a software system. It can be versioned, persistently saved, loaded from disk, and exists within the version space of a software system. In Web engineering, a component can be used to model a Web object in any phase of a software lifecycle. Each component carries a component identifier that serves to identify it *uniquely* within a Web system. Components are classified into two groups: *atomic* and *composite* components. An *atomic component* can *not* contain other components, but might have internal structure. A *composite component* is defined as a composition or aggregation of atomic components and/or other composite components. Composite components can share the same constituent components, and have arbitrary internal structure.

In our representation model, a directed graph that contains other graphs will have at least one node that *logically* contains another directed graph. Let us call that type of directed graph "composite" graph and that type of node "composite" node. Otherwise, let us call it an "atomic" graph. In our framework, an "atomic" or "composite" directed graph is encoded within a Molhado's atomic or composite component, respectively. In other words, a directed graph is used as the internal structure of a Molhado component. For a "composite" graph, an additional attribute, attribute "ref", is created to define for each "composite" node a versioned slot containing a reference to the Molhado component that corresponds to the subgraph nested at that "composite" node.

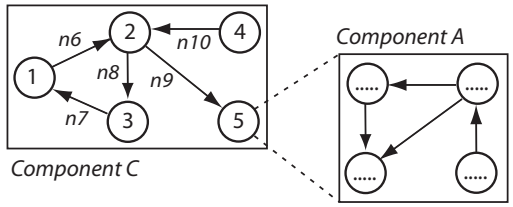
Figure 3 shows an example of the representation of an attributed, typed, nested, and directed graph using Molhado's data model. There are two graphs in the figure: the directed graph corresponding to the component A is nested within the directed graph corresponding to the component C via the node 5. The attribute table in Molhado representing for component C is shown. Nodes "n1" to "n5" are IR "node" nodes (i.e. representing for a graph node) while nodes "n6" to "n10" are IR "edge" nodes (i.e. representing for an edge). Each "edge" node has "source" and "sink" slots. For example, "edge" node "n6" "connects" IR nodes "n1" and "n2". Each "node" node has a children slot. For example, "n2" has two outgoing edges ("n8" and "n9"). Node 5 has no outgoing edge, thus, the "children" slot of "n5" contains *null*. However, it is also a *composite* node, therefore, its "ref" attribute refers to the component A. The attribute table for component A is similar (not shown).

5. WEB CONFIGURATION MANAGEMENT

Previous section presented our graph-based representation model for Web applications. Our goal is to provide structure-oriented version control supports for model-based Web objects, Web documents, and hypermedia structures. Thus, a fine-grained, structure-oriented version control scheme for that type of directed graphs is required. This section presents such a scheme, which takes advantage of Molhado repository. We also explain how configurations are maintained among atomic and composite graphs.

5.1 Fine-grained Version Control

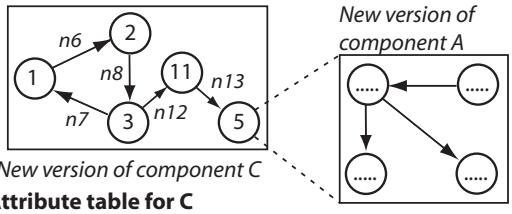
Our framework is based on the assumption that either one of these following conditions holds: (1) the associated tools or editors for model-based Web objects in a third-party Web engineering environment directly use our graph-based representation model for those objects and will call our provided library functions for graphs and attributes in order to modify the objects' structures or properties; or (2) if those editors/tools for logical models have their own native representations, the bridges/converters need to be constructed and call those library functions for graphs and attributes to reflect changes made to Web objects in the environment. Those functions will then update the values of slots in attribute tables including structural slots (i.e. "children", "source", and "sink").



Attribute table for C

IR node	"type"	"source"	"sink"	"children"	"ref"	"attr1"	....
n1	*node*	*undef*	*undef*	[n6]	null	....	
n2	*node*	*undef*	*undef*	[n8,n9]	null	....	
n3	*node*	*undef*	*undef*	[n7]	null	....	
n4	*node*	*undef*	*undef*	[n10]	null	....	
n5	*node*	*undef*	*undef*	null	comp_A	....	
n6	*edge*	n1	n2	*undef*	null	....	
n7	*edge*	n3	n1	*undef*	null	....	
n8	*edge*	n2	n3	*undef*	null	....	
n9	*edge*	n2	n5	*undef*	null	....	
n10	*edge*	n4	n2	*undef*	null	....	

Figure 3: Nested Graph Representation



Attribute table for C

IR node	"type"	"source"	"sink"	"children"	"ref"	"attr1"	....
n1	*node*	*undef*	*undef*	[n6]	null	....	
n2	*node*	*undef*	*undef*	[n8]	null	....	
n3	*node*	*undef*	*undef*	[n7, n12]	null	....	
n4	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	
n5	*node*	*undef*	*undef*	null	comp_A	....	
n6	*edge*	n1	n2	*undef*	null	....	
n7	*edge*	n3	n1	*undef*	null	....	
n8	*edge*	n2	n3	*undef*	null	....	
n9	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	
n10	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	
n11	*node*	*undef*	*undef*	[n13]	null	....	
n12	*edge*	n3	n11	*undef*	null	....	
n13	*edge*	n11	n5	*undef*	null	....	

Figure 4: Graph-based Version Control

Figure 4 displays a new version of C and A shown in Figure 3. In the new version, the attribute table was updated to reflect the changes to the graph structure as well as to the slot values. For example, since node 4 and edges corresponding to "n9" and "n10" were removed, any request to attribute values associated with those nodes will result in an undefined value. On the other hand, node 11 and two edges were inserted, thus, one new "node" node ("n11") and two new "edge" nodes ("n12" and "n13") were added into the table. Attribute values of these nodes were updated to reflect new connections. Attribute values of existing nodes were also modified. For example, "children" slot of "n3" now contains an additional child ("n12"), since that new edge ("n12") comes out of node 3. The attribute table for component A was similarly updated.

This fine-grained versioning scheme is very efficient since common structures are shared among versions and all information including structures and contents are versioned via one mechanism. Importantly, this scheme is general for any subgraph at a node. Therefore, fine-grained version control can be achieved for any Web entity in a logical model that is represented by a node. Molhado's storage mechanism is able to handle efficiently these three-dimensional attribute tables, which sometimes could be sparse.

5.2 Configuration Management

The graph-based structure versioning scheme enables the fine-grained version management of models and their associated Web objects. However, managing the evolution of a Web application is much more than versioning of individual Web entities. *Web configuration management* for a large-scale Web application must include other functionality such as consistent configuration management, transaction support, workplace management, and merging and differencing functionality for different versions of artifacts [22].

Fortunately, most of these tasks could be accomplished by using Molhado SCM infrastructure except the last two tasks. Molhado has a highly reusable and tailorable architecture [22]. This attributed directed graph-based representation model perfectly conforms to the Molhado framework as shown in the previous section. Therefore, it is possible to re-use those SCM services. Take a subtask of maintaining consistent configurations among Web objects as an example. The issue is how to determine the right versions of member objects for a version of a composite object. For an atomic object, via the Molhado's product versioning mechanism, when the current version of the Web application is globally selected, the values of properties (represented by versioned slots) and the internal structure of the object (represented by a directed graph, if any) will be correctly determined since Molhado knows how to retrieve versioned slots belonging to the current version. Similarly, a version of a composite object is easily retrieved: first of all, the internal structure of the composite object (i.e. a directed graph) is correctly retrieved after the current version is selected. Then, the "ref" versioned slots of "composite" nodes will refer to proper member objects of the composite object at the current version as well. The same process continues for each member object.

In our previous research [23], the merging and differencing tools were specifically designed for HTML, XML, Java scripts, and other *tree-structured* Web artifacts at the implementation level. The approach does not scale to model-based *graph-structured* objects. Our novel differencing/merging algorithms will be described later.

5.3 An Example

Figure 5 shows an example of our representation for an OOHDM conceptual schema. In OOHDM, conceptual design is the elaboration of a model of the application domain and determines the universe of discourse [27]. A conceptual schema is built upon classes, relationships, and sub-systems. Classes are described as usual in object-oriented models, though attributes may be multi-typed, representing different perspectives of the same real-world entity.

In Figure 5, a conceptual schema for an online newspaper is displayed. There are stories, which can be essays or interviews. Every story has an author, and an interview is related to the person who grants the interview. Class and relation in a conceptual schema are defined as Molhado atomic components. To represent this schema, we use our attributed directed graph model. Each entity (class, relation, etc) is represented by a node except that each inheritance relation is represented by an edge (e.g. between "n1" and "n2"). Edges connect nodes together to reflect the relationships in the schema. The "ref" attribute defines for each node a reference to the cor-

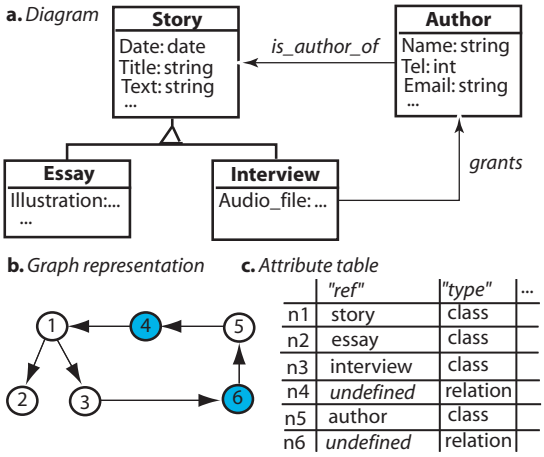


Figure 5: Conceptual Data Modeling

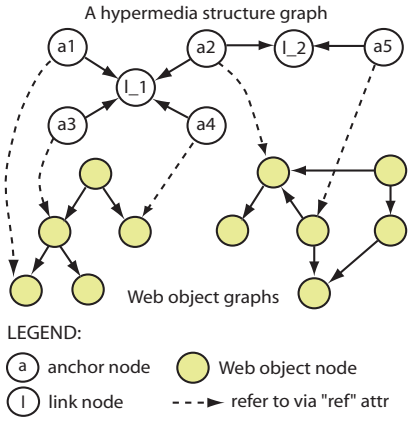


Figure 6: Hypermedia Structure Representation

responding component except for the "relation" nodes (e.g. "n4" and "n6" in Figure 5). Each property of a class and its data type are represented by attributes "property_name" and "data_type" associated with the class component node (e.g. "n1"). The "relation" nodes are associated with additional attribute-value pairs representing other properties of the relations such as arity, name, etc. Figure 5c) shows the partial attribute table.

Data records for objects instantiated from a class are stored in an attributed tree with the depth of two, branching off the class node. Each of those tree nodes at the first level represents a record. Each record can be associated with a sequence of fields, which are represented by children nodes of the record node. Each of these nodes has additional attributes such as "name", "type", and "value" to represent the name, type, and value of each field. In general, the conceptual schema and data records for objects are represented as attributed directed graphs and trees at two abstract levels: the schema level and individual data object level.

6. MODEL-TO-MODEL RELATIONSHIPS

Web objects in design models are related to each other and to Web documents at the implementation level. As discussed earlier, maintaining the logical connections among them over time is crucial for Web developers in having better understanding of the system's evolution. To manage model-to-model logical relations, we use the first-class hypermedia structures in which a link is repre-

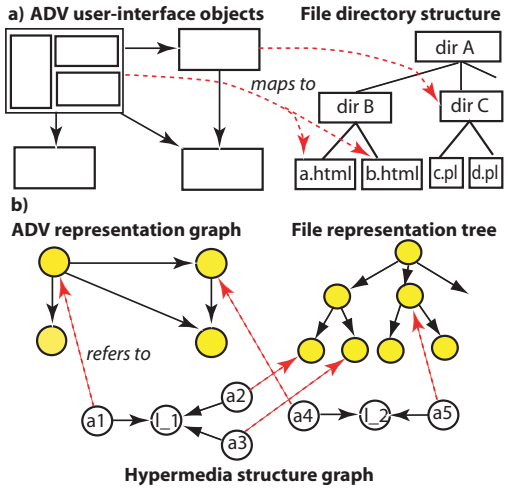


Figure 7: Mappings from Design to Implementation

sented as a first-class entity such as in XLink standard [40]. The advantages of first-class hyperlinks have been acknowledged by hypermedia research communities [38]. For example, they facilitate the process of browsing, visualizing, and analyzing of relationship networks among Web objects in different logical models.

We have built a hypermedia model that supports first-class hypermedia structures [22]. This section summarizes its core concepts. In that model, a *linkbase* is a container for *hypertext networks* and/or other linkbases. A *hypertext network*, which represents a hypermedia structure, 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 is n-ary and connects a set of its anchors together. 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 object. It *refers* to a graph node within a Web object or to entire object. Attribute-value pairs can be associated with any link or anchor.

To apply our versioning services to first-class hypermedia structures, we realize our hypermedia model via the graph-based representation. In particular, a hypertext network is implemented as an atomic component, whose internal structure is a directed graph. Each link or anchor is represented by a node in that graph (Figure 6). A directed edge connects an anchor's node to a link's node if the link contains the anchor. Attribute "ref" associates a slot to each anchor's node in the graph. The slot holds a reference to either a component or a graph node within a component. That node (or that component) is considered to be the position of the anchor. This separation between anchors and object nodes allows for the separation between hypertext networks and the contents of components. Also, a linkbase is implemented as a composite component, whose internal structure is a tree. In brief, a hypertext network, representing for a hypermedia structure, is modeled and versioned according to the graph-based versioning scheme as described earlier.

Figure 7 illustrates the use of our hypermedia infrastructure to maintain the logical mappings from user-interface objects in Abstract Data View (ADV) design model [27] to HTML documents or scripts at the implementation level. Figure 7b) displays the graph representing the hyperlink structure for the logical mappings. Note that the hyperlink graph is separated from the representation graphs and trees for Web objects/documents. Since a hypertext structure is realized as a Molhado component, multiple model-to-model re-

lationship networks can be defined for different purposes, without embedding multiple sets of HTML hyperlinks into Web documents.

7. STRUCTURAL DIFFERENCING

One of the basic functionalities in a SCM system is a differencing tool which displays changes between different versions of an artifact. Since model-based Web objects are represented as attributed directed graphs, a *structure-oriented* differencing algorithm is required for that type of graph. This section describes such an algorithm. A third-party Web engineering environment could use this algorithm, which is realized as API library functions, to build a specialized differencing tool for its supported Web objects.

Suppose that we have two versions V_1 and V_2. The question is how to determine if a node or an edge has been *deleted*, *inserted*, or *moved*, and if an associated attribute table has been *modified*. To detect if an attribute table and its values have been changed, we use a mechanism called *Versioned Unit Slot Information* (VUSI) [23]. VUSI attaches a "dirty" bit to a slot (containing an attribute value). Those bits will be set to "true" to signify changes. They are also saved into the repository for later retrieval. Note that functions for slots are called directly or indirectly (via the bridge) by editors.

To detect the deletion of a node or an edge from a directed graph, the values of structural attributes (i.e. "source", "sink", and "children") are examined. A special value (*undefined*) signifies a deletion. The removed node (or edge) is not permanently deleted in the SCM repository since it still exists in previous versions. The insertion of a node or an edge to a directed graph is signified by the appearance of a new row in the representation attribute table. To detect the relocation of a sub-graph, we examine the change of the "source" value of an "edge" node. If the "source" slot refers to a different node, the sub-graph starting from the "sink" node of that edge is relocated. These detection functions are applicable to any node and edge. A function to return differences between two arbitrary versions of an attributed graph is also provided.

There are several characteristics of our framework that make this structural differencing algorithm simple, efficient, and accurate. Firstly, *unique identifiers* of nodes and edges facilitate the maintenance of Web object histories, especially when objects are relocated. Furthermore, the unique identifiers are *immutable*. Secondly, we assume that the editors of third-party Web engineering environments for objects are *structure-oriented*, in which the operations will *preserve* those identifiers. Finally, the actual development history is accessible since our API functions for graph structures and attribute values are called by the bridge whenever Web objects are modified in a third-party environment. Therefore, changes that were actually performed from one version to another could be easily reconstructed by pairwise comparisons of versions without dealing with sequences of actual operations explicitly.

Note that our structural differencing algorithm is efficient because it does not use complex directed graph comparison algorithms as in many existing tools. We have applied this algorithm to build structural differencing tools for OOHDM's conceptual design diagrams, Java source code, HTML, and XML documents, and integrated them into WebSCM (see Section 9).

8. STRUCTURE-ORIENTED MERGE

8.1 Merging of Versions of Attributed Graphs

This section presents a generic three-way merge framework for attributed, directed graphs. In our merge framework, the information in the common ancestor from which both versions originated is also used during the merge process. Suppose that starting from a base version B, there are two alternative versions A_1 and A_2. A merge version M needs to be constructed which combines A_1 and A_2 with respect to B. When a conflict is detected during the merge process, developers will receive a message containing detailed description of the conflict. The principle of our merge algorithm is to analyze the presence and absence of nodes/edges and associated attribute values in those three versions. Depending on a particular scenario, different action would be invoked. The result is the complex case scenario analysis as follows.

Case 1: A node n satisfies: $n \in B$, $n \in A_1$ and $n \in A_2$. Node n will be added into the merged version M. Consider the attribute table of n. If there is an attribute-value pair that was inserted in one branch but not in the other, then it will be also inserted in M. If an attribute-value pair is deleted in one branch, but the value was modified in the other, a **conflict** will be notified. However, if the value was not modified in the other branch, we do not add that attribute-value pair into M. If the attribute set is unchanged, we analyze each attribute-value pair (a_i, v_i) of node n:

- From B to A_1, if v_i has been changed, and from B to A_2, it has also been changed, a **conflict** will be notified.
- From B to A_1, if v_i has been changed, and from B to A_2, it has *not* been modified: in merged version M, a_i will get the new value at A_1.
- From B to A_2, if v_i has been changed, and from B to A_1, it has *not* been modified: in merged version M, a_i will get the new value at A_2.
- From B to A_1, if v_i is un-changed, and from B to A_2, it is also un-changed: in merged version M, a_i will get the same value as the one at A_1 (or A_2).

Case 2: A node n satisfies: $n \in B$, $n \in A_1$, and $n \notin A_2$ (i.e. n was deleted at version A_2). If all attribute-value pairs associated with n are un-changed from B to A_1, we do not add n into version M. Otherwise, a **conflict** is notified.

Case 3: A node n satisfies: $n \in B$, $n \notin A_1$, and $n \in A_2$ (i.e. n was deleted at version A_1). Similar to Case 2.

Case 4: A node n satisfied: $n \in B$ but $n \notin A_1$ and $n \notin A_2$. That is, the node was deleted in both branches. Then, it will not appear in version M either.

Case 5: A node n satisfied: $n \notin B$ but $n \in A_1$ and $n \in A_2$. That is, the node was inserted in both branches. It will also be added at version M. Then, we do the same analysis for the associated attribute table of n as in Case 1.

Case 6: A node n satisfied: $n \notin B$, $n \in A_1$, and $n \notin A_2$. That is, the node was inserted in only one branch. Then, the node and its attribute table will be added at version M.

Case 7: A node n satisfied: $n \notin B$, $n \notin A_1$, but $n \in A_2$. This is similar to Case 6.

Case 8: $n \notin B$, $n \notin A_1$, and $n \notin A_2$ (not applicable).

Similar scenario analyses are applied for every edge. This process will be repeated for all sub-graphs nested within a graph. When using this algorithm, a third-party Web environment will interpret those cases in accordance with its interpretation of nodes, edges, and attributes. Moreover, one could customize this algorithm by adding domain-specific knowledge into the merge process for better decisions. We have used this algorithm as a foundation to build merge tools for different types of Web objects in WebSCM.

8.2 Merging of Hypermedia Structure

The merge algorithm that was presented also handles merging of *embedded* hypermedia structures since the HTML hyperlinks are

represented by the "HREF" attribute values in attribute tables associated with nodes of HTML document trees. To deal with first-class hypermedia structures and connected Web objects is not much more complicated. The main reason is that the set of link/anchor nodes in a hypermedia network does not intersect with the set of graph nodes in Web objects (see Figure 6). Also, both of the hypermedia structure and the structure of related Web objects are represented as attributed graphs. Thus, the aforementioned merge algorithm for attributed graphs is still applicable with some modifications. The procedure for merging of versions of a hypermedia structure and connected Web objects is as follows.

First of all, we examine all *anchor* nodes in the hypermedia network. For each anchor node a, the corresponding Web object node, n_a, is uniquely determined via the "ref" attribute. Let us denote a Web object graph containing n_a by $G(n_a)$. If the versions A_1 and A_2 of $G(n_a)$ have not been merged, the graph-based merge algorithm is now applied to them. The same procedure is carried out for all anchor nodes. After this phase, all changes to connected Web object graphs have been merged into version M.

Next, we apply the graph-based merge algorithm to the hypermedia structure itself. However, an extra procedure is added into the algorithm in the cases in which an anchor node a might potentially be *inserted* into the merged version M (i.e. cases in which according to the merge algorithm described earlier, a will be added into M, such as cases 6 and 7). However, if the corresponding object node of a was not added into M, there is no need to have anchors defined on that node. Therefore, the extra procedure checks if n_a (i.e. the corresponding object node of a) did not appear in M according to the first phase, then a and connected edges of a will *not* be added into M. If the object node appears in M, then a will be also added into M. When *link* nodes are analyzed, the extra procedure is not used since they do not refer to object nodes.

9. IMPLEMENTATIONS

To verify our framework, we have implemented all aforementioned models and algorithms. We have been investigating existing Web and hypermedia design methodologies such as OOHDM [27], RMM [16], WebComposition [12], etc, and their supporting development environments. We have carefully examined different types of logical models and their Web objects. We have also looked at editing environments for design models as well as for different types of Web documents. Although very few of them have open sources, we have successfully used our framework to build bridges and added model-based, structure-oriented versioning supports into an object-oriented, UML-based environment, named *Thorn* [33], and a Scalable Vector Graphic (SVG) and animation editor, named *DrawSWF* [9]. For the Thorn environment, the bridges for different types of UML diagrams and schemas make use of the graph-based representation. The bridges create the connection between Thorn's internal representation and our graph-based representation. They make sure that changes to UML model-based entities and structures are properly reflected in the representation graphs.

In the experiment with DrawSWF editor, since it has XML as the internal representation, attributed trees are sufficient to model its supported Web artifacts. In fact, the bridge that we have built for this SVG editor is used for all XML-based and hierarchically structured Web artifacts that were supported in the previous version of WebSCM such as HTML, XHTML, and many other markup documents. The bridge for this type of Web objects has a library that is very similar to DOM [8]. However, those API functions know how to properly update attributed trees in Molhado. Java scripts are also supported with the use of a bridge that creates the connection between abstract syntax trees and attributed trees used in our framework. Structural differencing tools have also been constructed to display changes between two versions of an HTML, XML document, a Java script, and file directory structure.

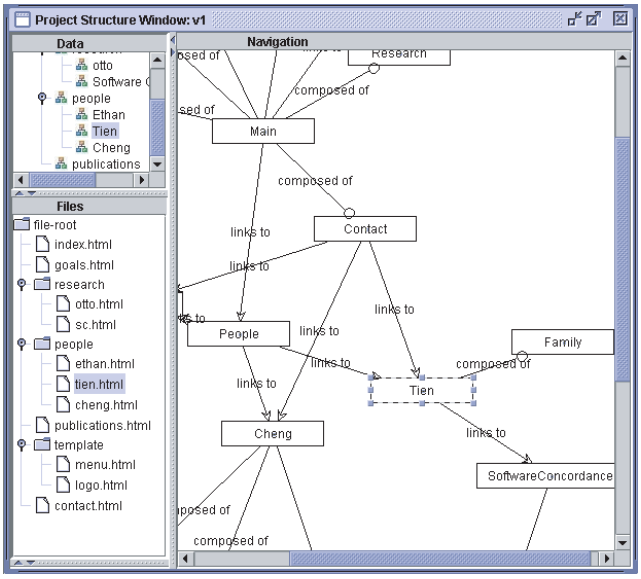


Figure 8: Model-to-model Logical Connections

To experiment with other types of Web objects in design models, we have been using WebSCM as an experimental platform. WebSCM has a pluggable architecture supporting the integration of new editors with any native internal representations. Different editors for Web objects were plugged into WebSCM. For example, we have built editors and provide structure-oriented version control and SCM supports for OOHDM conceptual schemas, navigation and composition schemas among *Web screens* (a similar concept to an ADV user-interface object in OOHDM [27]).

In WebSCM, we have used the versioned hypermedia infrastructure to manage *model-to-model* logical connections, dependencies, and mappings among Web objects in different logical models. For example, WebSCM is able to manage logical connections among Web screens, implementation files, and data records in a Web application. The right window in the Figure 8 shows the navigational and compositional designs for a Web site of a laboratory. The top left window displays data objects (e.g. lab staffs, students, projects, publications, etc) in a hierarchical view. The directory structure of the Web site at the implementation level is presented in the bottom left window. WebSCM is able to record the evolution of a mapping from a screen in the design model to actual files that realize the design of that screen (e.g. screen "Tien" and file "tien.html"). In addition, Web developers can manage the composition of screens and map them into HTML frames or pages. When users click on a composite screen, its member components will be presented (e.g. the "Main" page in Figure 8). The logical mapping from a data object to a Web screen in the screen design model is also captured over time. For example, when data object "Tien" is selected, the screen "Tien" is highlighted in the right window.

Figure 9 displays structural changes between two versions of a conceptual schema. Nature of changes is shown by attached icons. For example, between versions *v6* and *v7*, "CREDIT_CHARGE" was inserted (having an "i" icon), "ORDER_PICKUP_CLERK" was deleted (having an "X" icon), "SALE_DEPARTMENT" was modified (having a pencil icon), and the relation "charge" is newly added. The changes at the data level can be similarly displayed.

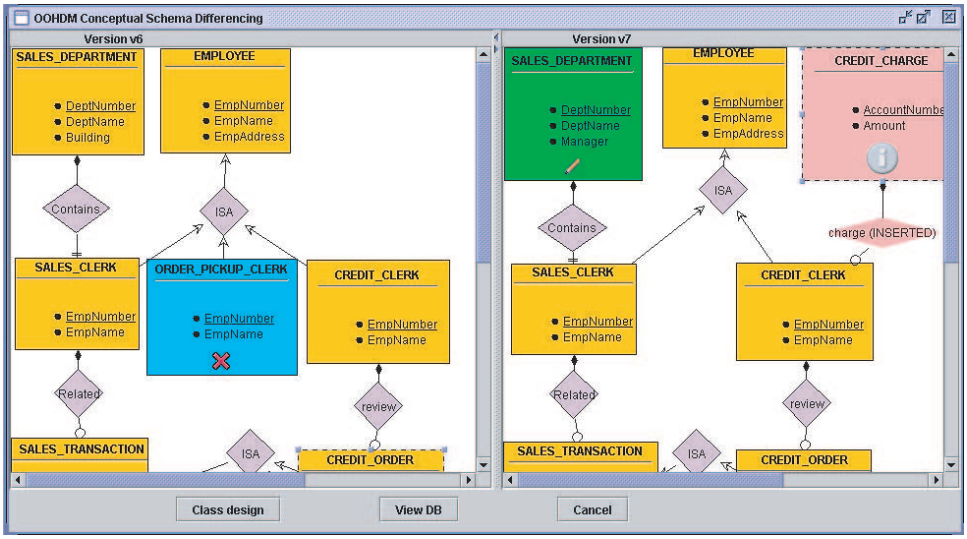


Figure 9: Structural Differencing of Versions of an OOHDM Conceptual Schema

10. RELATED WORK

Vendors in the SCM area are taking many approaches to Web configuration management. All have added Web functions to their SCM tools by offering access to some or all SCM functionality through a browser [6]. WebSynergy [36] provides a Web front-end into all of its existing SCM capabilities as well as Web authoring tools. MKS's WebIntegrity [35] integrates its version control facilities with an authoring tool, while in Merant's PVCS [20], version control is separated. However, both of them provide version control at the *file level*. StarTeam [30] is Web-enabled with the intention of tool integration. TrueChange [34] provides content change management along with its version control, but with less focus on *fine-grained* Web configuration management.

ClearQuest [4], a change request management tool of ClearCase, coordinates developers in editing Web documents. Content change management in SourceSafe [29] is line-oriented. Computer Associate's CCC/Harvest [3] pays considerable attention to supporting collaboration among distributed development teams. Perforce [25] has the ability to migrate repositories from other SCM tools. Although all of these commercial SCM tools have distinguished and valuable features, their main target is *files* that make up a Web application. None of them focuses on supporting *model-based* Web objects during the development process.

There are a number of advanced Web development methodologies and supporting Web engineering environments such as RMCase [7], WebComposition [12], Matilda [18], OOHDM-Web [26], HDM [13], etc. They introduce different design models and model-based Web objects but have not had a well-suited SCM tool for them. As in commercial Web site development environments (e.g. DreamWeaver [10], ColdFusion [5]), file-based SCM tools are often used. Other tools, such as TeamSite [32] and StoryServer [31], are designed to support many aspects of Web development, with particular strength in supporting collaboration. DynaBase [15] is an integrated content management and publishing platform for Web applications. It is XML-based and focuses on management and reuse of data. ArticleBase [1]'s versioning support is file-based.

Many researchers in *hypertext versioning* community [14, 24, 38] have focused on version control for documents in the presence of hyperlinks. However, the main goals of versioned hypermedia systems often do not include supports for Web application development. Therefore, supports for source code are very limited. Moreover, merging and differencing tools for hypermedia structures have not been addressed. To improve the authoring and browsing features for versioned contents of Web pages, some researchers in this area followed the language-oriented approach. They have attempted to change the Uniform Resource Locator (URL) of a Web page to include a version identifier [28]. Bendix and Vitali proposed VTML (Versioned Text Markup Language) [2] to express change operations for HTML documents. The WebDAV protocol [39] is an extension of the Hypertext Transfer Protocol (HTTP) to support distributed authoring and versioning. It extends HTTP to include versioning operations for Web pages. $\tau\tau$Apache is transaction-time HTTP server that supports document versioning [11]. To construct a document version history, snapshots of the documents files are obtained over time.

In the context of merging of versions of Web objects at the model level, previous software merging approaches are either too general or too specific, and fail to address the presence of hyperlinks among Web objects. Merging tools that are based on text files are not suitable for structured entities in design models since they disregard the underlying structures of objects [29]. To those tools, a line of text is considered as an indivisible unit. On the other hand, there are many approaches that are tailored toward specific languages [17, 37]. They allow *syntactic-oriented* or *semantic-oriented* merging with smarter decisions during the merge process. However, the knowledge about the language is encoded within the tools. Therefore, they could not be effectively used in a third-party Web development environment.

In general, existing version control and SCM systems for Web applications have a large variety of useful functionality. However, their approaches consider a file as an undividable unit for SCM. Their content change management is coarse-grained, with differencing done on a line-by-line basis. None of them has a representation model that adequately captures a wide variety of complex Web objects in hypermedia design models. On the other hand, other research approaches are too restricted to a particular type of Web content such as XML-based or hierarchically structured documents [17, 37]. Those versioning and merging approaches are not well-suited for complex graph-structured Web objects. In brief, existing Web configuration management approaches have not well

addressed the configuration management and version control for Web objects and their logical connections at the *model* level.

11. CONCLUSIONS

Systematic approaches to Web engineering become increasingly necessary as Web applications grow and have longer lifetimes. Advanced Web development methods put more emphasis on the separation between the high level of logical design models from the low level of implementation in Web documents. However, with existing Web version management tools, major Web design decisions are embedded within files in SCM repositories. The changes at the model level, and the logical connections among model-based objects and Web documents are not directly accessible. As a result, Web maintenance activities are error-prone and inefficient. Our framework presented in this paper addresses this problem.

Our framework allows developers to manage versions and configurations of design models and to capture changes to model-to-model logical relations among design objects. Model-based objects, Web documents, and logical connections among them are directly represented, persistently stored, and versioned in a structure-oriented and fine-grained manner. Our SCM tools with relationship management supports help Web developers to have better understanding about the evolution of their Web applications, and facilitate consistent management among models. Using our approach, a third-party Web engineering environment could provide structure-oriented SCM services for its supported Web objects at both design and implementation levels. Our future work includes conducting experimental studies with the emphasis on the automatic generation of bridges for different environments. Currently, space complexity is high relative to text-based SCM (three to six times), but we gain model-based, fine-grained SCM for a Web engineering lifecycle.

12. REFERENCES

[1] ArticleBase. http://www.runningstart.com/.

[2] L. Bendix and F. Vitali. VTML for Fine-grained Change tracking in Editing Structured Documents. In *Proceedings of the 9th International Workshop on Software Configuration Management*, pages 139-156. Springer Verlag, 1999.

[3] CCC/Harvest. http://www3.ca.com/.

[4] ClearQuest. www.rational.com/clearquest/index.jsp.

[5] ColdFusion. http://www.allaire.com/.

[6] S. Dart. *Configuration Management: the missing link in Web engineering*. Artech House, 2000.

[7] A. Diaz, T. Isakowitz, V. Maiora, G. Gilabert. RMC: A tool to design WWW applications. *The World Wide Web*, 1995.

[8] Document Object Model. http://www.w3.org/dom/.

[9] DrawSWF. drawswf.sourceforge.net.

[10] Macromedia DreamWeaver. http://www.dreamweaver.com/.

[11] C. Dyreson, H.-L. Lin, and Y. Wang. Managing Versions of Web Documents in a Transaction-time Web Server. In *Proceedings of the 13th International World Wide Web Conference (WWW 2004)*, pages 422-432. ACM Press, 2004.

[12] M. Gaedke and G. Graf. Development and Evolution of Web-applications Using the WebComposition Process Model. In *Proceedings of 2nd Web Engineering Workshop at the 9th International World Wide Web Conference*, 2000.

[13] F. Garzotto, P. Paolini, and D. Schwabe. HDM: A Model-based Approach to Hypermedia Application Design. *ACM Transactions on Information Systems*, 11(1):1-26, Jan 1993.

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

[15] DynaBase content management. http://www.rbii.com/products/dynabase/.

[16] T. Isakowitz, E. Stohr, and P. Balasubramanian. RMM: A Methodology for Structured Hypermedia Design. *Communications of the ACM*, 38(8):34–44, 1995.

[17] T. Lindholm. A three-way merge for XML documents. In *Proceedings of the 2004 ACM Symposium on Document Engineering*, pages 1–10. ACM Press, 2004.

[18] D. Lowe, A. Ginige, M. Sifer, and J. Potter. The Matilda data model and its implications. In *Proceedings of 3rd International Conference on Multimedia Modeling*, 1996.

[19] Luqi. A Graph Model for Software Evolution. *IEEE Transactions on Software Engineering*, 16(8):917–927, 1990.

[20] PVCS. http://www.merant.com/.

[21] S. Murugesan, Y. Deshpande, S. Hansen, and A. Ginige. Web Engineering: A new discipline for Web-Based System Development. In *Web Engineering: Managing Diversity and Complexity of Web Application Development (LNCS 2016)*, pages 3–13. Springer Verlag, 2001.

[22] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An Infrastructure for Development of Object-Oriented Configuration Management Services. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 215–224. ACM Press, 2005.

[23] T. N. Nguyen, E. V. Munson, and C. Thao. Fine-grained, structured configuration management for Web projects. In *Proceedings of the 13th International World Wide Web Conference (WWW 2004)*, pages 433–443. ACM Press, 2004.

[24] K. Østerbye. Structural and cognitive problems in providing version control for hypertext. In *Proceedings of the ACM Conference on Hypertext*, pages 33–42. ACM Press, 1992.

[25] Perforce. http://www.perforce.com/.

[26] D. Schwabe and R. de Almeida Pontes. A Method-Based Web Application Development Environment. In *Proceedings of the 1st Web Engineering Workshop at the 8th International World Wide Web Conference*, 1999.

[27] D. Schwabe and G. Rossi. An Object Oriented Approach to Web-based Application Design. *Theory and Practice of Object Systems*, 4(4):207–225, 1998.

[28] J. Simonson, D. Berleant, X. Zhang, M. Xie, and H. Vo. Version augmented URIs for reference permanence via an Apache module design. In *Proceedings of the WWW7 Conference, Computer Networks and ISDN Systems*, 1998.

[29] Microsoft Visual SourceSafe. http://msdn.microsoft.com/ssafe/prodinfo/overview.asp.

[30] StarTeam. http://www.startbase.com/.

[31] StoryServer. http://www.vignette.com/.

[32] TeamSite. http://www.interwoven.com/.

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

[34] TrueChange. http://www.truesoft.com/.

[35] WebIntegrity. http://www.mks.com/.

[36] WebSynergy. http://www.continuus.com/.

[37] W. Wei, M. Liu, and S. Li. Merging of XML Documents. In *Proceedings of the ER'04 Conference*. Springer Verlag, 2004.

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

[39] E. J. Whitehead, Jr. WebDAV and DeltaV: collaborative authoring, versioning, and configuration management for the Web. In *Proceedings of the ACM Conference on Hypertext and Hypermedia*, pages 259–260. ACM Press, 2001.

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