An Infrastructure for Development of Object-Oriented, Multi-level Configuration Management Services

Tien N. Nguyen
Computer Science Dept.
Univ. of Wisconsin-Milwaukee
tien@cs.uwm.edu

Ethan V. Munson
Computer Science Dept.
Univ. of Wisconsin-Milwaukee
munson@cs.uwm.edu

John T. Boyland
Computer Science Dept.
Univ. of Wisconsin-Milwaukee
boyland@cs.uwm.edu

ABSTRACT

In an integrated development environment, the ability to manage the evolution of a software system in terms of logical abstractions, compositions, and their interrelations is crucial to successful software development. This paper presents a novel framework and infrastructure, *Molhado*, upon which to build *object-oriented* software configuration management (SCM) services in a SCM-centered integrated development environment. Key contributions of this paper include a *product versioning* model, an *extensible, logical,* and *object-oriented system model*, and a *reusable product versioning SCM infrastructure*, that allow new types of objects to be implemented as extensions of the system model's basic entities. Versions and configurations of objects are managed at different levels of abstraction and granularity. A new SCM-centered editing environment or development environment for a specific development paradigm can be rapidly realized by re-using Molhado's infrastructure and implementing new object types and their associated tools. This paper also demonstrates our approach in creating prototypes of SCM-centered development environments for different paradigms.

Categories and Subject Descriptors

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

General Terms

Management

Keywords

Software Configuration Management, Version Control

1. INTRODUCTION

A variety of software development methodologies and frameworks have been introduced to provide developers with systematic approaches for producing high-quality software. These development methods often introduce *abstractions* and impose different *logical structures* among them, permitting one to concentrate on problems at some level of generalization without regard to irrelevant low-level details. For example, in object-oriented frameworks, abstractions typically are classes and objects, and examples of logical structures include class inheritance hierarchy, control hierarchy, etc. In the relational database framework, tables are related via relations to form an entity relationship diagram.

Software development is a dynamic process where software engineers constantly modify and refine their systems. As a consequence, everything evolves. Unlike source code, for which the use of a software configuration management (SCM) system is the predominant approach to capturing evolution, approaches to managing evolution of high-level abstractions span a wide range of disconnected alternatives [26]. Many *file-oriented* SCM systems treat a software system as a "set of files" in directories on a file system, and stable configurations are defined implicitly as sets of file versions with a certain label or tag. Following a development method, developers usually think and reason in terms of high-level abstractions, compositions and their interrelations. This creates an impedance mismatch between the design and implementation domain (logical level) and the configuration management domain (file level) [6]. SCM systems, whose concepts are heavily based on storage structure can become burdensome for ordinary developers partly because design/implementation methods and SCM infrastructures require different mental models [13].

To bridge that mental gap between software development methods and SCM, it is necessary to have SCM systems that allow users to manage the evolution of a software system in terms of logical abstractions and relationships without worrying about the concrete level of actual file storing and versioning. In this paper, these systems are referred to as *object-oriented* SCM systems. However, despite many successes, most SCM systems are either specifically designed and too restrictive to a certain development framework or domain, or too generic and do not provide sufficient SCM supports at the logical level. For example, while architectural SCM systems support a fixed set of architectural objects, some text file-oriented SCM systems depend heavily on a line-oriented model of internal changes that totally disregards logical structures of contents. Unfortunately, building object-oriented SCM systems from scratch for a new development framework is a difficult endeavor that regularly requires a lot of time and efforts from developers.

To reduce this effort, several infrastructures [24, 27, 29, 32] have been developed which support reuse of a generic SCM model (i.e. repositories and data structures used to capture the evolution of artifacts), and/or critical parts of SCM policies (i.e. specific procedures for creating, evolving, and assembling versions of artifacts stored in SCM repositories). However, their main goal is to provide a flexible and pluggable architecture for developers to compose a wide range

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

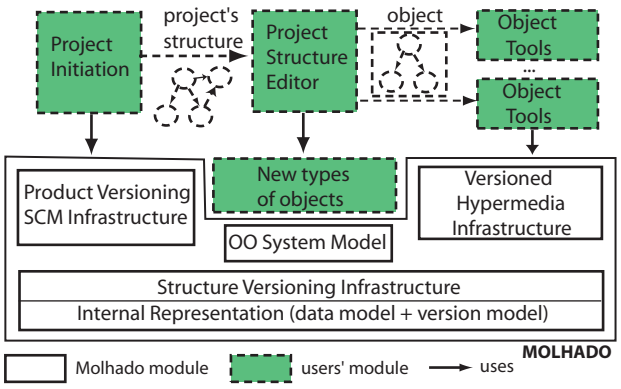


Figure 1: Molhado architecture

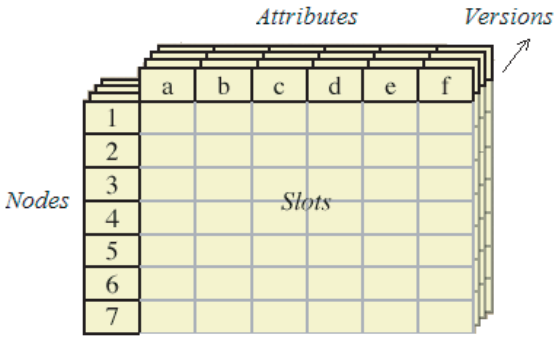


Figure 2: Data model

of SCM systems from a base set of provided modules and developers' code. Since they are not specialized toward object-oriented SCM systems, when using them, developers have to pay a lot of attention to detailed and technical points of many SCM aspects. Additionally, the complex implementations of mappings from versions of logical objects to physical files can easily distract them from focusing on modeling a system in terms of new objects and structures in a new development paradigm. Therefore, developers' effectiveness is reduced. In brief, those generic infrastructures are not well-suited for building this type of SCM systems.

2. MOLHADO APPROACH

In this paper, we describe *Molhado*, a novel SCM framework and infrastructure, that was specifically designed to support the development of object-oriented, multi-level SCM systems. Each of these SCM systems requires the use of specialized tools such as editors for the objects that are supported. To address this problem, Molhado also provides a simple pluggable architecture that enables the integration of a resulting SCM system and specialized tools (see Figure 1). The result of this integration is an *SCM-centered* development or editing environment, in which the SCM system is the heart of the environment and every changes to logical objects and structures during the development process will be recorded. By maximizing the reuse of SCM infrastructures via built-in modules, Molhado can minimize developers' effort in building versioning and SCM services for objects in these environments.

The key point of Molhado is its *object-oriented* approach to reuse in object-oriented configuration management systems. An *extensible, logical,* and *object-oriented system model* provides a framework for developers to model a software system (according to any particular software development paradigm) in terms of logical objects and their relations at different levels of abstraction and granularity (Section 5). With a provided editor, developers can define new types of objects as normal Java classes inherited from basic entities in Molhado's system model, and code to handle versioning and storing for objects will be automatically inserted (Section 6). Entities in this system model and new object types are defined based on a representation model in which a version model is built into a primitive data model (Section 3). This combination approach facilitates version control for logical objects since their internal properties and logical structures are visible to the version model. To support versioning for structured objects, a structure versioning infrastructure including fine-grained version control algorithms for trees and directed graphs has been developed on top of the representation model (Section 4).

Another important module is the *product versioning* SCM infrastructure including SCM policy and transaction supports (Section 7). In the product versioning approach, a *version* is global across entire project, rather than being a particular state of an artifact. That is, all system objects are versioned in a global version space of a software project. This product versioning choice enables a mechanism for managing configurations among any objects of new types with a minimum requirement of code from developers. The system model is also enhanced by a *versioned hypermedia* infrastructure, which manages the evolution of fine-grained links among objects separately from object contents.

To produce a SCM-centered environment, developers just need to implement new object types and associated tools, while reusing Molhado's modules. Details of users' modules will be described in Section 8. To demonstrate the use of Molhado, we have built several prototypes of object-oriented SCM systems and associated SCM-centered development environments, accommodating different development frameworks (Section 8). Although none of these systems are fully functional as real-world development environments, they have several advantages over the existing systems in the same domains. More importantly, they illustrate the key benefit of Molhado: the reduction in developers' effort.

3. DATA MODEL AND VERSION MODEL

3.1 Primitive Data Model

To be able to expose internal properties of a logical object to a version model, Molhado follows a combination model between a version model and a primitive data model, called Fluid Internal Representation (IR) [5]. The main concepts in this data model are *node*, *slot*, *attribute*, and *sequence*. A node is the basic unit of *identity* and is used to represent any abstraction. A slot is a location that can store a value in any data type, possibly a reference to a node. A slot can exist in isolation but 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. The primitive data model can thus be regarded as an attribute table whose rows correspond to nodes and columns correspond to attributes. The cells of the attribute table are slots (see Figure 2).

There are three kinds of slots. A *constant slot* is immutable; such a slot 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 different values in different versions (*slot revisions*). A *sequence* is a container with slots of the same data type. It has a unique identifier. Sequences may be fixed or variable in size and share common slots together. Slots in

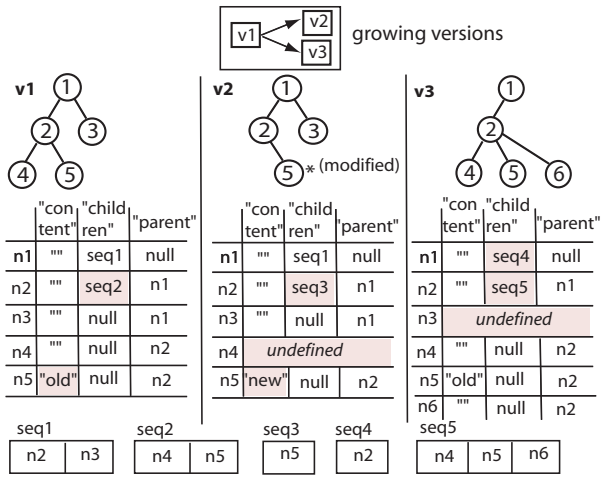


Figure 3: Structure Versioning

an attribute table can also be simple slots or constant slots. Once we add versioning, the table gets a third dimension: the version (see Figure 2).

3.2 Product Versioning Model

A *version* in our model is *global* across entire software project 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. All system objects are versioned in a uniform, global version space across a software project, called *product versioning space*. Molhado emphasizes the evolution of a software system as a whole. In our version model, the state of the whole software system is captured at certain discrete time points and only these captured versions can be retrieved in later sessions. The *current version* is the version designating the current state of the project. When a particular version is set to be current, the state of the whole project is set back to that version. Changes made to versioned slots of the project at the current version create a temporary version, *branching off* the current version. That temporary version will only be recorded if a user explicitly requests that it be captured. On the other hand, no version is created if a simple slot is modified. To record the history of an individual object, the whole project is captured. Capturing the whole project is quite efficient because Fluid IR only records changes and works at the slot level [5]. It uses techniques derived from the work of Driscoll [10] to store and retrieve different data types in attribute tables. It also uses a generic delta algorithm for versioned slots in different primitive data types.

4. FINE-GRAINED STRUCTURE VERSIONING

From the primitive data model, trees and directed graphs are built. A tree is defined with two main attributes: 1) the "children" attribute maps each node to a sequence holding its children, and 2) the "parent" attribute maps each node to its parent. Figure 3 illustrates our tree-based structure versioning algorithm via an example. In this example, a "content" attribute is also defined that holds a string value for some of the nodes. Assume that there are three versions: *v1*, *v2*, and *v3*. Versions *v2* and *v3* branch off from version *v1*. The shape of a tree at each of the three versions is shown. Version *v2* has two differences from the version *v1*: node 4 was deleted and the content of node 5 was changed. Version *v3* has

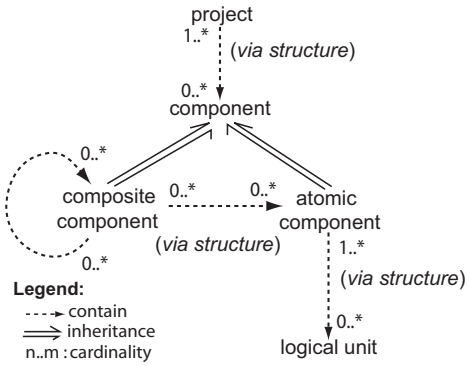


Figure 4: Object-Oriented System Model

an inserted node (node 6) and node 3 was deleted. The values of versioned slots in the attribute table are changed to reflect modifications to the tree at these versions. For example, at the version *v2*, the "content" slot (i.e. the slot defined by the attribute "content") of node 5 contains a new value (the string "new"), and the "children" slot of node 2 contains a reference to a new sequence object (*seq3*). *Seq3* has only one slot, which contains a reference to node 5 since node 4 has been deleted. If there is a request for the values of slots associated with node 4 at *v2*, an error will be reported. The fine-grained versioning algorithm for a directed graph is similar except that the attribute table does not have the "parent" attribute.

This structure versioning infrastructure is powerful and flexible since a directed graph can be used to represent various forms of structures. It is also very efficient since common structures are shared among versions and all information including structures and contents are versioned via one mechanism. Importantly, the algorithm is general for any subtree or subgraph, therefore, fine-grained version control is achieved for any abstraction represented by a node, such as any syntactical unit in a program.

5. OBJECT-ORIENTED SYSTEM MODEL

System modeling is the activity of describing the structure of a system in terms of its components and the relationships among them. With the purpose of providing an SCM infrastructure to manage a software system in terms of logical objects, we need to have a system model that works at the logical level and is capable of supporting any new types of objects. Here, the object-oriented approach comes into play. Molhado's object-oriented system model with its basic entities, constructed via the data model, provide both infrastructure for extensibility and a framework to version and to model a system in accordance with any development paradigm.

Figure 4 summarizes entities in the system model. A *project* is a named entity that represents the *overall system logical structure* of a software system (also called "software project" in this paper). Depending on the software development framework chosen, a project will represent different forms of system structures such as architectural design diagrams, UML design diagrams, entity relationship (ER) diagrams, or control hierarchies. In software systems with simple designs, a project can directly represent the source code structures at the implementation level such as class/package or program/directory structures. Technically, a project contains a structure that is composed of *components*. Developers can use that structure to represent various forms of overall system structure in a software project. That structure is implemented in the form of a directed graph. In a system with a hierarchical overall structure, using that directed graph as a tree would be sufficient.

A *component* is basically an entity that represents a *logical object* in a software system. It can be versioned, saved, loaded, and exists within the version space of a software system. A component is designed to represent a *coarse-grained* abstraction and can belong to multiple projects. It is *not* designed to model fine-grained objects or abstractions such as a character or a word, though it is not prohibited to do so. A component can be used to model logical abstractions at the design level such as design modules, architectural components, UML diagram entities, data flow diagram entities, ER diagram entities, subsystems, etc. At the implementation level, components can model programs, object-oriented classes, functions, packages, files, directories, documentation, etc. A component is implemented based on the primitive data model. Each component carries a component identifier that serves to identify it uniquely within a project. A user-assigned name for a component, which might change at different versions, is implemented as a versioned slot. A component also sets up object loading and saving functions in derived types of objects in accordance with our persistence model. Components are classified into two groups: *composite* and *atomic* components.

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. Examples include compound documents, Java packages, architectural composite components, system logical structures, class hierarchy, UML diagrams, etc. Each of those objects and structures can be defined as an extended type of this composite component type (see Section 6).

Unlike composite components, an *atomic component* can *not* contain other components. Therefore, it is the basic entity of the composition or aggregation in a composite component. However, it *may* have internal structure. In Molhado, to increase modularity and accommodate *fine-grained* version control for components, an atomic component can be internally composed of finer units, called *logical units* or *structural units*. For example, a class can be defined as an atomic component consisting of syntactical units (as logical units) in an abstract syntax tree (AST). In general, internal structure of an atomic component might also be more complex than a tree-based hierarchy, for example, a graph-based structure in a graphical document. In contrast, some atomic components, such as binary data files or images, are considered to have no internal structure for the versioning purpose. Molhado treats a logical unit as the smallest versionable unit. A logical unit is implemented as a node in the primitive data model.

The key point that distinguishes this system model from others is its logical and object-oriented approach that enables the extensibility of new types of objects and allows them to be versioned at the logical level and *independently* of the *physical* storage structure. The logical model level is detached from the physical file level, allowing developers to focus on modeling a software system in accordance with a development paradigm without worrying about actual file storing. Our system model is not tied to any particular domain and its entities do not contain version numbers since all objects share a global version space. Therefore, this enables the reuse of our product versioning SCM infrastructure (see Section 7). The idea of explicitly versioning at *different levels of abstraction* is novel and also realized via this object-oriented approach.

The distinction between component and logical unit levels creates modeling *flexibility* and increases the *modularity* for our system model. Depending on the development method chosen, an object can be implemented as a derivation of either a component or a logical unit. For example, a Java class can be atomic in a class hierarchy (as a composite component), but it can also be modeled

```
import molhado.project.*;
...
class JavaClass extends AtomicComponent {
...
 static final int magic = 0x4A415641;//'JAVA'
 private Slot name = VersionSlotFactory...;
 SyntaxTree java_ast = ...;
 private JavaParser parser = ...;
 public JavaClass() {super(magic);}
 public JavaClass(UniqueID id){
  super(magic,id);
 }
 public void writeContents(IROutput out){...}
 public void readContents(IRInput in){...}
 public void writeChangedContents(...){...}
 public void readChangedContents(...){...}
...
 public void loadDelta(...){...}
 public void saveDelta(...){...}
 public void loadSnapshot(...){...}
 public void saveSnapshot(...){...}
...
 private static class Factory extends
                      ComponentFactory {
   ...
 }
  // Import/Export functions
  // AST manipulating functions
  // Parser functions
...
}
```

Figure 5: Java Class Component

as a logical unit in a program (as an atomic component), which might contain other classes. In addition, modeling a software system in two different levels of composition (coarse-grained: atomic components being part of composite components, and fine-grained: logical units being part of atomic components) would make the system more manageable and modular to users. Remind that in Molhado, from users' point of view, a software system evolves over time as a whole. Next, we describe how to define object types.

6. OBJECT VERSIONING FRAMEWORK

6.1 Atomic components

A simple editor is built to help developers define new types of objects. Template code for objects is automatically inserted. To define a new atomic component, a developer needs to derive his/her Java class from the atomic component type and declare a 32-bit constant magic number that uniquely identifies the new object type. Only concrete classes need the numbers. A component's internal property whose history needs to be recorded must be represented by a versioned slot. For example, the component name can be declared as a versioned slot containing a "string" value. Other properties that need to be stored must be defined as simple or constant slots. Otherwise, regular Java fields with standard types can be used. For each slot that was used in a new class, Molhado generates methods to read and write slot contents.

In Molhado, an atomic component can be internally composed of logical units. The developer can manage its internal structure by declaring a tree or a directed graph (described earlier) in a new class. The developer can use the associated programmatic interface for trees and graphs to manipulate the component's internal structure. For each declaration of a tree or a graph, Molhado inserts code for loading and saving nodes and associated slots in the tree or the graph. If a new atomic component has *no* internal structure, Molhado's trees or graphs must not be used. The developer can use trees or graphs to represent various forms of structures in an atomic

component such as AST in a program or compositional hierarchy in a structured document. For a new concrete (non-abstract) class, Molhado also generates a "factory" inner class to facilitate the Java dynamic loading mechanism that is used to load a class. The factory class name is saved and used when the class is loaded.

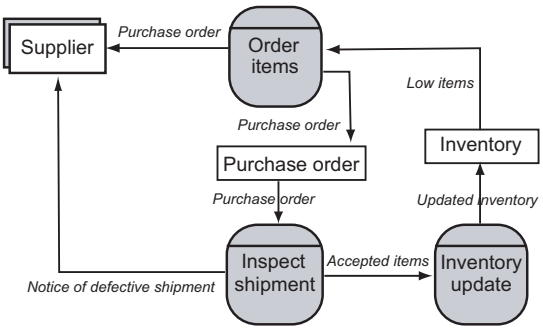
Figure 5 illustrates the definition of the class "JavaClass" for a Java class component. This new type is derived from the atomic component type. Since "JavaClass" is non-abstract, a magic number is chosen. Since the name of the Java class is declared as a versioned slot, methods handling writing and reading values for the name are automatically inserted. Similarly for the syntax tree that was declared for the AST of the class, loading and saving functions ("loadDelta", "saveDelta",...) are generated. To import a Java file, this component needs a parser, which is an external tool that does not need to be versioned. So, it is declared with a regular Java type. The templates for two constructors are created and developers are free to add more code. Technically, one constructor is used when a new object is created, and the other is used when an object needs to be loaded from disks. Finally, since "JavaClass" is a concrete class, an inner class "Factory" is inserted. Other methods and fields can be added for other purposes. The internal structure of a Java class component defined by this will be versioned via the tree-based fine-grained versioning algorithm (see Section 8).

6.2 Composite components

The principles for the definition of new composite components are similar to those for atomic components. The most important difference is that a new composite object type inherits from the composite component type (in the system model) a special tree or directed graph in which an additional attribute ("*component*") defines for each node in the tree or the graph a versioned slot holding a reference to a member component of the composite component. Other principles on generated methods and fields still hold in this case. Figure 6 shows the top level of a Data Flow Diagram (DFD), which is implemented as a composite component. Processes in the DFD might be composite objects. The structure of the DFD is represented by a directed graph, whose attribute table is partially shown. The "component" slots associated with "n1", "n3", and "n5" refer to the corresponding components, while nodes "n2" and "n4" represent outputs and inputs (i.e. edges) in the DFD.

Via product versioning, Molhado retrieves versions of composite objects without using version selection rules or tagging version numbers. The current version of the project is globally selected. The properties (as versioned slots) and the internal structure of a composite component will be correctly retrieved via the fine-grained versioning algorithm. The "component" versioned slots of nodes in the tree or the graph will refer to proper member components of the composite component at the current version as well. Then, the same process is applied for each member component. The process stops either at atomic components that have no internal structure or at logical units of some atomic components.

With this representation, any kind of complex composite objects with nested levels can be implemented. Also, both *logical* relations and compositional hierarchies among components can be modeled. For example, a composite component can be created to represent a class hierarchy or a package containing other packages and classes. In a "class hierarchy" composite, "class" components are related to each other via inheritance relations. Additionally, via directed graphs, any structure among direct sub-components of a composite can be modeled, e.g. connections among "processes" in the DFD in Figure 6. This framework and tools facilitate the definition of any (structured) object types and allow objects to be versioned at both fine (i.e. logical unit level) and coarse (i.e. component level) granularities in a uniform manner. The framework is not only applicable to design artifacts but also to source code (see Section 8).



	"component"	"children"	"type"	"description" ···
n1	Supplier	*null*	External Entity	···
n2	*undefined*	[n1]	Output	"Purchase order"
n3	Order_items	[n2,n6]	Process	···
n4	*undefined*	[n1]	Output	"Notice of def..."
n5	Inspect_ship..	[n4,n8]	Process	···
···	···	···	···	

Figure 6: Representation for a Data Flow Diagram

6.3 Versioned Hypermedia

With our infrastructure so far, the relationships among objects are encoded *within* an atomic or composite component. This embedding approach is *not* suitable for management tasks of logical relationships or fine-grained traceability links that are *un-structured* and *span* multiple components. For example, the interdependency among a requirement item, a design choice, and a function in source code evolves over time and involves different components.

To address this issue, the *Molhado versioned hypermedia* model was implemented [17]. The principle concepts of that model are: *linkbase, hypertext network, link*, and *anchor*. A linkbase is a container for hypertext networks and/or other linkbases and is implemented as an extension of a composite component. A hypertext network, which represents for a logical relationship structure or a traceability link network, contains links and anchors. A link, representing for a logical relationship, is an association among a set of anchors. An anchor is a *mark* used to denote the region of interest within a component, and it *refers* to a logical unit in a component. Links and anchors can be associated with any attribute-value pairs. Technically, a hypertext network is defined as a directed graph-based atomic component. Each link or anchor is represented by a node in the graph. A directed edge will connect an anchor's node to a link's node if the link contains the anchor. An additional attribute ("ref" attribute) is defined to associate an extra slot to each anchor's node in the graph. The slot holds a reference to either a logical unit within a component or a component (but *not* to a hypertext network or a linkbase). That logical unit (or that component) is considered as the position of the anchor. Basically, since hypertext networks are represented as directed graphs with extra attributes, logical relationship structures or traceability link networks are versioned via the directed graph-based versioning mechanism. Changes made to an individual link are recorded as well since each link is represented as a node (i.e. logical unit) in a directed graph [17].

Molhado's versioned hypermedia infrastructure has several advantages over existing versioned hypermedia tools in managing the evolution of component relationships [17]. For example, hyperlinks are first-class entities, facilitating systematic analyses and vi-

sualization of logical relationships. Hyperlinks are multiheaded (n-ary) and connect fine-grained fragments of any components. The separation between anchors and logical units implies the separation of the hypermedia structures from component's contents, allowing for multiple structures on the same set of documents, without modifying their contents. With fine-grained versioning, Molhado manages not only the evolution of traceability link networks (as graph-based components) but also of individual links (as logical units). In addition, traceability links are maintained among *logical* objects, rather than among *physical* software documents as in traditional traceability tools. The process of software tracing is improved since our model allows developers to work at the logical level.

7. CONFIGURATION MANAGEMENT INFRASTRUCTURE

Section 6 described how configurations among objects belonging to a composite can be managed. However, a configuration management infrastructure is required to maintain configurations among *all* system objects in a software project. This section describes such infrastructure and transaction supports for committing changes.

7.1 Product versioning SCM infrastructure

Unlike many existing systems, Molhado provides a SCM infrastructure, called *product versioning SCM*. This approach addresses the configuration management among objects and the version control for a software system simultaneously. Molhado manages the evolution of a system and its software components as a whole via the *project* entity in the system model. The project entity, representing the overall structure of a software system, contains a directed graph in which for a given graph node, the associated "component" slot contains a reference to an atomic or composite component of a software system. Developers must use the *directed graph* in the project entity to represent various forms of an overall system logical structure as mentioned in Section 5.

Notice that modeling for a project is similar to a composite component. However, a project can *not* be used as a composite component in any compositional hierarchy. Instead, it must be regarded as the outermost composite of all components and sets up the context for the global version space. That is, the scope of the current version is at the project level. This product versioning SCM, which versions a project as a whole entity, always assures the construction of a consistent configuration among all objects in the project. The reason is that when a project version is chosen as current, the project's directed graph will be correctly retrieved and versioned slots associated with nodes in the graph will refer to proper components at the current version as well. Then, the internal properties and structure of each component are determined as described.

The most significant benefit of Molhado's product versioning SCM approach is that developers can *reuse* this infrastructure in managing the configurations among defined objects (even of any *new* types) in a structure or in entire project. For example, to construct an architectural SCM system, developers define architectural element types such as architectural components, connectors, etc. They can use provided library functions to write code to *manipulate* the architecture. However, they do *not* need to write any code for *managing versions* and *configurations* among architectural elements. The rationale behind this advantage is the ability of our product versioning SCM infrastructure to relate together *versioned slots* of the *same version* of a project (i.e. a configuration). Therefore, it is able to retrieve and relate objects in the same configuration or the same version of a structure together. Note that in this product versioning approach, versions of objects (including properties and structures) are *implicitly* determined by the versions of the whole project. Additionally, in a resulting product versioning SCM system, configurations are maintained among *logical objects* (rather than among *physical files*) at all levels of granularity, relating changes together in a *uniform* manner.

7.2 SCM transaction supports

Molhado's transaction support infrastructures are provided in the form of library functions. They include opening and creating a project, retrieving and modifying meta-data of a project (name, authors, version history, etc), setting the current version (i.e. *switching* a version), manipulating a project's directed graph, discarding, capturing, or saving changes to a project at a version, retrieving *structural* differences between two arbitrary versions of system structures, verifying whether a versioned slot or a set of versioned slots (including "children" and "parent" slots) associated with a node have changed between two versions, etc. The operational model of Molhado SCM is as follows. A user creates a software project or opens an existing project. The version history of the project is displayed. After selecting the current (working) version, the overall system structure at the current version is loaded. The user can modify this structure or select a component to edit. An appropriate component editor will be invoked.

If any modification is made to the versioned objects of the project at the current version, a new version would be temporarily created, branching off the current version. Subsequent modifications will not change the temporary status of the version until a *capture* or a *commit* command is issued. The user can choose to discard any temporary version, or to *capture* the state of the project at a version. *Capture* command changes a temporary version into a captured one. A unique name as well as date, authors, and descriptions can be attached to the newly captured version for later retrieval. The captured version plays the role of a checkpoint that the user can retrieve and refer to. While working on one version, the user can always *switch* to work on (view or modify) any other version. Switching the current version can be done implicitly via the user's mouse focus (if a GUI is provided) or explicitly via the user's assigning a new current version. If the user modified overall structure of the project or its versioned components at the new working version, an additional derived version will be branched off that new version. The old version (even an uncaptured one) is still available should the user want to do additional work on that version. This switching feature allows the user to work on many versions at the same time during one session (multi-version editing).

The user may *commit* changes at any time. Upon issuing this command, the user is asked which *uncaptured, temporary versions* should be saved and the chosen versions are then saved along with any already captured versions. Only the differences are stored. The user may also save complete version snapshots, which can improve version access time. This differs from the capture command, in which no data is saved to disk after a capture. Furthermore, while the capture command records changes to the current version (i.e. it works at a branch), the commit command takes care of modifications to versions at all branches in entire version history. In the current infrastructure, each user has his own data files for the project and they can be stored anywhere in a file system. Users do not see changes made by others. Therefore, no locking mechanism is needed. Users can share data files. A functional-level merging tool is being developed to help them in collaboration.

8. EXPERIMENTS

Based on Molhado framework, we have built a library of commonly used object types and associated editors including Java pro-

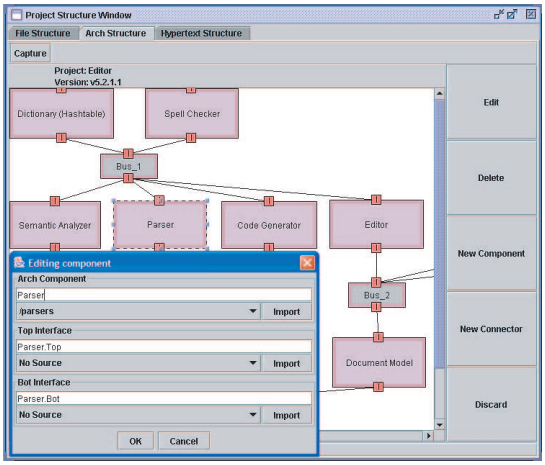


Figure 7: Architectural SCM

grams with embedded multimedia documentation, Java classes, documentation in XML, HTML, Scalable Vector Graphics (SVG). Image, audio, and binary files are considered to have no internal structure for the versioning purpose. Facilities are provided to import and export components (stored in our binary format) from and to corresponding external formats at any version.

To build a new SCM-centered environment, a developer needs to implement the set of new object types and associated tools (see Figure 1). In general, developers are free to implement object tools. Here are a few requirements on interactions and responsibilities of users' modules. For setup tasks, the "project initiation" module is needed. That module is mainly responsible for creating or loading the selected project, retrieving and displaying information of the project (names, date of creation, authors, version history, etc), setting the current version from users' selection, and passing the overall system structure at the current version to the "project structure editor" module. This module must provide an editor for the system overall logical structure at the selected version. It must invoke appropriate object tools and pass to them the user-selected objects. Any new object tools such as editors or analyzers can be implemented and plugged into the architecture [18]. However, it requires to have, at a minimum, *editors*, providing editing services for selected objects at the current version. Project structure and object editors can use Molhado's functions to provide fine-grained versioning for logical units, or to retrieve structural changes between versions. They must be able to call Molhado's functions to manipulate structures and objects. Each of those editors also needs to implement the "capture" functionality for the version that it is displaying. When users switch to work on an editor, the displayed version of the editor must be set as the current version.

To demonstrate how Molhado promotes reuse in building object-oriented SCM environments, we built prototypes of object-oriented, multi-level SCM systems and associated SCM-centered editing environments for different development paradigms. While none of them is as fully functional as real-world environments (only component editors are provided and other specialized tools are omitted), all of the systems are functional and allow users to manage the evolution of a system in terms of logical objects and structures during editing sessions. An simple GUI-based "project initiation" module is also implemented and reused among prototypes. These environments share Molhado's infrastructure, the "project initiation" module, and the library of common object types and associated editors. Next, we describe these prototypes and analyze research results.

8.1 SCM-centered development environments

Architectural SCM: Our first prototype is an C2-style *architectural* SCM system and its associated architecture-based development environment, *MolhadoArch* [16]. To version system architecture, five additional types of components are introduced: 1) architectural component, 2) architectural atomic component, 3) architectural composite component, 4) architectural connector, and 5) architectural interface. The architectural structure of a system is modeled by the *project* entity containing a directed graph. Details of the representation of a system architecture and new types of objects can be found in another paper [16].

Molhado's approach for managing architectural evolution has distinguished characteristics over existing SCM approaches for architecture [16]. First, due to product versioning, architectural evolution is captured together with implementations in a natural and cohesive manner. That is, consistent configurations are maintained not only among source code but also with the high-level software architecture. Second, since the system is able to import/export system architectures from/to *xADL 2.0* [8], an XML-based architecture description language (ADL), it supports both *planned* (via xADL constructs) and *unplanned architectural evolution* (via Molhado). Note that existing ADLs deal only with variants and options in architecture. That is, all variants must be represented in an architecture description, so that users can choose among them. Third, Molhado precisely captures how the architectural elements are connected and interact with each other via the the explicit and separate treatment of architectural components and connectors. Finally, since Molhado's system model is *extensible*, new components can always be defined to accommodate different ADLs and architectural styles. Figure 7 shows a snapshot of the project structure (architecture) editor in MolhadoArch. To specify the implementations of architectural elements, a user can create new source code via built-in editors or importing external programs.

Web engineering development: Our second implementation is a SCM-centered Web engineering environment, *WebSCM* [18]. Many types of Web contents and associated editors were created to support Web application development. The resulting environment distinguishes itself from existing ones by its ability to manage the evolution of a Web-based application including its Web contents and important structures such as *navigational*, *compositional*, *internal*, and *logical* structures. Navigational structure of a Web system refers to the structure of Web documents with respect to hyperlinks among them. Compositional structure refers to the composition of pages or frames from pieces of data or texts. Internal structure of a Web document is often hierarchical. The *logical* and *physically-independent* structure among Web documents allows access to Web content without having to know the actual physical address of the files. This ability is very important for large-scale Web applications whose documents may be globally distributed.

The most important benefit Molhado brings into WebSCM is the ability to manage the versions and configurations among design artifacts at multiple levels (concept, navigation, and presentation levels), and source artifacts at the implementation level in a Web project (e.g. HTML documents or scripts). The mapping and relations between objects at those levels are also recorded over time. In this experiment, the two-level composition and version control in Molhado's system model has made the modeling of a complex Web-based system easier and more natural. Complex design artifacts and structures among Web objects are modeled as atomic or composite components, while internal structure of a Web page is versioned in a fine-grained manner as logical units. A set of *structural* difference tools was developed for Web objects such as HTML pages, XML documents, programs, scripts, graphics, etc.

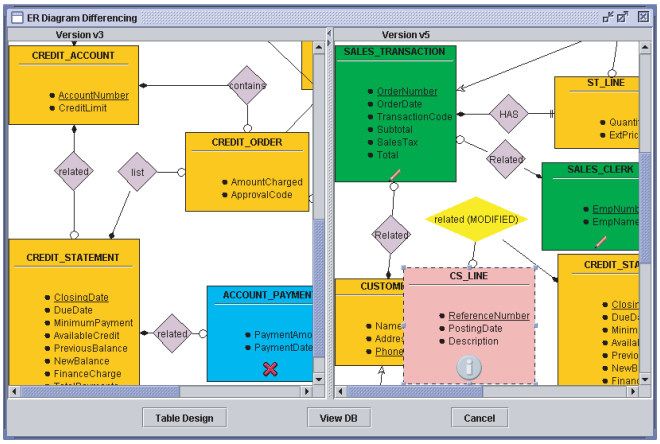


Figure 8: Structural difference tool for ER diagrams

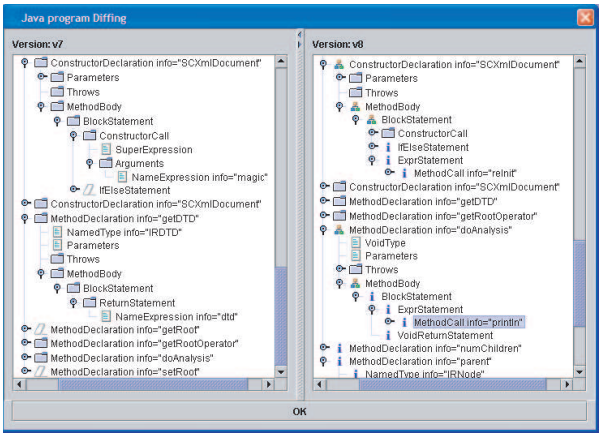


Figure 9: Structural difference tool for programs

UML-based OO software development: Another prototype is a UML-based object-oriented development environment. We have wrapped around the internal representation of Thorn UML editor [22] to make it compatible with our structure versioning framework. In this case, the directed graph of the project entity represents the structure of a UML class diagram. Each node in the graph has its "component" slot referring to a *UML class component* or a *UML interface component*. Similar to MolhadoArch, product versioning helps to maintain *version consistency* between UML diagrams and source code. Developers will never have the version mismatch problem between design objects in a diagram and source code.

Relational database application development: We have also used Molhado to build a prototype of a relational database application development environment. The overall system structure in this case is an ER diagram, represented by the project's directed graph in Molhado's system model. New types of components are defined (*entity*, *relationship*, etc). Although no real back-end database is currently used (data in tables is stored in XML), this prototype shows its unique ability to help engineers not only to manage versions of ER diagrams and tables, but also to control fine-grained and structural changes between versions. Figure 8 shows structural differences between two versions of an ER diagram. Between *v3* and *v5*, entity "ACCOUNT PAYMENT" was deleted, entity "CS LINE" was inserted, entities "SALE TRANSACTION" and "SALE CLERK" have been modified in their attributes, and the relation "related" was changed. This kind of fine-grained, structural difference tool was easily built due to Molhado's structure-oriented representation, in which every change to the representation graph of the diagram is recorded as described in Section 4.

Multi-level, fine-grained version control system: Molhado's fine-grained, structural differencing and versioning mechanism benefits not only for design objects but also for source code. Applying Molhado's structure-oriented representation for source code, we easily produced a multi-level, fine-grained versioning system for Java programs and structured documents. The main idea is to implement a program or a structured document as an atomic component by using our tree data structure with additional attributes. For example, a program AST is realized as a tree where each node is associated with a slot that define a syntactical unit in a program [16]. To accommodate comments, formatting and presentation information, embedded multimedia documentation, and embedded hyperlinks within a program, we created special attributes, defining additional slots that hold those information for each node in an AST [18]. Since the tree-based versioning algorithm in Section 4 can be applied to a subtree at any node, the history of *any syntactical unit* in an AST is captured. In addition, those non-program information can also be versioned. A structural difference tool for programs was also built (see Figure 9). The icon next to a syntactical unit shows its changing status from one version to another. For example, the methods "getRoot" and "setRoot" have been deleted (see the left window), and the methods "numChildren" and "parent" have been added at version *v8* (see the right window).

The same representation approach is applied to structured documents in XML in a straightforward manner. Each element node in an XML document tree is associated with a slot containing the element's name, and has one additional slot for each XML element-level attribute that is defined for the element. Each XML's character data (CDATA) string is represented by a node with an additional slot (defined by "content" attribute) that holds the string. With Molhado's fine-grained versioning, an XML document is versioned at any element level. Since XML elements are defined by users in their documents, the users have the control over the levels of granularity that need to be versioned in that document. This multi-level, fine-grained version control is more flexible than existing fine-grained versioning systems [4, 13] where the granularity of versionable information unit is predefined and fixed. In fact, fine-grained version control is achieved not only for programs or XML documents, but also for any abstraction represented by a node since all changes to versioned slots associated with the node are recorded. For example, in Molhado's versioned hypermedia system, a first-class link, represented by a node, can also be versioned [17]. In addition, this mechanism can easily be applied to version any artifacts that can be represented in XML format.

Traditional document-directory structure: In many existing file-oriented SCM systems, the organization of files in a repository directly reflects the file/directory structure in a file system. To support this, we defined an additional object type, *directory component*. In this case, the directed graph of the project entity is in a tree form, representing the organization of programs and documentation. The "component" attribute maps each tree node to a versioned slot that refers to a document component (if the node is a leaf node) or to a directory component (if an intermediate node).

8.2 Discussions

Those SCM-centered environments have shown clear advantages over existing ones in the same domains. Their distinguished characteristics are due to the Molhado's infrastructure. First of all, the most crucial feature is their ability to manage versions of a software

system and its objects at the logical level. In addition to versioning supports for source code, they are able to provide SCM supports for artifacts that have very different nature than source code and live in contexts that are traditionally not suitable for existing SCM systems. An important benefit for developers is that they are free from worrying about actual file storing. Second, Molhado can accommodate complex software development paradigms due to its extensible and logical system model. Abstractions, structures, and relations imposed by any paradigm can be modeled and versioned. Fine-grained versioning and configuration management can be achieved at multiple levels of abstraction and granularity.

Third, thanks to product versioning, consistent configurations are maintained among logical objects across entire software project. Modifications made to objects are integrally captured and related to each other. Therefore, the evolution of interrelations between objects of different logical models or levels in a development process is also managed. For example, Molhado always guarantees version consistency among design objects and source code. Finally, Molhado's structure-oriented representation enables multi-level, fine-grained version control for any kind of structured objects from complex artifacts to source code and specifications. This also facilitates the constructions of structural difference tools for software artifacts. It is very cumbersome for text file-based SCM systems to build this sort of fine-grained structural difference tools.

While experimenting with Molhado infrastructure, we have observed that the reuse of Molhado's modules significantly eases development. Development cost can be reduced since technically, each of those systems only requires new object types, associated tools, and a project structure editor. Time and efforts are reduced in developing those systems. Each was completed by one or two graduate students in about three weeks and comprised of about a thousand lines of new code. Most of the new code implement user interfaces for new project structure and object editors. In summary, our experiments showed that Molhado can reduce the time and efforts from developers and help them to rapidly develop object-oriented SCM systems and associated SCM-centered environments.

9. RELATED WORK

Numerous industrial and research systems have been produced in the area of SCM. According to Van der Hoek *et al* [27], SCM system architectures have also been improved over time from monolithic systems to reusable, pluggable architectures.

Traditionally, a SCM system is constructed as a *single* and *monolithic* entity. Early systems provided only version control supports such as SCCS [20] and RCS [23]. Advanced and recent systems provide more complete configuration management facilities such as CoED [4], Perforce [19], etc. The design of these SCM systems precisely support their associated SCM needs. Although these infrastructures are inflexible and inextensible, their storage mechanisms are optimized to match the needs of SCM policy.

To reduce the efforts and time to build these monolithic SCM systems from the ground up, SCM developers and researchers have turned to reuse existing SCM systems or generic database systems as storage facilities [27]. For example, several SCM systems were constructed by wrapping around RCS with additional functionality such as CVS [15], Ragnarok [6]. On the other hand, ClearCase [12], SYNERGY/CM [7], and TrueCHANGE [25] were built on top of generic database systems. A drawback of this approach is the disconnection between the SCM model as provided by the underlying technology and the needs of the new SCM system [27]. For example, RCS focuses on version control for individual files and does not support the management of the collections of objects. Addressing this disconnection often requires developers to implement complicated mappings from a desired SCM model to an actual SCM model as supported by the reused database or SCM system.

To address the problem, many SCM infrastructures have been created. Instead of reusing entire SCM system or generic database, they each provide a reusable repository that supports the storage of versioned artifacts, and exposes a programmatic interface designed for manipulating those artifacts. This class of infrastructures is focused on *reusable repository* with a *generic SCM model* [27]. Developers use provided functions to customize the generic repository to the needs of the desired SCM policy, and build the rest of the SCM system on top of the customized repository. ICE [32] and EPOS [29] are logic-based and support the reuse of SCM models only. ICE is based on feature logics and puts emphasis on reusing logic rules. Feature logic in ICE can be employed as a base mechanism on top of which different version models may be realized. In EPOS, configuration rules put constraints on delta application. CME [11] extends a programmatic interface to RCS with composite component management. CME only supports the check-in/check-out model and is limited to file versioning. Gradient [3] is a SCM repository that is based on automatic replication. But, as with CME, it only supports the check-in/check-out policy.

The version engine in CoMa [28] is graph-based. CoMa introduces PROGRES, a graph rewrite language, as a tool for constructing SCM policies. In contrast to our combination approach between version and data models, CoMa is based on a composition version model, which is on *top* of a data model. This makes CoMa complicated for building object-oriented SCM systems. DAMOKLES [9] uses the combination approach, but it is based on EER database model. Although Molhado currently does not provide the same level of robustness and reliability as a database, but its data model is more specialized towards SCM. Additionally, the process of defining a new type of a composite object in DAMOKLES is based on a database schema, which is not as natural as the object-oriented approach in Molhado. WebDAV and DeltaV [30] provide a generic SCM model and, through new HTTP methods, provide a programmatic interface for building Web clients.

An infrastructure that is more advanced in this class (i.e. reusing a generic SCM model) is NUCM [24]. NUCM separates repositories from configuration management policies by providing a generic model of a distributed repository and an associated programmatic interface. Specific SCM policies are programmed as extensions to the generic interface, while the underlying distributed repository is reused across different SCM policies [24]. To support composition, NUCM has *atoms* and *collections* in its storage model. Collections in NUCM need to track the physical repository in which each member artifact is stored. Developers have to write code to map new object types in their SCM systems into atoms and collections. In contrast, in Molhado, the details of mapping an object to physical storage are not exposed to developers. Molhado leverages the definition of objects to the logical level while its storage model handles the persistence of attribute tables. In addition, the modeling of composite objects in NUCM is less general due to a restriction in the directed graph representing a composition of atoms and collections: no directed edge is allowed between siblings of a collection node. Therefore, no structure can be defined among direct sub-components of a composite component.

To go beyond the reuse of a generic SCM model, MCCM [27] additionally allows reuse of SCM policies. MCCM provides an enhanced architecture that supports pluggable modules. This SCM infrastructure represents for an advanced class which emphasizes reusing SCM policies in association with a generic, pluggable repository. BAMBOO [31], taking a model-driven approach, is a new infrastructure leveraging a generic SCM domain model to gener-

ate new SCM repositories. In general, compared to Molhado, the infrastructures in this class are more flexible since both SCM policies and a generic model are reused. However, Molhado is highly specialized towards *object-oriented* SCM systems. By imposing SCM policy design choices (such as product versioning) on resulting SCM systems, Molhado maximizes the reuse of built-in SCM modules and minimizes developers' effort. In Molhado, the object-oriented approach to reuse in object-oriented SCM systems makes the definition of new object types more natural and easier.

The object-oriented, versioned databases related to Molhado include ODE [1], TVOO [21], etc. However, they are highly specific to the ER model with SQL. One could consider to implement this sort of models by using our infrastructure. In addition, tree-based version control in COOP/Orm [14] and the Unified Extensional Versioning Model [2] are closely relevant to Molhado's fine-grained product versioning. Although the fine-grained versioning framework in COOP/Orm is based on similar principles as Molhado, it is restricted to tree-based hierarchical documents. The Unified Extensional Versioning Model manages versions of a composite component via *link nodes*, which are similar in nature to slots with references in Molhado. However, versions of a composite are handled in a more complicated way: when a component is modified to create a new version, new versions of all components that are ancestors in the compositional hierarchy must be created.

10. CONCLUSIONS

This paper has demonstrated the feasibility and practical application of an object-oriented approach to reuse in object-oriented SCM systems, which manage the evolution of a software system in terms of logical objects, structures, and relationships among them. We presented Molhado, a novel SCM framework and infrastructure, that consists of a product versioning model, an extensible and logical system model, an object versioning framework, a product versioning SCM infrastructure, and a pluggable architecture. Through that SCM infrastructure, which is highly specialized toward object-oriented SCM, Molhado can achieve a high level of reuse and help developers to quickly create the core of an object-oriented, multi-level SCM system and a SCM-centered development environment for any paradigm. As shown by the application of Molhado's infrastructure to the implementation of prototypes of SCM systems and associated editing environments, Molhado creates a significant reduction in developers' effort and time. These prototypes showed advantages over existing environments in the same domains. The framework and infrastructure can also be applied to build version control and SCM services in any editing environments.

While experimenting with Molhado's infrastructure, we have experienced limitations. The specialized editors for new object types need to be Molhado versioning savvy, i.e. they must be able to call Molhado's SCM library functions. *Structured editors* for programs are more appropriate and efficient for our structure versioning and differencing tools. To work with external editors, import/export functionality must be implemented. Limited support for collaboration is also an issue. Our infrastructure is still experimental and there may exist frameworks that it does not support well.

11. ADDITIONAL AUTHORS

Cheng Thao (Computer Science Dept., University of Wisconsin-Milwaukee, email: `chengt@cs.uwm.edu`)

12. REFERENCES

[1] R. Agrawal, S. Buroff, N. Gehani, and D. Shasha. Object versioning in ODE. In *Proceedings of IEEE 7th International Conference on Data Engineering*, pages 446–455, Kobe, Japan, 1991.

[2] U. Asklund, L. Bendix, H. Christensen, and B. Magnusson. The Unified Extensional Versioning Model. In *Proceedings of the 9th International Workshop on Software Configuration Management, SCM-9*. Springer, 1999.

[3] D. Belanger, D. Korn, and H. Rao. Infrastructure for wide-area software development. In *Proceedings of the 6th International Workshop on Software Configuration Management, SCM-6*, pages 154-165. Springer, 1996.

[4] L. Bendix, P. Larsen, A. I. Nielsen, and J. L. Peterson. CoED-A Tool for Cooperative Development of Hierarchical Documents. In *Proceedings of the 8th International Workshop on Software Configuration Management, SCM-8*, pages 174-187. Springer, 1998.

[5] J. Boyland, A. Greenhouse, and W. L. Scherlis. The Fluid IR: An internal representation for a software engineering environment. http://www.fluid.cs.cmu.edu.

[6] H. Christensen. The Ragnarok software development environment. *Nordic Journal of Computing*, 6(1):4-21. January 1999.

[7] SYNERGY/CM. http://www.telelogic.com/products/synergy/cmsynergy/index.cfm.

[8] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 24th Int. Conference on Software Engineering*. IEEE, 2002.

[9] K. Dittrich, W. Gotthard, and P. Lockemann. DAMOKLES: a database system for software engineering environments. In *Proceedings of the International Workshop on Advanced Programming Environments*. Springer Verlag, 1986.

[10] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, Feb 1989.

[11] J. Hunt, F. Lamers, J. Reuter, and W. Tichy. Distributed configuration management via Java and the World Wide Web. In *Proceedings of the 7th Int. Workshop on Software Configuration Management, SCM-7*. Springer, 1997.

[12] D. Leblang. The CM challenge: Configuration management that works. *Configuration Management*, 2, 1994.

[13] Y. Lin and S. Reiss. Configuration management with logical structures. In *Proceedings of the 18th Int. Conference on Software Engineering*. IEEE, 1996.

[14] B. Magnusson and U. Asklund. Fine-grained revision control of Configurations in COOP/Orm. In *Proceedings of the 6th Int. Workshop on Software Configuration Management, SCM-6*, pages 31–48. Springer, 1996.

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

[16] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. Architectural Software Configuration Management in Molhado. In *Proceedings of 20th Int. Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2004.

[17] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. The Molhado Hypertext Versioning System. In *Proceedings of the 15th Conference on Hypertext and Hypermedia (Hypertext)*, pages 185-194. ACM Press, 2004.

[18] T. N. Nguyen, E. V. Munson, and C. Thao. Structured Software Configuration Management for Web projects. In *Proceedings of the 13th International World Wide Web Conference (WWW 2004)*, pages 433-443. ACM Press, 2004.

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

[20] M. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[21] L. Rodriguez, H. Ogata, and Y. Yano. An access mechanism for a temporal versioned object-oriented database. *IEICE Transactions on Information and Systems*, E82-D(1), January 1999.

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

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

[24] A. van der Hoek, A. Carzaniga, D. Heimbigner, and A. Wolf. A testbed for configuration management policy programming. *IEEE Transactions on Software Engineering*, 28(1):79-99, January 2002.

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

[26] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic. Taming architectural evolution. In *Proceedings of 9th ACM International Symposium on Foundations of Software Engineering, (ESEC/FSE-9)*. ACM Press, 2001.

[27] R. van der Lingen and A. van der Hoek. An experimental, pluggable infrastructure for modular configuration management policy composition. In *Proceedings of the 26th International Conference on Software Engineering, (ICSE)*, pages 573–582. IEEE Computer Society, 2004.

[28] B. Westfechtel. A graph-based system for managing configurations of engineering design documents. *Journal on Software Engineering and Knowledge Engineering*, 6(4):549–583, December 1996.

[29] B. Westfechtel, B. Munch, and R. Conradi. A layered architecture for uniform version management. *IEEE Transactions on Software Engineering*, 27(12):1111–1133. IEEE Press, December 2001.

[30] E. J. Whitehead, Jr. WebDAV and DeltaV: collaborative authoring, versioning, and configuration management for the Web. In *Proceedings of the 12th ACM Conference on Hypertext and Hypermedia (Hypertext)*. ACM Press, 2001.

[31] E. J. Whitehead, Jr., G. Ge, and K. Pan. Automatic generation of hypertext system repositories: a model driven approach. In *Proceedings of the 15th ACM conference on Hypertext and Hypermedia (Hypertext)*. ACM Press, 2004.

[32] A. Zelller and G. Snelting. Unified versioning through feature logic. *ACM Trans. on Software Engineering and Methodology*, 6:397–440, October 1997.