Object-Oriented, Structural Software Configuration Management

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

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

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

ABSTRACT

Capturing the evolution of logical objects and structures in a software project is crucial to the development of a high-quality software. This research demonstration presents an *object-oriented* approach to managing the evolution of system objects at the *logical* level. Keys to our approach are its *extensible, logical,* and *object-oriented system model* and *structure versioning framework* in which types of logical objects and structures in a software system are extended from a small set of the system model's basic entities, allowing them to be versioned in a *fine-grained* manner and *independent* of the physical file structure. Changes to all logical objects and structures are captured and related to each other in a tightly connected and cohesive manner via the *Molhado* product versioning software configuration management (SCM) infrastructure. We also demonstrate our object-oriented SCM approach by applying it in different development paradigms such as UML-based object-oriented software development, architecture-based software development, and Web application development.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management

General Terms: Management

Keywords: Object-Oriented, Software Configuration Management, Version Control

1. INTRODUCTION

The ability to manage the evolution of these logical abstractions, objects, and structures during a design and implementation process is crucial to the development of a high-quality software. However, many *file-oriented* SCM systems treat a software system as a "set of files" in directories on a file system, and consistent configurations are defined implicitly as sets of file versions with a certain label or tag. This creates an impedance mismatch between the design and implementation domain (architectural level) and the SCM domain (file level). SCM systems, whose concepts are heavily based on physical structure can become burdensome for ordinary developers partly because design and implementation methods and SCM infrastructures require different mental models.

Even advanced SCM systems, which can capture the evolution of non-program software artifacts, do not provide much support beyond storing different versions of those artifacts. For example, in current practice, system architecture is often versioned as simple text or graphic files, whose logical contents are irrelevant to SCM systems. Therefore, the semantic connection between the architectural entities and source code is difficult to manage. Some systems heavily depend on a line-oriented model of internal changes that disregards *logical* structures of software artifacts. In file-oriented SCM systems, logical relationships among software objects during a development process are not versioned. For example, Java classes are often versioned in terms of files, therefore, the evolution of the class hierarchy among them is hardly managed.

To minimize the gap between software designs and SCM, the SCM research community has investigated ways to provide version control for software objects and structures. However, in most of existing SCM systems, a text file or a source code file is often divided into smaller logical units (such as modules, sections, classes, or functions), which are versioned as individual files. Then, versions of those files are combined to create the versions of a composite object. The problem with this approach is that the granularity of versionable information units is fixed and the versioning system is inflexible and inextensible. For example, in POEM [2], where the basic versionable unit is a function, it is impossible to manage versions of finer units such as statements or expressions. A more serious problem is that these systems actually control versions of *parts* of software *documents*, rather than versions of *logical objects*. For example, they are able to manage versions of a fragment of a software architecture *specification* that describes an architectural object in a system architecture. However, the evolution of the architectural object itself is *not* really managed. In brief, it is necessary to have *object-oriented SCM tools* that allow users to manage the evolution of logical abstractions and relationships without worrying about the concrete level of actual file versioning and storing.

2. MOLHADO APPROACH

The key departure point of Molhado from other SCM approaches is its *extensible, object-oriented system model* and *structure versioning framework*. The system model is logical and extensible to enable the definition of any new type of objects in accordance with the structure versioning framework. The versioning framework allows logical objects to be versioned at both fine and coarse granularities. Versions and configurations among objects are managed via the Molhado *product versioning* SCM infrastructure, which relates all changes in objects at both design and implementation levels in a cohesive manner. The details are as follows.

Unlike other SCM models, Molhado places all system objects in a *uniform, global version space*. A *version* is global across the whole project and is a point in a *tree-structured discrete time* abstraction, rather than being a *particular state* of a system object as in other versioning models. The state of the whole software system is captured at certain discrete tree-based 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.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

When the current version is set to a captured version, the state of the whole project is set back to that version. Changes made to 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. This approach is called *product versioning*.

In Molhado, logical objects and structures are built from a primitive data model, named *Fluid Internal Representation* (IR) [1]. In the model, a *node* is the basic unit of identity and is used to represent any abstraction. A *slot* is a memory location that can store a value in any data type, possibly a reference to a node or a sequence of slots. A slot can exist in isolation but more typically slots are attached to nodes, using an *attribute*. 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. The data model can thus be regarded as an attribute table whose rows correspond to nodes and columns correspond to attributes and the cells are slots. Once we add versioning, the table gets a third dimension: the version. Fluid's persistence model can store and retrieve different types of versioned data in attribute tables. From Fluid IR, a fine-grained versioning algorithm is developed for *trees* and *directed graphs*. Trees and directed graphs are built from nodes, slots, and attributes. Depending on the current version, the algorithm can properly retrieve the shape of a tree or a directed graph and versioned slots that are associated with its nodes.

To model a software project and its logical objects, an object-oriented system model is defined. Its basic set of entities includes *logical unit*, *component* (*atomic* and *composite component*), and *project*. Concrete logical object types in a software project will be extended from these entities and must be built in accordance with our structure versioning framework. In that framework, a *component* basically represents for a *logical object*. An *atomic component* (representing for an atomic object) is the basic unit for composition and aggregation in a *composite component* (representing for a composite object). In Molhado, to accommodate fine-grained versioning, an atomic object can be internally composed of finer units, called *logical units*. For example, a class hierarchy (modeled as a composite component) is composed of classes (as atomic components), where a class can be represented by an abstract syntax tree (AST) of syntactical units (as logical units).

An atomic component might have no internal structure. But if it has (e.g. a class), its internal structure is modeled by a tree or a directed graph where each node represents a logical unit. An object derived from this type of component must contain a tree or a directed graph data structure built from Fluid IR. Therefore, the internal structure of an atomic component is versioned via the tree-based and graph-based fine-grained versioning algorithm. For example, the history of any syntactical unit in a program or any XML element in an XML document can be recorded. Composite objects or logical structures such as compound documents, architectural composite components, class hierarchy, UML diagrams, can be constructed in a similar way as atomic components. That is, the internal structure of a composite must be represented by either a tree or a directed graph. However, an additional "component" attribute defines for each node in that tree or graph a versioned slot referring to a component that the composite contains. A component's internal properties whose history needs to be captured are represented by versioned slots that are contained within the component as its fields. When a project version is chosen as *current*, the directed graph of a composite component will be correctly retrieved and versioned slots associated with nodes in the graph will refer to proper constituent components at the current version as well. Then, the internal structure of each constituent component and the contents of versioned slots are also properly determined.

A *project* is a named entity that represents the *overall system logical structure* of a software project. A project has a structure that is composed of components. Depending on the software development framework used, a project will represent various forms of system structures. Classical hierarchical types of overall system structure such as classes/packages, files/directories, or more advanced types such as UML class hierarchies or control hierarchies can be easily modeled as trees. More complex types including architectural design diagrams, data flow diagrams, or ER diagrams require the use of a graph-based scheme. To be general, a project contains a directed graph. For a given node in that graph, the associated "component" slot contains a reference to a composite or an atomic component. Since the overall structure (i.e. a *project*) is represented as a directed graph, it is versioned according to the graph-based versioning algorithm. Similar to composite components, the construction of a consistent configuration is always guaranteed since when a project version is chosen as current, the project's directed graph and components will be correctly retrieved.

The Molhado's object-oriented SCM approach created several benefits: 1) structure versioning infrastructure allows for definitions of new object types (components) to accommodate different software development paradigms; 2) objects can be versioned at both fine (logical unit level) and coarse (component level) granularities since the history of any abstraction represented by a node can be recorded; 3) the approach also facilitates the construction of comparison tools for two different versions of any structured objects, for example, the system hierarchical structure, a structured document or a program in both structural and line-oriented fashions; 4) versioned hypermedia infrastructure for software traceability is easily built since all hypermedia structures and individual relationships are uniformly versioned as other structured objects; 5) traceability links are maintained among *objects*, rather than among *software documents* as in traditional traceability tools.

To demonstrate the use of Molhado, we built four prototypes of object-oriented SCM systems to accommodate different software development frameworks including architecture-based software development [4], Web application development [5], UML-based object-oriented development [3]. Types of objects span a wide range from architectural elements in software architectural designs, UML diagrams, XML, HTML, graphic documentations, to program source code. To make Molhado compatible with file-based SCM systems, we also produced an SCM system that is able to support the logical document-directory organization of programs and documentations. In brief, the OO approach in Molhado allows us to produce SCM systems that works at the object level.

3. REFERENCES

[1] John Boyland, Aaron Greenhouse, and William L. Scherlis. The Fluid IR: An internal representation for a software engineering environment. http://www.fluid.cs.cmu.edu.

[2] Y. Lin and S. Reiss. Configuration management with logical structures. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 298–307, 1996.

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

[4] Tien N. Nguyen, Ethan V. Munson, John T. Boyland, and Cheng Thao. Architectural Software Configuration Management in Molhado. In *Proceedings of 20th International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society Press, 2004.

[5] Tien N. Nguyen, Ethan V. Munson, and Cheng Thao. Fine-grained, structured software configuration management for Web projects. In *Proceedings of the 13rd International World Wide Web Conference*. ACM Press, 2004.