Managing Software Architectural Evolution at Multiple Levels of Abstraction

Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University, Ames, IA 50011, USA
Email: tien@iastate.edu

Abstract— **Software development is a dynamic process where engineers constantly modify and refine systems. As a consequence, system architecture evolves over time. Software architectural evolution has been managed at different abstraction levels: the meta level, the architectural level, the application level, and the implementation level. However, management supports for architectural evolution are limited to evolution mechanisms in architectural description languages such as subtyping, inheritance, interface, and genericity. This paper presents a *model-oriented* version and configuration control approach to managing the evolution of architectural entities and relationships among them in configurations at different levels of abstraction.**

This paper also illustrates our approach in building an architectural configuration management system, *MolhadoArch*, that is capable of managing configurations and versions of software architecture across multiple levels of abstraction in a uniform and tightly connected manner. In MolhadoArch, consistent configurations are maintained not only among source code but also with the high-level software architecture. MolhadoArch supports the management of both planned and unplanned evolution of software architecture. We have conducted an experimental study to show that MolhadoArch can handle large and real-world systems. By evaluation, we learned that the benefits outweigh the extra space needed to represent architectural entities.

I. INTRODUCTION

Software architecture [1] defines the overall logical structure of high-level design of a software system in terms of components, interactions, and relationships among them. Software architecture provides conceptual integrity for a system and the mental framework that engineers use to design, discuss, document, and reuse its elements. It can also be used for generating partial or full implementations and for structuring the repository of software artifacts.

The ability to manage architectural evolution is crucial to a successful software development and maintenance process. Software architectural evolution has been managed at different abstraction levels: the meta level, the architectural level, the application level such as in the Software Architecture EVolution Model (SAEV) [2]. Unlike source code, for which the use of a software configuration management (SCM) system is the predominant approach to capturing evolution, the management supports for architectural evolution are still limited to evolution mechanisms in architectural description languages (ADLs) such as subtyping, inheritance, interface, and genericity [3]. In addition, architectural SCM systems treat a software architecture as a set of text files in a file system, and consistent configurations are defined implicitly as sets of file versions with a certain label or tag as in CVS [4] or Subversion [5]. Existing architectural SCM systems focus only on the application level and could not manage the evolution of architectural entities that belongs to other levels of abstraction as well as the logical relations among them.

A. Motivation Example

Let us consider a client-server system (used as an example in [2]). According to SAEV [2], at the highest level of abstraction (i.e. the meta level), all ADL architectural entities are defined such as *configuration*, *component*, *connector*, and *interface*. The architectural level is the level of the description of any architecture using one or more architectural entities defined in the meta level. Figure 1 presents an architecture with a configuration C and three component types: SERVER, CLIENT, DATABASE, and two connector types K1 and K2. Application level is the level of description of any application in accordance with its architecture. For example, in Figure 1, an application can made up of: one instance of the configuration type C, two instances of the component CLIENT (c1,c2), one instance of the component DATABASE (Oracle), one instance of the component SERVER (s1), and three instances of connectors (k1,k2, and k3). The dotted arrows represent the architectural relationships such as "instance_of" among entities.

In current practice, a system's architecture described in some graphical notation or in some ADL is often versioned as simple text files, whose logical contents are *irrelevant* to SCM systems. Those systems are able to manage versions of an architecture description file for a software system. However, the architectural relationships among architectural entities, and between those entities and source code that realizes them are hardly managed. These logical relationships are very crucial since they help developers to have good understanding of the architectural design and implementation of a system. When designs or implementation source code are changed, those mappings

This paper is based on "Multi-level Architectural Evolution Management" by Tien N. Nguyen, which appeared in the Proceedings of the 40th IEEE Hawaii International Conference on System Sciences, HICSS 2007, Big Island, Hawaii, USA, January 2007. © 2007 IEEE.

© 2008 ACADEMY PUBLISHER

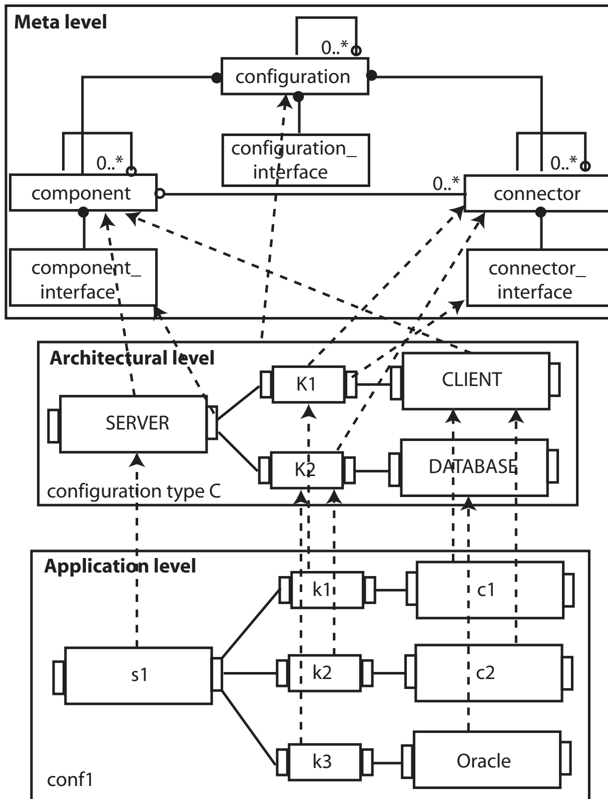


Figure 1. Motivation Example

also evolve over time. Unfortunately, no SCM system provides supports for managing the version consistency of architectural relationships. Therefore, developers often run into the version mismatch problem in which versions of software architecture and source code are not compatible. Furthermore, these relationships connect *logical* entities together, rather than files. Therefore, text file-oriented SCM systems are not well-suited to manage architectural evolution at multiple levels of abstraction. In other words, supports for software architectural evolution in ADLs are moving toward *model-oriented*. However, architectural SCM supports are still *file-oriented*, disregarding logical contents of architectural description files.

Text file-oriented architectural SCM systems represent changes in architecture in term of *text lines* of a file that have changed, instead of entities that changed. In those systems, changes between versions are reported in term of disconnected sets of *changed lines* in the *textual* representation of the model. Furthermore, in those systems, merging support, which is to incorporate changes from multiple developers to the same architectural design, is very limited and error-prone. Making branches in those systems is easy, but merging the changes can be hard. File-oriented SCM systems can merge changes automatically if they are to different parts of a file, but if two branches change the same lines of texts in a file then the merge fails and must be done manually. Even successful file-based merge might result in an *incorrect* model since the lines do not match very well with logical entities in an architectural design.

B. Model-Oriented Architectural Configuration Management Approach

To bridge that gap and to address the aforementioned problems, we introduce a **model-oriented** configuration management approach to managing versions of architectural entities, source code, and the multi-level, architectural relationships among them. The relationships can be within- or between levels. The departure point of our approach is the focus on the models of architectural evolution, rather than architectural description files. Another distinguished feature is its capability of managing the relationships across different levels of abstraction in an architectural evolution model. A configuration at each level is represented by an attributed, directed graph. Architectural entities are explicitly represented and versioned. A novel directed graph-based version control framework is developed to support the version management of system architecture, entities, and relationships. To illustrate our ideas, we have also taken advantage of Molhado's versioned data repository and infrastructure [6] to build *MolhadoArch*, a model-oriented architectural SCM system. The evolution of architecture entities in multiple abstraction levels and implementations are captured in a tightly connected and cohesive manner.

The next section discusses related work on architectural configuration management. Sections 3 and 4 describe our graph-based representation model for architectural entities. Section 5 explains our fine-grained graph-based version control scheme for architectural evolution at different levels. Section 6 focuses on our representation for artifacts at the implementation level. The tool development in MolhadoArch is presented in Section 7. Section 8 reports our empirical evaluation. Conclusions appear last.

II. RELATED WORK

This section discusses existing SCM-based approaches to managing architectural evolution. In their analysis, Westfetchtel *et al* [3] classified those approaches into following categories: *orthogonal integration*, *SCM-supported software architecture*, *SCM-centered software architecture*, and *architecture-centered SCM*.

The SCM community has developed a large number of models for capturing the software evolution [7]–[9]. SCM systems are traditionally focused on source code. More advanced tools such as ClearCase [10], Rational-Rose [11], and System Architect [12], can manage non-program software artifacts and structures among them. However, in these systems, architectural evolution is only recorded via the management of architecture descriptions. Since the description contents are *irrelevant* to the SCM tools, the semantic relationships between architectural entities and source code are often unmanageable. If a description contains multiple objects, configuration control is required. This type of integration between SCM and software architecture is called *orthogonal integration*, where both of them are decoupled as far as possible.

In *SCM-supported software architecture approach*, an architectural design tool makes use of the services from

© 2008 ACADEMY PUBLISHER

an SCM tool. The overall software architecture is decomposed into units (packages or subsystems) which are submitted to the SCM tool. To compose an architecture from a set of architectural units, the architectural design tool offers commands for browsing the versioned object base, for supplying configuration descriptions, and for initiating check in/check out operations. In contrast to orthogonal integration, design tools in this approach are aware of and take advantages of SCM systems. Ragnarok [13] manages architectural evolution via a *total versioning* model [14]. It hides the concrete level of actual file versioning supported by CVS [4], allowing designers to work at the architecture level. SOFA/DCUP [15] has a version model for components employing user-defined attribute taxonomies and entity relations. SubCMTool [16] versions for sub-systems but lacks of supports for connectors and interfaces.

In *SCM-centered software architecture approach*, software architecture is typically represented with the support of system modelling languages, which include mechanisms to manage architectural evolution. *Module interconnection languages* (MILs) [17] address the structure and partly the evolution of systems, but they do not deal with behavior. Some MILs offer simple version control, for instance, multiple realization versions for the same interface. Adele [18] offers a predefined MIL whose system model is defined in terms of a database schema.

ADLs [19] deal with architectural components, rather than modules, implying a potentially coarser granularity. Examples of ADLs include C2SADEL [20], Rapide [21], UniCon [22], PCL [23], Koala [24], etc. These languages often offer facilities and mechanisms to improve reusability and to handle *planned* evolution. Mechanisms, including *genericity*, *inheritance*, *subtyping*, and *interfaces*, effectively introduce different kinds of variants [3]. xADL 2.0 [25] supports the architectural evolution via *versions*, *options*, and *variants* XML schemas. SAEV [2] extends architectural evolution mechanisms in an ADL to support for multiple levels of abstraction. Despite successes, ADLs deal only with variants and options, and do not fully address *unplanned* architectural evolution with the exception of the work by Van der Hoek in Menage [26]. All variants must be represented in an architecture description, and designers choose among them [3]. Esaps and Cafe [27] focused on managing variants in software product families via structural rules.

Architecture-centered SCM approach combines architectural and SCM concepts into a single, unified system model, called *architectural system model*. To cope with architectural evolution, Mae [28] provides concepts of revisions, variants, optionality, and inheritance. The main difference with MolhadoArch is that each instance in Mae's architectural system model is an instance of a *specific version* of a type (i.e. a type has its version number) [28, p. 3]. In MolhadoArch, architectural elements and other objects are placed under a global version space. Therefore, version selection in Mae is done for each individual architectural entity, while in MolhadoArch, the selection of the working version of the whole software system implicitly determines the versions of architectural entities and source code. This product versioning approach in MolhadoArch facilitates the management of implementation mappings between system architecture and source code. However, the separation between types and instances in Mae provides better run-time supports.

Menage [26] manages the product line architectures in terms of both time and space. To address space variabilities, Menage supports the specification of all three kinds of variation points defined by xADL 2.0 [25]. Time variabilities are introduced through check in and check out in its SCM policy. Unicon [22] is focused on implementation-level variability. Based on a property selection mechanism, each component in a given architectural configuration is instantiated with a particular variant implementation. However, its system model does not capture architectural revisions and options. Koala's system model [24] supports variability and optionality via a property mechanism. Koala does not integrate versioning information into its representation. It utilizes an external SCM system instead. ShapeTool [29] does not provide mechanisms beyond grouping, versioning, and version selection. ArchTrace [30] addresses the traceability between architecture and source code through a policy-based infrastructure for automatically updating traceability links every time an architecture or its code base evolves.

III. GRAPH-BASED REPRESENTATION MODEL

System architecture, composite components, configurations are all structured. To represent those structured entities, we use a graph-based representation model. Graphs are commonly known, well understood, have an established mathematical basis (graph theory), and encompass a huge number of concepts, methods and algorithms [31]. 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 architectural entities and configurations.

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. 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 our model, each edge in a

© 2008 ACADEMY PUBLISHER

directed graph can be associated with attribute-value pairs 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. 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* [31]. 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. This constraint is needed to ensure that we have 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 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 [31]. Directed graphs are sufficiently general to be used for a wide variety of entities, 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 architectural entities and configurations. The nested graphs also enable an encapsulation and layering mechanism to reduce the complexity and to hide unimportant details of an artifact from others.

Thirdly, the association of an attribute table to a node or an edge facilitates the modelling of complex configurations and allows us to take advantage of underlying SCM and version control services for different data types provided by the Molhado repository (will be explained later). Molhado relies on the attribute table technology [6]. Finally, source code at the implementation level can be nicely encoded via this attributed, directed graph-based representation model since their abstract syntax trees form a sub-class of this type of graph.

IV. Model-oriented Version Control

A. Versioned Data Model

This section describes how the attributed, directed graphs are implemented. First of all, we would like to summarize Molhado versioned data model. Figure 2 conceptually illustrates the main concepts in that data model: *node*, *slot*, and *attribute*. A Molhado node is the basic unit of *identity*. A Molhado 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 a Molhado node or a set of slots. A slot may exist in a *sequence*, an entity with identity and ordered slots. In a sequence, slots may have the same or different data types. A slot can exist in isolation but typically slots are attached to Molhado nodes, using an attribute. An attribute is a mapping from Molhado nodes to slots. 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 Molhado nodes and columns correspond to attributes. The cells of attribute tables are slots. Once versioning is added, the tables get a third dimension: the version (see Figure 2).

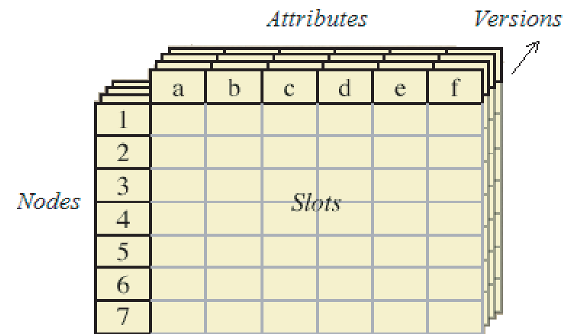


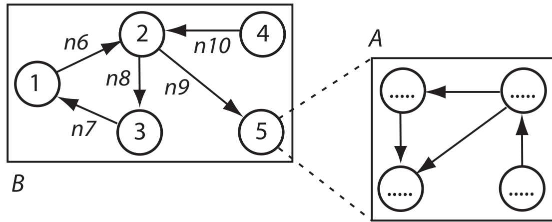
Figure 2. Versioned Data Model

Version control is added into the data model by a third dimension in attribute tables. That is, slots can be versioned. Molhado's version model is called *product versioning* in which a version is *global* across entire system [6]. The third dimension in attribute tables is tree-structured (to accommodate branching) and versions move discretely from one point to another. 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. Molhado has a mechanism to store and retrieve versioned slots that belong to a particular version point.

B. Attribute Table for a Graph

Since a directed graph in our representation model is also based on attribute-value pairs, it is reasonably straightforward to realize a graph via Molhado's data model for versioning purpose. An attribute table is constructed for a directed graph as follows. First of all, each graph node is represented by a Molhado 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 Molhado node. The attributes in those attribute-value pairs are added into the set of attributes of that table. Each edge in the graph is also represented by a new Molhado node (i.e. a new row in the attribute table). Let us call it an "edge" node. The associated attribute-value pairs of an edge are integrated into the attribute table as for graph nodes. 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. Finally, for each Molhado 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 representation will be given next.

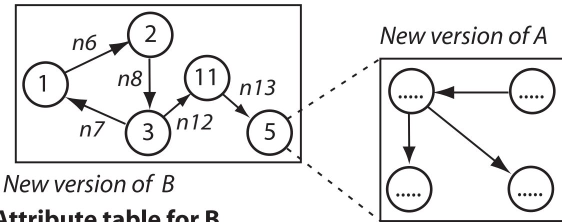
© 2008 ACADEMY PUBLISHER



Attribute table for B

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	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. Composite Component Representation



Attribute table for B

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	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

C. Graph-based Composite Component Versioning

Each architectural entity carries an identifier that serves to identify it *uniquely* within a software system. In our framework, an atomic entity does not contain any other entity. For a composite or structured architectural entity, which contain other entities, we use an attributed, directed graph as the entity's internal structure. Both types of entity are stored in the repository.

In our representation model, a directed graph that contains other graphs will have at least one node that *logically* contains another 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. For a "composite" graph, an additional attribute, attribute "ref", is created to define for each "composite" node a versioned slot containing a reference to an architectural entity that corresponds to the subgraph nested at that "composite" node.

Figure 3 shows an example of the representation of an attributed, directed graph. There are two graphs in the figure: the directed graph corresponding to entity A is nested within the directed graph corresponding to entity B via the node 5. The attribute table in Molhado representing for entity B is shown. Nodes "n1" to "n5" are "node" nodes (i.e. representing for a graph node) while nodes "n6" to "n10" are "edge" nodes (i.e. representing for an edge). Each "edge" node has "source" and "sink" slots. For example, "edge" node "n6" "connects" 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 entity A. The attribute table for entity A is similar (not shown).

The API functions for attributed graphs will be called after the modifications to architectural entities occur. Those functions will update the values of slots in attribute tables including *structural slots* (e.g. "children", "source", and "sink").

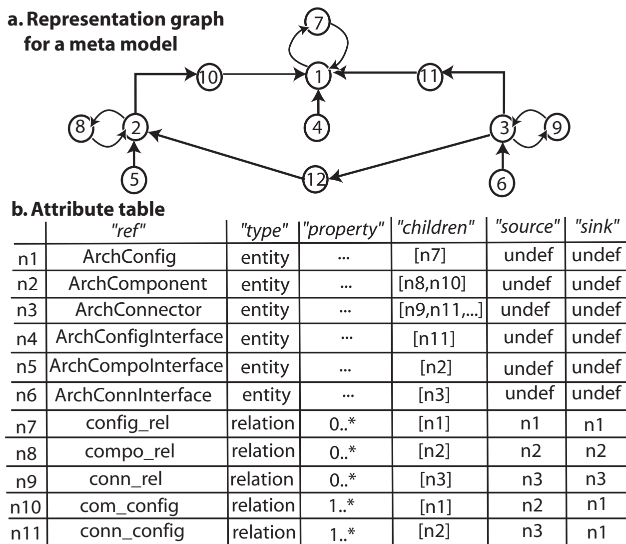
For example, Figure 4 displays a new version of B and A shown in Figure 3. In the new version, the attribute table was updated to reflect the changes to the graph structure as well as to the slot values. For example, since node 4 and edges corresponding to "n9" and "n10" were removed, any request to 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 entity 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 architectural object that is represented by a node.

V. Software Architecture Representation

This section describes how MolhadoArch uses the aforementioned model-oriented versioning mechanism for architectural entities and architectural structure.

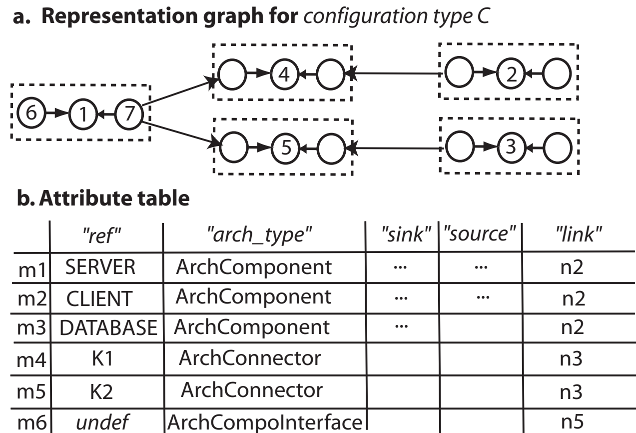
To support the modeling of software architectural evolution at the multiple levels of abstraction, several

© 2008 ACADEMY PUBLISHER



	"ref"	"type"	"property"	"children"	"source"	"sink"
n1	ArchConfig	entity	...	[n7]	undef	undef
n2	ArchComponent	entity	...	[n8,n10]	undef	undef
n3	ArchConnector	entity	...	[n9,n11,...]	undef	undef
n4	ArchConfigInterface	entity	...	[n11]	undef	undef
n5	ArchCompoInterface	entity	...	[n2]	undef	undef
n6	ArchConnInterface	entity	...	[n3]	undef	undef
n7	config_rel	relation	0..*	[n1]	n1	n1
n8	compo_rel	relation	0..*	[n2]	n2	n2
n9	conn_rel	relation	0..*	[n3]	n3	n3
n10	com_config	relation	1..*	[n1]	n2	n1
n11	conn_config	relation	1..*	[n2]	n3	n1

Figure 5. Architectural Meta Model



	"ref"	"arch_type"	"sink"	"source"	"link"
m1	SERVER	ArchComponent	...	...	n2
m2	CLIENT	ArchComponent	...	...	n2
m3	DATABASE	ArchComponent	...		n2
m4	K1	ArchConnector			n3
m5	K2	ArchConnector			n3
m6	*undef*	ArchCompoInterface			n5

Figure 6. Architectural Structure

types of components can be defined and added into the architectural system model. For example, they are: 1) architectural component (*ArchComponent*), 2) architectural atomic component (*ArchAtomicComponent*), 3) architectural composite component (*ArchCompositeComponent*), 4) architectural connector (*ArchConnector*), 5) configuration interface (*ArchConfigInterface*), 6) component interface (*ArchCompoInterface*), 7) connector interface (*ArchConnInterface*), and 8) architectural configuration (*ArchConfig*). Instances of those object types will be used at the architectural and application levels. The *properties*, *constraints*, *style*, and other information of entities are represented as versioned slots associated with entities. The internal structure of an ArchCompositeComponent or ArchConfig is represented via an attributed tree or directed graph depending on the complexity of the structure.

Project is a named entity that represents the *overall architectural structure* of a software system. The basic units in a project are components (atomic and composite). That is, a project is considered to be composed of components. Unlike systems in which each object has its own version space, in MolhadoArch, all instances of concepts are uniformly versioned under a global version space via our *version model*.

Figure 5 shows MolhadoArch's representation for the meta model presented in Figure 1 (ArchAtomicComponent and ArchCompositeComponent are not shown). Technically, an attributed, directed graph is used to represent the architectural meta model. Each entity (configuration, component, connector, interface) is represented as a graph node. The relations between component, configuration, and connector are modelled as a node as well. However, for simplicity purpose, the relation between an entity and its interface is represented as a directed edge (e.g. between component and its interface). The "ref" attribute of a node contains a reference to corresponding entity. For example, the "ref" slot of "n1" refers to ArchConfig. Invariants and properties associated with an entity are represented in "properties" attributes. The connection information is encoded in structural slots such as "children", "source", and "sink".

At the *architectural level*, the architectural structure of a system is represented as an ArchConfig containing an attributed directed graph. Each architectural entity is represented by a node in the graph. Each node is associated with a referential attribute containing a reference to the corresponding architectural entity. Each architectural interface associated with the architectural component is also represented by a node associated with a pointer to the corresponding ArchInterface component. A directed edge connects that ArchInterface node to the corresponding ArchComponent node. A similar representation is used for an architectural connector, except that an ArchConnector node has a reference to an ArchConnector object. Also, if an ArchInterface is associated with an ArchConnector, a directed edge is added to the graph connecting the corresponding ArchInterface node to the corresponding ArchConnector node. For each connection link between two ArchInterfaces (one from an ArchComponent and one from an ArchConnector), there is a directed edge from the ArchInterface node of the ArchComponent to the ArchInterface node of the ArchConnector.

Figure 6 shows the representation for the configuration type C in Figure 1. In the attribute table, each node is associated with a reference to the corresponding architectural element via the attribute "ref". An "arch_type" slot refers to the architectural entity type at the meta level. For example, "SERVER" is an ArchComponent type. The "edge" nodes and the labels of some "interface" nodes are not shown. When an attribute is not applicable to a node, the special value "undef" is used. Finally, the architectural structure of a concrete system at the *application level* is modelled in the similar manner.

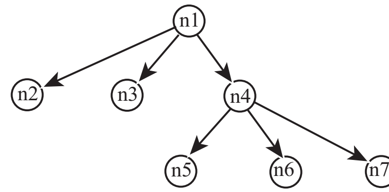
The relationships among architectural entities across different models of abstraction are stored as a special attribute "link" and versioned as well. For example, in Figure 6, the "link" attribute of "m1" contains "n2", meaning that "SERVER" in the configuration C refers to the "ArchComponent" in the meta model.

© 2008 ACADEMY PUBLISHER

Program

```
package nodes;
import content.Packet;
public class PrintServer extends LANNode {
  public void print(Packet p) {
    String packetInfo = getPacketInfo(p);
    System.out.println(packetInfo);
  }
  public String getPacketInfo (Packet p) {
    return p.contents;
  }
  public void accept (Packet p) {
    if (p.addressee == this) this.print(p);
    else super.accept(p);
  }
}
```

Tree Representation



Attribute Table

	"NodeType"	"Name"	"parent"	"children"	"SuperType"	"Modifier"	"RetType"	"Parameters"	"MethodBody"
n1	CompiUnit	"PrintServer"	null	[n2,n3,n4]	undef	undef	undef	undef	undef
n2	PackageDecl	"nodes"	n1	null	undef	undef	undef	undef	undef
n3	ImportDecl	"content.Packet"	n1	null	undef	undef	undef	undef	undef
n4	TypeDecl	"PrintServer"	n1	[n5,n6,n7]	LANNode	"public"	undef	undef	undef
n5	MethodDecl	"print"	n4	null	undef	"public"	"void"	[(Packet, p)]	"String packet..."
n6	MethodDecl	"getPacketInfo"	n4	null	undef	"public"	"String"	[(Packet, p)]	"return p.cont..."
n7	MethodDecl	"accept"	n4	null	undef	"public"	"void"	[(Packet, p)]	"if (p.addresse.."

Figure 7. Source code Representation

VI. Implementation Level

A. Software Documents

This section describes our representation for software artifacts produced during implementation phase including source code and documentation. In MolhadoArch, each documentation is considered to have a hierarchical internal structure, called *document tree*. Each node in the tree represents a structural unit at a level. A child node represents a sub unit of the parent unit. Programs or documentation are represented in this manner.

In particular, to capture the semantics of a program, we develop an entity named *CompilationUnit*, with a tree-based structure representing for the program's Abstract Syntax Tree (AST) (see Figure 7). The class name is one of its properties. That tree-based structure representation is based on Molhado versioned data model, that is, a tree is also encoded within an attribute table. In general, an AST node is represented as a Molhado node with unique, persistent identifier (i.e. a row in the attribute table). To model the parent node and children nodes of an AST node, each Molhado node is associated with two attributes: "parent" attribute defines for each node the *parent* node in the AST, and "children" attribute defines a sequence of references to its children nodes. In addition to those structural attributes, each Molhado node also has an attribute ("NodeType" attribute) that identifies the syntactical unit represented by that node and other attributes representing properties.

B. File Directory Structure

Based on MolhadoArch's SCM infrastructure, we have defined another type of objects called *directory* component. The directory structure of a software project is represented by a tree data structure. An intermediate node in that tree is associated with an attribute referring to a directory component. A leaf node is associated with an attribute containing a reference to either a program component or a documentation component. Figure 8a) shows an example of a directory structure in a software project. Each node in the representation tree in Figure 8b) has an attribute referring to either a directory component or a document component. For example, "n2" is associated with an attribute containing a reference to the directory component "word count", which is consisted of two program components "prog A" and "prog B".

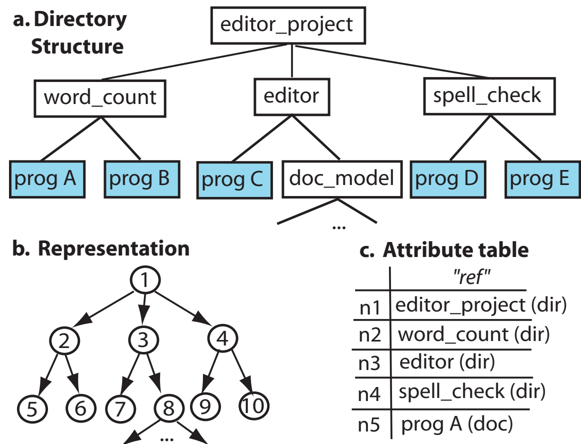


Figure 8. Directory Structure Representation

C. Architecture Implementation Mappings

As mentioned earlier, architectural relations may be established between entities at the application level (in

© 2008 ACADEMY PUBLISHER

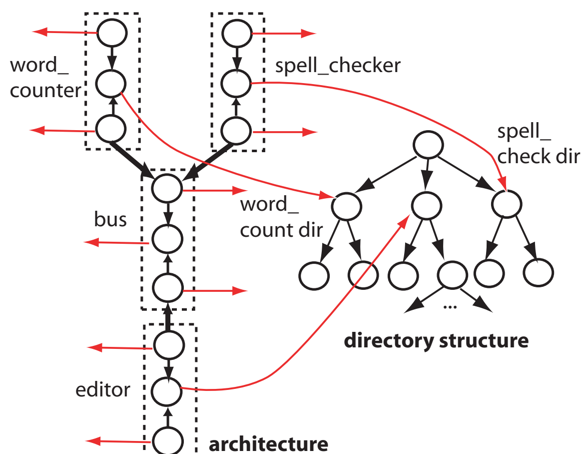


Figure 9. Architectural Relationship Graph

software architecture) and entities at implementation level (in program source code). Note that this traceability relations can be created by developers or automatic analysis tools. MolhadoArch helps developers to manage versions of these mappings over time. Each architectural relationship is represented by an edge between a node in the representation graph of the system architecture and the corresponding node in the representation tree of the program directory structure. The result of those connections is a larger graph, called *architectural relationship graph*. Figure 9 shows the architectural implementation mappings between the architecture of the editing system and source code that realizes it.

The fine-grained versioning scheme described in Section IV is directly applied to achieve fine-grained version control for programs (as ASTs) and structured documents (as XML document trees) since their internal structures and contents are represented as document trees. The history of any document node can be recorded since the scheme is generic for any tree or graph node.

The combined graph between an architecture representation graph and a directory structure representation graph is also versioned according to the directed graph version control scheme. Every single change to the graph can be captured and retrieved. Therefore, not only the directory structure is versioned, but also the evolution of architectural implementation relations can be managed.

VII. Tool Development

This architectural SCM model has been implemented in the MolhadoArch architectural SCM system and its associated SCM-centered architecture-based development environment. Document editors in MolhadoArch are reused from the SC environment [32]. This section highlights important and distinguished SCM features of MolhadoArch.

A. *SCM transaction supports*

MolhadoArch and the SC user interfaces support a variety of transactions. First of all, MolhadoArch parses a description file, creates architectural elements at the meta

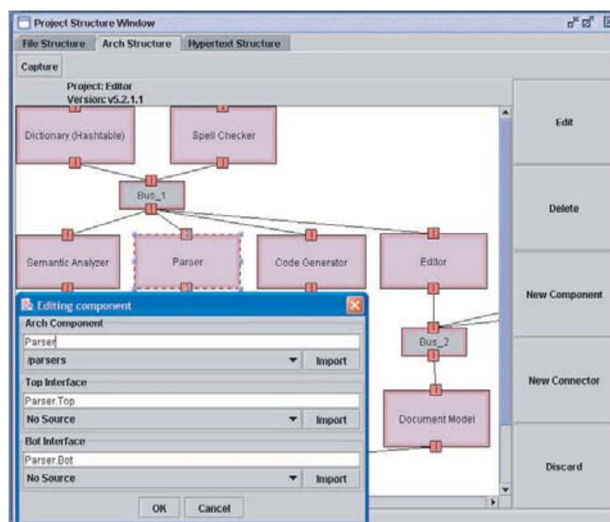


Figure 10. System Architecture Window

level, the architectural level, and the application level. A user can create the initial version of a project's software architecture by either using built-in graphical editing tools. MolhadoArch also supports a system architecture description written in xADL 2.0 [25], an XML-based ADL. A version of the architectural structure can be displayed (see Figure 10). From this window, the user graphically manipulates the architecture to create different versions and variants. To specify the implementations of architectural elements, the user can create new source code via built-in editors or importing external programs into the system. The user can display this logical structure in the same window as architecture (see "Document Model" component in Figure 11) by double-clicking on architectural elements, or in a different window.

The version that is initially displayed in the project structure window as in Figure 10 is called *the base version*. From this window, the user can also edit, delete, import, export programs and documentations, and graphically modify their organization. The user can choose to display any component and an appropriate component editor will be invoked such as structured Java program, XML, HTML, SVG graphics, UML diagrams, or text editors. These editors are version-savvy [32] and are able to import and export documents from and to external formats (Java, XML, HTML, SVG, ASCII) at any version.

If any modification is made to the project at this base version, a new version would be temporarily created, branching from the base version. The word "modified" attached to the base version name signifies that the state of the project at the temporary version has not been recorded yet. The user can choose to discard any derived (temporary) version, or to *capture* the state of the project at a version. Capturing 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 and

© 2008 ACADEMY PUBLISHER

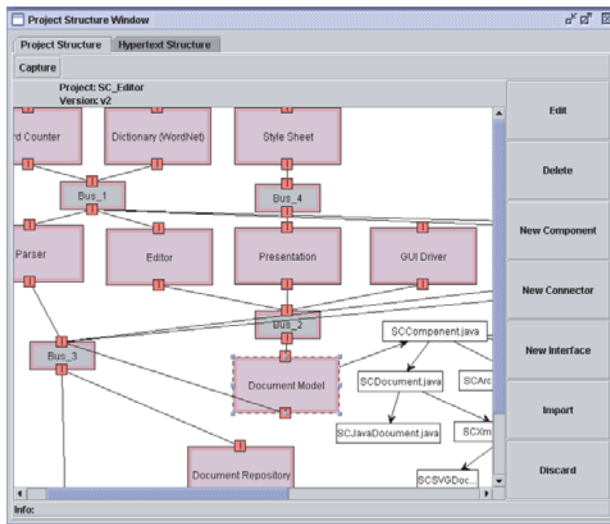


Figure 11. Mapping to Implementation

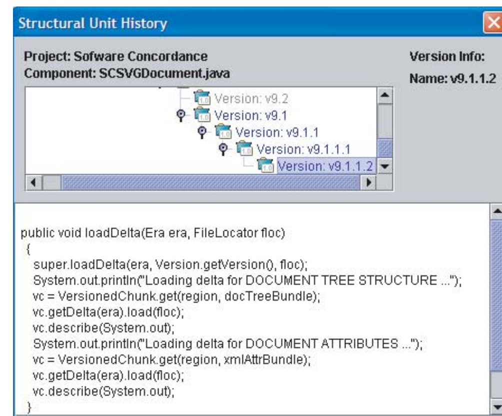


Figure 12. Fine-grained Version Control

it becomes the new base version of the project structure window. However, no data is saved to disk after a capture. While working on one version, the user can always switch to work on (view or modify) any other version. If the user modified the project at the new working version, an additional derived version will be branched off from that new version. If the user moves the mouse focus to a window, the working version is automatically set to the version that window is displaying. The user can also explicitly select a different version from the project history window and open it. Any windows showing old versions (even un-captured ones) are still available should the user want to do additional work on those versions.

The user may *commit* changes at any time. Upon issuing this command, the user is asked which un-captured, temporary versions should be saved and the chosen versions are then saved to the file system 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. All changes in architecture and implementations are integrally saved and related to each other. In current system, each user has his own data files for the project and they can be stored anywhere in a file system. Each user does not see changes made by others. Therefore, no locking mechanism is needed. Users can share data files and use merging tools to collaborate. We implement a centralized repository similar to CVS [4] with an "official" version graph. If the user wants to make a branch, he will just need to copy files of the desired version to his own workspace and work on them. The user can create many "private" versions that others will not see, and copy meaningful versions back to the repository.

B. Fine-grained version control

Unlike many fine-grained SCM systems where granularity of versionable information unit is predefined and hard-coded, MolhadoArch can version any *logical unit* of a component if it is built according to our graph-based versioning framework as demonstrated earlier. For example, with Java classes and XML documents, the evolution of *any syntactical unit* in a Java program or *any element* in an XML document can be captured. In the SC editor, a user can select any logical unit and view its history (see method "loadDelta" in Figure 12).

C. Structural comparison tools

A set of *comparison* tools (*diff* tools) was developed to show differences between two arbitrary versions of 1) a system's hierarchical structure, 2) any component, and 3) any logical unit in both structural and line-oriented fashions. The tools are based on the *Versioned Unit Slot Information* (VUSI) mechanism (see details in [33]). VUSI attaches versioned slots of type "boolean" to nodes in a tree or graph to track if there is a change in attributes of nodes between any two versions. For a program or a structured document, the system visually displays the differences between two versions of any document node (i.e. logical unit). If the chosen document node is the root of a component, changes of the entire component will be shown. Figure 13 shows structural changes in a Java program. The icon next to a syntactical unit's entry shows the change in its status from one version to another. For example, the "tree", "i", "eraser", and "truck" icons represent a modification, insertion, deletion, and relocation of a syntactical unit respectively. For textual nodes, changes can also be displayed in line-by-line manner similar to ViewCVS [34]. It is very cumbersome for SCM systems that heavily depend on line-by-line comparison between versions (e.g. CVS [4]) to build this sort of fine-grained comparison and change tracking tool.

VIII. Experiments and Evaluation

A. Experiment on Space and Time Complexity

An experimental study has been conducted to evaluate the performance and efficiency of MolhadoArch. The results are shown in Table I. We have imported into MolhadoArch Java-based open source systems ranging from small to large sizes (JGraph [35], JTidy [36], Jext [37],

© 2008 ACADEMY PUBLISHER

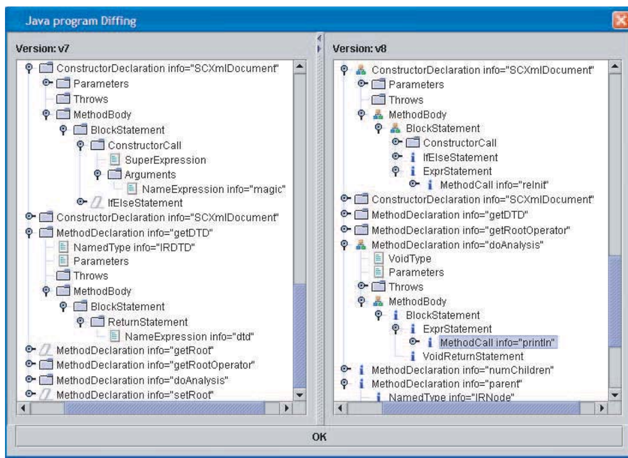


Figure 13. Java Program Comparison

	jGraph	jTidy	SC	Jext	jEdit
No. lines (K)	102	73	62	148	172
Lines of code (K)	18	25	59	98	114
No. Components	235	364	144	659	613
Ext. space (MB)	3.3	5	7.2	8.1	9.2
Int. space (MB)	19	23	27	29	32
Import time (sec)	22	25	29	33	35
Commit time (sec)	2.3	2.8	3.3	5.1	5.3

TABLE I.
SYSTEM EVALUATION RESULTS

TABLE II.
EVOLUTION OF ECLIPSE'S CORE.REFACTORING

Version	LOC	Changed LOC	#Pack	#Classes	#Methods
01/31	19933	-	14	114	868
02/28	19993	1786	13	114	871
03/29	20405	526	13	114	875

and JEdit [38]). SC is also imported into MolhadoArch. The experiment was carried out on a Windows XP computer with Pentium (R) 4 processor at 2.4 GHz, 512 MB of RAM, and 37 GB hard disk and using Java Virtual Machine JDK 1.5.

The components include system architecture and program entities. For each project, the program files in ASCII text format were input into Rigi [39], a reverse engineering tool to re-construct the software architecture of the system. Then, we used the output of Rigi to manually create the corresponding system architecture within MolhadoArch. After that, we checked into MolhadoArch all the architecture as well as program and documentation files. The import time was recorded. Since MolhadoArch's versioning is fine-grained, documents of a system are parsed into trees and graphs, and saved in our persistent format. The import time also includes parsing and saving time. Notice that the system's size (stored in our persistent format) is about 3 to 5 times larger than its external size. We randomly made 10000 changes, each of which modified about 10 bytes of the textual slot of a node, and then measured the time to commit changes. Note that commit time for larger systems is slower due to the overhead to maintain more structures. In brief, the results show that the performance and time efficiency is satisfactory. Storage cost is high relative to ASCII text. Given the current trend that disk space is getting cheaper, we believe that the benefits gained by being able to track fine-grained, structural changes for any logical unit and architectural components, far outweigh the extra space.

B. Experiment on Compactness

In this experiment, we compared the compactness and expressiveness of change representation in MolhadoArch with those produced by a representative of traditional SCM tool, CVS [4].

Firstly, we checked out of Eclipse CVS repository three revisions of org.eclipse.ltk.core.refactoring. This subcomponent is the core of the refactoring engine in Eclipse. These revisions are tagged in the Eclipse repository at 01/31, 02/28/ and 03/29 2006. Table II shows how the source code evolved along this time interval. Even though the total number of lines of code does not reveal a great number of changes, the component passed through a great deal of changes revealed by the number of individual lines of code changed. We examined those changes and manually reconstructed architectural changes in term of UML elements. For example, between versions 01/31 and 02/28, we found out several structural changes: **four classes moved** to other packages, **one class** was **renamed**, **five classes** were **deleted** and **five totally new classes** were **added**, **four methods** were **renamed** and **four changed their signatures**, **one method moved** to another class. Between versions 02/28 and 03/29, most changes are edits, e.g., all the classes changed their copyright notice.

MolhadoArch correctly retrieves the history of classes and methods renamed or moved in the 02/28 version, while CVS loses their history. In addition, browsing through the history with MolhadoArch reveals those aforementioned architectural changes and editing changes, thus offering a higher-level understanding of software evolution. On the other hand, CVS shows a lot of changes scattered throughout the textual files, with no connection between them. For example, CVS reported that there were 1786 LOCs that has been changed from 01/31/2006 to 02/28/2006 (see Table II).

IX. CONCLUSIONS

Managing architectural evolution in a software system is crucial for software development. Approaches for software architectural evolution in ADLs are moving toward model-oriented, describing the architectural evolution at different levels of abstraction. However, architectural SCM supports are still *file-oriented*, disregarding the underlying semantics of architectural description files. Thus, the mismatch creates several problems for architectural evolution models.

This paper has shown the feasibility and the advantages of a model-oriented, architectural SCM tool that is capable of managing the architectural evolution at differ-

© 2008 ACADEMY PUBLISHER

ent abstraction models. Architectural entities at multiple levels and relationships among them are managed in a cohesive manner. In MolhadoArch, versions are uniformly maintained at all levels of abstraction including the architecture and implementation levels. Developers never have the version mismatch problem between architectural objects and source code that realizes them.

REFERENCES

[1] D. Garlan and M.Shaw, "An introduction to software architecture," *Advances in Software Engineering and Knowledge Engineering*, 1993.

[2] N. Sadou, D. Tamzalit, and M. Oussalah, "How to Manage Uniformly Software Architecture at Different Abstraction Levels," in *Proceedings of the International Conference on Conceptual Modeling (ER 2005)*. Springer Verlag, 2005, pp. 16–30.

[3] B. Westfetchtel and R. Conradi, "Software Architecture and Software Configuration Management," in *Proceedings of the Software Configuration Management Workshop*. Springer Verlag, 2001.

[4] T. Morse, "CVS," *Linux Journal*, vol. 1996, no. 21es, p. 3, 1996.

[5] "Subversion.tigris.org," http://subversion.tigris.org/.

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

[7] P. Feiler, "Configuration management models in commercial environments," Software Engineering Institute, Tech. Rep. CMU/SEI-91-TR-7, 1991.

[8] P. Ingram, C. Burrows, and I. Wesley, *Configuration Management Tools: a Detailed Evaluation*. Ovum Limited, 1993.

[9] S. Dart, "Concepts in configuration management systems," in *Proceedings of the 3rd International Workshop on Software Configuration Management*. ACM Press, 1991.

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

[11] "Rational Software," http://www.rational.com/.

[12] P. Software, *System Architect*. McGraw-Hill, 2000.

[13] H. Christensen, "The Ragnarok software development environment," *Nordic Journal of Computing*, vol. 6, no. 1, January 1999.

[14] 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 Verlag, 1999, pp. 100–122.

[15] J. Gergic, "Towards a versioning model for component-based software assembly," in *Proceedings of 19th International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2003.

[16] H. Volzer, B. Atchison, P. Lindsay, A. MacDonald, and P. Strooper, "A tool for subsystem configuration management," in *Proceedings of 18th International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2002.

[17] R. Prieto-Diaz and J. Neighbors, "Module interconnection languages," *Journal of Systems and Software*, vol. 6, no. 4, 11 1986.

[18] J. Estublier, "Workspace management in software engineering environments," in *Proceedings of the 6th International Workshop on Software Configuration Management (SCM-6)*. Springer Verlag, 1996.

[19] P. Clements, "A survey of architecture description languages," in *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996.

[20] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor, "A language and environment for architecture-based software development and evolution," in *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*. IEEE Computer Society Press, 1999, pp. 44–53.

[21] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using Rapide," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, Apr 1995.

[22] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, Apr 1995.

[23] E. Tryggeseth, B. Gulla, and R. Conradi, "Modeling systems with variability using the PROTEUS configuration language," in *Proceedings of the 5th International Workshop on Software Configuration Management (SCM-5)*. Springer Verlag, 1995.

[24] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronic software," *IEEE Computer*, vol. 33, no. 3, 2000.

[25] 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 International Conference on Software Engineering (ICSE 2002)*. ACM Press, 2002, pp. 266–276.

[26] A. van der Hoek, "Design-time product line architectures for any-time variability," *Science of Computer Programming, special issue on Software Variability Management*, vol. 53, no. 3, pp. 285–304, 2004.

[27] F. van der Linden, "Software Product Families in Europe: The Esaps and Cafe Projects," *IEEE Software*, vol. 19, no. 4, pp. 41–49, 2002.

[28] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic, "Taming architectural evolution," in *Proceedings of the 9th ACM SIGSOFT Foundations of software engineering*. ACM Press, 2001, pp. 1–10.

[29] J. Kuusela, "Archirectural evolution," in *IFIP Conference on Software Architecture*, 1999.

[30] L. Murta, A. van der Hoek, and C. Werner, "ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links," in *21st IEEE International Conference on Automated Software Engineering (ASE'06)*. IEEE Computer Society, 2006, pp. 135–144.

[31] Luqi, "A Graph Model for Software Evolution," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 917–927, 1990.

[32] T. N. Nguyen and E. V. Munson, "The Software Concordance: A New Software Document Management Environment," in *Proceedings of the 21th International Conference on Computer Documentation (ACM SIGDOC 2003)*. ACM Press, 2003, pp. 198–205.

[33] T. N. Nguyen, "Model-based Version and Configuration Management for a Web Engineering Lifecycle," in *Proceedings of the 14th International World Wide Web Conference*. ACM Press, 2006, pp. 437–446.

[34] "Viewing CVS Repositories," viewcvs.sourceforge.net/.

[35] "Jgraph," www.jgraph.com.

[36] "JTidy," sourceforge.net/projects/jtidy.

[37] "JExt," www.jext.org/.

[38] "JEdit," www.jedit.org/.

[39] M.-A. D. Storey, K. Wong, and H. A. Miller, "Rigi: a visualization environment for reverse engineering," in *Proceedings of the 19th international conference on Software engineering*. ACM Press, 1997, pp. 606–607.

© 2008 ACADEMY PUBLISHER