Automatic Generation of Version Control Systems

E. James Whitehead, Jr., GuoZheng Ge, Kai Pan
Dept. of Computer Science
Baskin Engineering
Univ. of California, Santa Cruz
{ejw, guozheng, panai}@cs.ucsc.edu

Abstract

We describe Bamboo, a system capable of generating working version control systems. Bamboo takes as input a specification of a version control system’s data model expressed using containment modeling, the pattern used to represent version histories, and choices concerning fine-grain version control behavior. Output is generated C language source code for a working version control system and textual command line shell for interacting with the system. Key contributions include the first ever version control generator, and taxonomies for patterns of representation of version histories, and version control options. The paper presents version control capabilities as a semantic layer added to containment data models, a unique separation of these design concerns.

1. Introduction

Version control facilities lie at the heart of Software Configuration Management (SCM) systems, and are also found in a wide range of document and content management systems. These systems look to version control to provide control over change, persistently storing important prior states of source code and documents as revisions, recording the sequence of creation of these revisions (revision history), noting where multiple variants existed at a given time (branches) as well as reverting to (or just viewing) previous revisions. The Source Code Control System (SCCS) [15] and the Revision Control System (RCS) [16] are two pioneering version control systems designed to provide change management for software developers.

Today, most information systems with version control capabilities support the library metaphor of checking out a file to work on it, making changes, and then checking the file back in, making it read-only. Though there is broad agreement on the checkout/checkin process, there remains substantial variation in how version histories are represented in each system’s repository. Some systems, like SCCS and RCS, have a version history abstraction that holds all of the revisions of a particular document. Others, like ClearCase [21,12], have an abstraction that explicitly represents branches, which hold revisions, and a separate abstraction that acts as the version history by aggregating all branches for a given document. Some systems do not have an explicit version history abstraction, either using link abstractions to represent predecessor and successor relationships (PCTE [9]) or having no additional abstractions beyond the revision, implicitly recording predecessor and successor information in a version identifier (e.g., 1.1, 1.2), as is the case for Shape/AtFS [13]. These systems all share the quality of being state-based, supplying an abstraction to represent revisions, rather than change-based, where changes are explicitly represented [6].

Similar variation exists in the constraints concerning interactions with version histories. Many document management systems only permit linear version histories, with branching forbidden, while most SCM systems allow branching. Some systems permit checking out from a previous revision, while others limit checkout to the latest revision. Even the effect of checkout varies, with some systems creating a new, writable object in their repository on checkout, while others, like RCS, create a writable object outside the repository, creating a new, read-only object in the repository on checkin.

Additionally, there are variations in the kind of repository used for representing revisions, branches, and version histories. SCCS and RCS use the filesystem, storing each version history and associated versions in a single file. SCM systems with significant functionality tend to use a database, taking advantage of its searching, transaction, and consistent backup facilities. Most repositories are centralized (SCCS, RCS), while others can be distributed (Adele [8], NUCM [17]). Repositories tend to employ some form of inter-revision compression, usually some form of delta, choosing from several available options [11].
Designers of version control capabilities face a multidimensional design space where they need to make decisions about which options they will select. This involves analysis of the options for how to represent a version history, which repository to use, and what use constraints to impose, along with issues like the user interface and metadata facilities. In the literature, information about design choices and tradeoffs is currently fragmented, spread across multiple conference and journal articles, as well as book chapters and system documentation, requiring significant effort to acquire and study. Even existing survey work, which is thorough and comprehensive, does not capture all of the necessary choices, or structure it in the form of design tradeoffs [6].

Ideally we would like designers to be able to view all of the version control design issues in one place, and then select capabilities from a palette of known options. It should be easy to explore different combinations of options to better understand tradeoffs among them. To this end we have constructed a system that takes as input the characteristics of a repository and desired version control options, automatically generating a version control repository, API, and command-line shell matching that specification. The system is named Bamboo, after this plant’s quality of rapid growth.

Bamboo has several uses. First, designers and educators can rapidly explore, learn, and teach different options for creating version control systems. Instead of various options being discussed only on paper, they can quickly be realized into an interactive, working system. The architecture of generated systems has a programmatic API separating the repository and user interface layers, and permitting a full-featured version control user interface can be built on top. As a result, a full content management system could more rapidly be constructed on the generated repository core than if it were completely written from scratch. Bamboo could also be used to tailor a version control system to match the needs and processes of an individual project or organization. Many organizations today have a source code control system that is built from a series of shell scripts built on top of a version control system, and Bamboo could be used as a more customizable and more easily integrated versioning core for such systems.

Of course, if it is possible to automatically generate version control systems, it should also be possible to generate the full range of facilities supported by SCM systems, including configuration control, logical change tracking, and workspace management. Furthermore, other kinds of content management capabilities should also be generable, such as hypertext links and compound documents, thereby extending the scope of Bamboo beyond strict SCM boundaries. These are goals for our future work.

2. Layering Version Control Semantics onto Containment Data Models

Prior to starting development on the generator, we performed a survey of a broad range of SCM, hypertext, and hypertext versioning systems (the latter two categories can be viewed as specialized forms of content management systems with rich hypertext linking capabilities). A recurring pattern visible in these systems was that of system abstractions containing other abstractions. For example, a version history contains all of the revisions in that history, or, in a system with first-class branches, a version history contains branches, which then contain revisions. In these examples, the containment relationship takes the form of an identifier pointing to the contained item, with, say, a version history holding a list of identifiers for the contained revisions. This is referential containment, where the container refers to the containee.

It is typical for revisions to have associated metadata such as the checkin time, version identifier, a brief comment, etc. Each metadata item is considered distinct in that modifying metadata does not affect revision contents. However, deleting a revision eliminates its associated metadata. Since the metadata items have individual names, and are capable of being read and manipulated, they have sufficient independence to warrant consideration as separate system abstractions. However, this independence is limited due to deletion semantics, which behaves as if the metadata items are physically part of the revision. We model this situation as a revision inclusively containing the metadata items.

In prior work we have examined the data models of 27 SCM, hypertext, and hypertext versioning systems, generating models of these systems that incorporate their significant abstractions, and the containment relationships (referential and inclusion) among them [23, 22, 10]. This modeling exercise can be viewed as employing a deliberately constrained form of entity-relationship modeling where the entities are abstractions found in system data models (e.g., revisions, version histories, branches, metadata items), and the relationships are forms of containment (referential or inclusion) [3]. Another perspective is that containment models are a restricted form of UML structure diagram where the classes represent abstractions in the system data models, and the only relationships are simple (referential) and composite (inclusion) aggregation [2]. Classes are restricted to hold just a single data item, since they are used to model abstractions like a document’s content (a text chunk), or a metadata item.

An example of a containment data model is shown in, Figure 1, which describes the RCS system. In this diagram, system abstractions that act as containers are
Figure 1 - Containment data model of the RCS system.

depicted as circles, while non-container (atomic) abstractions are squares. The RCS history file (representing an RCS .v file) acts as a version history, containing all of the revisions of a particular file. This containment relationship is inclusive (a solid line), since deletion of the revision history results in deletion of all revisions. A revision is shown as a container, inclusively containing all of the metadata items associated with the revision, including the content of the revision (revision text), creation timestamp, version identifier, author, comment, etc., which are atomic. Successor relationships are modeled as referential containment relationships (dashed lines) among revisions, reflecting that each revision keeps a list of version identifiers (references) of its successors.

Limiting allowed relationship types to just kinds of containment is intentional, and yields several benefits. The first of these is simplicity. The data models of SCM systems can be quite complex, with many interactions among data model abstractions. By limiting the amount of information depicted, it is possible to capture the entire data model in a single diagram. Additionally, the containment relationships capture many important qualities about system data models. Containment models can be used to develop scenarios involving multiple instances of all system abstractions, difficult to accomplish from just an inheritance hierarchy. Many questions concerning SCM data models involve consistency management issues, such as, “if we create/delete an instance of X, what other abstractions need to be updated?” Since containment models involve a minimal set of entity and relationship types, they carry very little modeling baggage, and hence permit uniform comparison of the data models of a broad range of SCM and content management systems (e.g., as shown in [23,22,10]).

However, having reduced system data models to just containers, atoms, and containment relationships, there is no longer sufficient information to generate a version control system, since the generator is unaware of which container is acting as a version history or a revision. This information must be conveyed to the generator. We view the process of mapping entities in the containment model to specific version control roles (version history, revision, metadata item, etc.) as one of overlaying version control semantics onto the containment model. This notion of a semantic overlay is useful, since it allows major design concerns in the construction of SCM and content management systems to be separated into distinct overlays. So, the design issues concerned with version control capability can be isolated in one overlay, while workspaces, configurations, and hypertext linking can each be a separate overlay. This permits system designers to study and tweak individual design concerns in relative isolation from each other, and to develop better understanding of the interaction of concerns.

Systems can be constructed and understood as stacks of overlays, and hence, for example, a new content management system might be the combination of version control and workspace overlays. Multiple overlays might affect the same abstraction, with a hypertext versioning system having a given container entity affected by the link and versioning overlays, thereby becoming a versioned link. While this composition of overlays still remains future work, this goal of composable layers motivates the separation between containment model and the version control model we overlay.

The version control overlay implemented by the current Bamboo system contains two parts: a versioning options specification, and a mapping of versioning roles to entities in the containment data model. The versioning options specification involves a choice from among a list of patterns—each derived from an examination of existing version control systems—for how to represent version histories. Each pattern contains a set of entities and containment relationships. For example, the Ordered Revision pattern, based on the RCS system, has a version history containing revisions (but no first class branches). Since the containment data model being layered onto may have many entities, it is necessary to identify which containment model entities correspond to given roles in the version model pattern. The generator needs to know which entity will play the role of a revision, and of a version history. Similar mapping is needed for metadata items; we need to know which entity will take responsibility for recording the version identifier, comments, checkin time, etc.

In the next section, we describe the set of known patterns for representing version histories, and then enumerate the possible roles entities may play in these patterns. In Section 4, we describe a set of design options
that can be specified to the generator, and in Section 5 we describe the architecture of the Bamboo generator, and the systems it creates. Related work and conclusions conclude the paper.

3. Versioning Patterns and Roles

3.1 Version History Patterns

After surveying existing state-based SCM systems, we classify versioning models into five generic patterns:OrderedRevision, FirstClassBranch, PredSuccSet, FloatingObject, and LinkedRevision. Each versioning pattern has specific operations and semantics for entities and relationships in the containment model. Choosing a particular versioning pattern is the first step in overlaying version control semantics on top of a containment model. As an overview of the five patterns, Figure 2 shows a version history tree for the file foo.c described using each pattern.

The **OrderedRevision** pattern, Figure 2(b), describes systems like RCS, SCCS, PVCS [14], and CVS [1]. It has two containers, version history and version, each with several predefined atoms describing metadata information. The version history inclusively contains a series of versions. Both version history and version entities have a series of predefined metadata items, which are atomic entities that record versioning-specific metadata such as a revision identifier, comments, etc. The version entity also has a referential relationship with itself, used to represent the predecessor/successor relationships among versions.

The **FirstClassBranch** pattern, Figure 2(c), describes systems where branches are treated as first class entities, as is the situation with ClearCase. In FirstClassBranch systems, the branch is explicitly represented, and users have to explicitly create or destroy a branch. This contrasts with other systems where branches are represented by predecessor/successor relationships, or in the version identifier naming conventions. For example, in RCS 1.0.1 is considered to be a branch from trunk 1, and this branch is automatically created when users check in using explicit version identifier 1.0.1.1 [16]. In the FirstClassBranch pattern, a version history referentially contain branches, and branches referentially contain versions.

The **PredSuccSet** pattern, Figure 2(d), describes systems like DeltaV [5]. This pattern is similar to OrderedRevision, adding an additional version controlled resource entity as a container of the version history. In the DeltaV standard, the version controlled resource acts as a handle for operations on the version history, accepting operations like checkin/checkout and reads/writes. Since this is distinct from the version history, multiple version controlled resources for the same version history can be used to populate multiple workspaces.

The **LinkedRevision** pattern, Figure 2(e), describes systems that do not have a first class version history, instead using first-class predecessor/successor links to represent a version history tree. Since a link has two endpoints that are essentially references to versions, we treat a link as a container. In this pattern, the link is a container that referentially contains versions thereby representing predecessor and successor relationships. One example of the LinkedRevision pattern is PCTE [19].

The last pattern, **FloatingObject**, Figure 2(f), is comprised of only a set of versions and their associated...

---

**Figure 2.** A version history tree (a) as represented by different version history patterns (b-f).
metadata items. The predecessor/successor relationships among versions are expressed using a specific naming convention for version identifiers. Hence, if two versions have the name “foo.c,” then the one with identifier “1.2” (stored in a metadata item) is by definition the successor of “1.1.” In order to retrieve a specific version, whole version space needs to be searched, and hence this pattern is only usable with a centralized repository. An example of the FloatingObject model is Shape/AtFS [13].

<table>
<thead>
<tr>
<th>Versioning Role</th>
<th>Meaning and Usage</th>
<th>Maps to RCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_revision</td>
<td>Entity representing a version in a containment model, the container for versioning metadata items (atoms)</td>
<td>revision</td>
</tr>
<tr>
<td>s_rev_content</td>
<td>Textual or binary content for a revision</td>
<td>revision text</td>
</tr>
<tr>
<td>s_rev_identifier</td>
<td>Unique identifier for each revision</td>
<td>version identifier</td>
</tr>
<tr>
<td>s_rev_checkin_timestamp</td>
<td>Checkin timestamp</td>
<td>timestamp</td>
</tr>
<tr>
<td>s_rev_comment</td>
<td>User comments submitted with each revision</td>
<td>comment</td>
</tr>
<tr>
<td>s_rev_author</td>
<td>Author for each revision</td>
<td>author</td>
</tr>
<tr>
<td>s_rev_history</td>
<td>Version history entity in containment model, usually contains for revisions but may contain branches</td>
<td>RCS history file</td>
</tr>
<tr>
<td>s_rev_history_identifier</td>
<td>Identifier for locating the revision history</td>
<td>filepath (implicit)</td>
</tr>
<tr>
<td>s_rev_history_description</td>
<td>Textual description for the entire revision history</td>
<td>description</td>
</tr>
<tr>
<td>s_rev_lock</td>
<td>Storage for per-version lock information</td>
<td>locks</td>
</tr>
<tr>
<td>s_rev_trunktip</td>
<td>Latest version in the repository on the main trunk</td>
<td>head</td>
</tr>
<tr>
<td>s_version_controlled.rev</td>
<td>Version controlled resource entity in containment model</td>
<td></td>
</tr>
<tr>
<td>s_col</td>
<td>Collection revision entity in containment model (similar to file system directory), usually a set of version entities</td>
<td></td>
</tr>
<tr>
<td>s_col_url</td>
<td>Identifier for accessing collection revisions</td>
<td></td>
</tr>
<tr>
<td>s_col_identifier</td>
<td>Unique identifier for each collection revision</td>
<td></td>
</tr>
<tr>
<td>s_colhistory</td>
<td>Collection version history in containment model</td>
<td></td>
</tr>
<tr>
<td>s_col_history_description</td>
<td>Textual description associated with the entire revision history</td>
<td></td>
</tr>
<tr>
<td>s_col_author</td>
<td>Author for each collection revision</td>
<td></td>
</tr>
<tr>
<td>s_version_controlled_col</td>
<td>Version controlled collection entity in containment model (version controlled resource, but not for collections)</td>
<td></td>
</tr>
<tr>
<td>s_col_lock</td>
<td>Lock information for collection revisions</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Versioning roles. Gray shaded classes correspond to container entities, and the others map to atomic entities (metadata).

3.2 Versioning Roles

Version patterns are the first half of our approach to overlay semantics on top of containment models; the other half is mapping entities in a containment model to the versioning roles found in a pattern. Assigning roles to entities informs the generator about what data and operations that entity will support in the generated system. For example, an entity assigned to the version history role will have an operation for listing the version history added to it by the generator. Similarly, in order to implement the checkin command, the generator needs to know where to place the comments, version identifier, timestamp, and author information in containment model entities. Versioning roles give this information.

Table 1 shows all versioning roles found in existing patterns. The table additionally shows how these versioning roles are mapped onto abstractions found in the RCS containment model in Figure 1.

4. Versioning Options Taxonomy

In addition to choosing a particular version history pattern, versioning system designers working with Bamboo must also make fine-grain choices concerning the behavior of checkin and checkout, whether locking is supported, if collections are versioned, and if delta compression is used. For example, designers can choose linear or branching for version history topology, locking or non-locking for concurrency control. We call these choices versioning options.

Versioning options are designed as a couple of switches to tune the versioning functionalities for a generated SCM system. By selecting values for each option, users precisely define versioning semantics in a straightforward way. Some combinations of option values can be nonsense or conflicting, e.g. allowing checkin at nontip location (which creates branches) and allowing checkout at tiponly location (which only checks out tip revision of the trunk) at the same time. These conflicting option values are identified and predefined as illegal. The following is a list of versioning options we have designed.

Pattern indicates which version history pattern a version control system is using. The values can be selected from OrderedRevision, FirstClassBranch, PredSuccSet, FloatingObject, and LinkedRevision (as described in Section 3.1).

Checkin Position identifies where in the version tree a newly created revision can be checked in. It has two possible values: tipOnly and allowNonTip. TipOnly means that when a user checks a new revision into repository, a new revision will only be created at the tip of the version tree. AllowNonTip means the user has the option to check in at a particular position on the version tree, not necessarily at the tip. Checking in at non-tip position will create a branch (or sub-branch) of the version tree. One consequence is that Checkin Position also controls whether the system allows branching. TipOnly means “non-branching” (linear) and allowNonTip means “branching allowed”.

Checkout Arguments are the parameters that select the revision a user wants to check out. The most typical and default one is a version identifier. Other possible checkout arguments are a timestamp (latest revision prior to this
timestamp), and user name (last checked-in revision by this user).

**Checkout Position** has same value set as Checkin Position: tipOnly and allowNonTip. It is used to tell where a user is allowed to check out a revision from within the version history. TipOnly means a user can only check out the tip revision from the trunk, or the latest revision across multiple branches. AllowNonTip means a user can check out any revision in the version tree. It is a conflict to combine the allowNonTip value for Checkin Position with the tipOnly value for Checkout Position since it is useless to check in at any place on the version tree but be able to only check out the tip version on the trunk.

**Locking** shows if locking mechanism should be supported as a concurrency control mechanism, true or false. Currently, we only implement exclusive locks. A true value for the locking option means that an exclusive locking protocol should be used on revisions. The exclusive lock requires that in order to have write access to a particular resource, a user must obtain a lock on that resource. If he cannot get the lock, i.e. another user has locked the resource, he can only checkout a read-only revision. A false value means that no locks should be used: each user can check out a resource with write access and when he checks in he has to merge his changes with the latest revision in the repository (because other users may have checked in new revisions since this user checks out). In future work we intend on extending this locking model to encompass a richer array of lock and merge characteristics.

**Scope** indicates whether file system directories (or entities acting like the same) can be versioned in the system. This option has two values: revisionOnly and revisionAndCollection. If the value is revisionOnly, then only revisions (plain files) can be versioned; otherwise, both revisions and collections (plain files and directories) can be versioned.

**Compression** decides whether and how we should apply compression. There are four choices for this option: tipOnly, deltaOnly, tipAndDelta, complete, and no. TipOnly means we only compress the tip version, which stores the complete content for latest revision. DeltaOnly means we only compress the deltas. TipAndDelta applies compression to both latest revision and deltas. Complete is used when no delta storage is used and we keep every complete revision and apply compression to them. No means no compression is used.

**Storage Method** indicates how we store every revision. There are three possible values for this option: complete, forwardDelta, and reverseDelta. Complete means each revision is stored in its completeness; this choice has the advantage of fast speed of retrieval, but consumes much storage space. ForwardDelta keeps the complete root revision, and stores subsequent changes from root revision in deltas (as does SCCS [15]). ReverseDelta keeps a complete copy of the tip revision, and stores backward changes from the tip as deltas (RCS [16]). Deltas save storage space, but slow down retrieval speed because we have to compute content from deltas every time we check out an old revision.

5. SCM System Generator

A software generator is a program that translates domain specific languages or specifications into application [4]. The Bamboo system takes as its inputs code templates and specification documents that characterize the containment model, repository model, versioning options, and version role mapping for a desired version control system. The outputs of the Bamboo generator are the source code of a version control system and a Makefile to build it. Also, the physical storage for this SCM application is created during the generation process, based on a MySQL database. In the remainder of this section we describe the Bamboo generator, describing its architecture, the input files, and the generated system.

5.1 Architecture of Bamboo generator

The Bamboo generator consists of four major parts (depicted in Figure 3): an XML parser, a physical repository creator, a modeling code generator, and a template file processor. The XML parser analyzes the documents holding the containment model, repository description, versioning options, and the version role mappings, checks the consistency of these documents, and converts them into internal data structures. Based on the specifications given in the repository description document, the physical repository creator creates and initializes the MySQL database used for the version control system’s physical repository. The **modeling code generator** is responsible for generating the source code files according to the containment model and the version role mapping documents. The **template file processor** parses the template source code files and generates output source code according to the options specified by the versioning options document.
5.2 Inputs to the Generator

Bamboo requires six inputs to generate a version control system: user-generated documents specifying the containment model, repository description, versioning role mapping, versioning options, along with system-provided files giving source code templates, and a Makefile template. These input files are described below.

**Containment Model Document.** In previous work on containment modeling, we only provide a graphical representation of containment data models since the goal was communication to a human audience [23]. To make a machine-readable containment model, we created an XML structured document. XML containment model documents have three required elements under the root element model: entities, relationships, and er_model. Entities define all the entities in an SCM system, which can be sub-element container or atom. Relationships define relationship types that are supported in an SCM system. There are four possible relationship types, the result of four combinations of containment type (referential or inclusive) and ordering attribute (ordered or unordered) for a relationship. Each type is represented as a relationship sub-element under relationships element. Er_model defines all existing relationship instances (arcs) in an SCM system. Each relationship is represented by an arc sub-element.

**Repository Description Document.** The repository description document maps the logical data types in the containment model to the physical repository data types, e.g. a logical string maps to a physical VARCHAR(255) in a MySQL database. With the help of this document and the containment model document, database tables can be created in the generation process.

**Versioning Options Document.** This document allows a designer to specify their choices for the version control options discussed in section 4. In this XML structured document, each versioning option is defined as an element that records the value of an option specified by the user.

**Versioning Role Mapping Document.** As described in Section 3.2, this XML structured document represents the mapping of entities in the containment model to roles found in a given versioning pattern.

**Code Template Files.** The code template files are C source code files with generation tags in the source code. The generation tags represent variable aspects of the source code that is dependent on the specification for a given version control system. The generator replaces tags with generated source code.

**Makefile.** The Makefile is used to build the version control system from the generated source code.

5.3 Generation Process

Bamboo’s code generation process is as follows. First the user defines the input XML documents: containment model description, repository description, versioning options, and versioning role mappings. Next, the XML file parser parses the input XML documents, checks their consistency, and converts all the information into internal data structures. After that, the generator creates database tables according to the repository description document. Finally the generator processes the code template files and generates the resulting source code according to the input documents.

A major task of the generator is to transform the code template files into the desired target source code file for a version control system. The code template files can be classified into two categories. First are those that can be just duplicated to the target source code files without any change. Most C header files and some “static” C source code, such as the repository API code, fall in this category. The other category includes code template files with generation tags in them. The generation tags represent the dynamic parts of an SCM system that is dependent on different containment models, versioning role mapping and versioning options.

The versioning options document illustrates how the Bamboo generator processes the code template files. There is a checkin-position option in the version options document. Possible values for this option are "tipOnly" or "allowNonTip". A pair of generation tags, "/*tagBamboo_begin_option_checkinposition<tripOnly>*/" and "/*tagBamboo_end_of_option*/", in the version.c code template file are related to this option. The code between
these two tags takes effect only when the value of the checkin-position option is tipOnly. So, when the Bamboo generator processes the code version.c template file, it keeps the code between these two tags if the “checkin-position” option value is “tipOnly”, otherwise, it skips the code between these two tags. Bamboo generator handles the other tags in the similar way for each of the code template files. The results of the generation process are the generated source code of an SCM system.

5.4 Architecture of Generated Systems

Following the generation process, the Bamboo generator produces source code for SCM systems. There are four layers for the generated system: repository layer, repository API layer, logic layer, and front end. Figure 4 shows the architecture of the generated system. The repository layer physically stores data about entity and relationship instances. The repository API defines methods to create, destroy, and retrieve data from the physical repository. The logic layer stores the definition of entities and relationships between them. It also defines version control methods using the repository API and enforces constraints defined in the containment model, such as multiplicity constraints. For example, if a container has a 2..N containment relationship with a containee, then every time the container is created, it must have two containees as well. The front end is the interface between users and the generated SCM systems. It can either be a textual command line shell program (as we have now) or a more complicated GUI-based client, which would need to be written after Bamboo has completed (Bamboo does not currently generate GUIs).

The repository layer is responsible for persistent storage for instances of entities and relationships. In the current implementation, a MySQL database is the physical repository for this layer. The database has two tables: entity table and relationship table. The entity table stores information for each entity instance: entity identifier (unique integer assigned to each entity instance), entity class name (e.g. for RCS these would include RCS history file, revision, version identifier, locks, etc.), container type (either container or atom), and value (for atomic entity instances that store content). The relationship table keeps all the information about relationship instances: relationship identifier, starting entity identifier, end instance identifier, index (for ordered relationship instances to keep ordering), and relationship type (referential or inclusive). Note that our architecture makes it possible to use other physical repositories than a MySQL database.

The repository API layer provides a standard set of methods to manipulate and access data in the physical repository layer. In another words, if we choose to use different repositories, we need to use their corresponding APIs as well. But the logic layer that calls methods in the API does not have to change at all because the set of methods are standardized. The methods in API layer provide capabilities to initialize the repository, to add or drop entity or relationship instances, and to retrieve data stored for specific entity or relationship instances.

The logic layer is built on top of the repository API layer. It stores definitions for each entity class and each relationship obtained from the containment model. This layer also implements various version control methods, such as checkin and checkout. A constraint checker is also inserted in this layer to ensure all the constraints defined in containment model, e.g. limits to membership and cardinality, are strictly obeyed and the system is kept in a consistent state.

The front end is the interface of version control system exposed to users, enabling them to interact with the system. Currently we’ve implemented a textual command line shell that accepts commands from the users, operates on the repository and then returns results. Currently, we only provide checkin and checkout functions. Later, more complex features will be added as well.

5.5 Implementation Status

The current implementation of the Bamboo system, while not complete, still demonstrates the majority of the novel aspects of our approach. At the lowest level, Bamboo is capable of generating a wide range of executable containment data models, including a complete API layer and textual interactive shell. When adding version control capabilities to these containment models, we currently support only the OrderedRevision pattern,
and the checkin-position, checkout-position, and locking options. This provides an initial proof-of-concept validation of the ability to layer version control semantics on top of a containment model. We are actively pursuing future work that shows composition of multiple design spaces, as well as greater depth in supported version control patterns and options.

6. Related Work

There are three existing systems that closely match our work, NUCM [18], CoMa [20], and DAMOKLES [7].

NUCM provides a generic repository whose native data model includes collections that referentially contain atoms, and hence is very similar to containment models, though lacking some containment model specifics (e.g., ordered collections, inclusion containment). To create a new version control system, a designer uses NUCM’s API and writes code that interacts with the NUCM repository. While this provides significant flexibility in the construction of version control capability, it has the drawback of embedding design decisions into the source code. This makes it more difficult to explore different version control options, or different patterns for representing version histories, since the exact choices are embedded in the implementation, and must be changed to explore different options. The Bamboo approach allows system designers to work with ideas and abstractions that are closer to their problem formulation as a set of desired version control capabilities, rather than having to translate those capabilities into source code.

CoMa provides a predefined data model that includes abstractions representing version histories, configurations, and first-class dependency relationships. Using CoMa, system designers define version control capabilities using the PROGRES graph rewrite language. This is essentially writing code in the PROGRES language, and hence shares many of the drawbacks of the NUCM. While PROGRES is a somewhat more natural way to represent operations on version history graphs than the C language NUCM uses, it still has the effect of embedding design choices inside PROGRES specifications. Additionally, CoMa has seven major abstractions in its data model, increasing the likelihood of mismatch between the CoMa data model and the desired data model for a new system being designed. By using the very basic containment modeling, very few data model biases are introduced.

The DAMOKLES system shares our goal of tailoring version control capabilities directly to the needs of a given project. The DAMOKLES data model has a built-in notion of revision, which is used to build up data models containing a wide range of versioned persistent data types using a version-aware data modeling language. DAMOKLES can be viewed as combining together the flexibility of our containment modeling approach with a single fixed version history pattern. DAMOKLES does not provide control over version options, though like NUCM these could presumably be supplied in source code that interacts with a DAMOKLES database.

In [6], Conradi and Westfechtel give a taxonomy for version models and classify relationships between version models and data models into three groups: version model on top of the data model, version model within the data model and data model on top of the version model. Our versioning extension to containment model belongs to the first group. Our data model is described by containment model to define entities and their relationships. The version model is described by versioning patterns, roles and options as versioning layer on top of the containment model. This way keeps data model unaware of versioning semantics and separates storage of data and semantics of data. In addition, our three-tiered design for generated systems cleanly separates storage layer, semantic layer and presentation layer of data. This makes it easy to change other SCM functionalities such as workspace and configuration management without affecting versioning functionality.

7. Conclusions and Future Work

In previous work, we developed the containment modeling technique, and used it to uniformly model the data models of a wide range of SCM, hypertext, and hypertext versioning systems [23,22,10]. In this paper, we have shown that containment models have utility that extends beyond data model survey work, and in fact can be turned into executable containment models with the use of Bamboo. Additionally, by layering version control semantics onto containment models, it is possible to automatically generate version control systems that match a specific pattern for the representation of a version history, and that implements desired fine-grain version control options. This shows that the more complex semantics found in SCM and content management systems can be added to containment models, making containment models a simple yet powerful abstraction layer upon which to automatically generate content management systems.

The taxonomy of version control patterns presented in this paper is unique in its focus on the details of different ways state-based revision histories can be represented. It, along with the versioning options described in section 4 demonstrate that it is possible to statically enumerate most common design options, and represent them declaratively. Additionally, describing versioning roles distinct from versioning options and the containment model provides a novel separation of concerns for aspects of version control
systems. Together these contribute to our knowledge of different ways to design version control capabilities.

Although our implementation is preliminary so far—the generator can only produce version control systems matching the OrderedRevision pattern (e.g., RCS)—we believe our approach holds significant promise for improving our understanding of the design choices and tradeoffs in existing SCM and content management systems. It certainly shows that automatic generation of version control systems is, in fact, possible, since Bamboo is the first system that automatically generates version control systems.

Our future work includes extending Bamboo to support more versioning patterns and versioning options, developing other SCM function layers, e.g. workspace management, and finally attacking highly complicated systems like ClearCase and DeltaV.

Acknowledgements

This project is supported by the National Science Foundation under Contract Number NSF CAREER CCR-0133991. We also want to thank Jennifer Bevan, Elias Sinderson, and Sung Kim from GRASE lab at the University of California, Santa Cruz for their helpful comments and discussion.

References


