Automatic Generation of Hypertext System Repositories A Model Driven Approach

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

ABSTRACT
In this paper, we present a model-driven methodology and toolset for automatic generation of hypertext system repositories. Our code generator, called Bamboo, is based on a Containment Modeling Framework (CMF) that uniformly describes data models for hypertext systems. CMF employs a lightweight modeling approach in which entities (system abstractions) and containment relationships are used to model hypertext system repositories. Given a description of a system repository data model using CMF, as well as a specification of the mapping between the domain specific roles (link, version history, etc.) and the entity definitions, Bamboo can generate an open hypertext repository that matches the specification. The benefits of this approach include a shorter development cycle, lower design and implementation costs, fewer design faults, a standard repository API, and extensibility for adding new features. We validate our approach by automatically generating repositories in accordance with the models of five existing hypertext systems. We also demonstrate the extensibility of our approach by quickly building a GUI client on top of a repository, and then subsequently adding version control capabilities by altering the containment model and regenerating the system.

Categories and Subject Descriptors
D.2.1 [**Requirements/Specifications**]: Elicitation Methods – *rapid prototyping*; H.2.1 [**Database Management**]: Logical design – *data models*; H.2.3 [**Database Management**]: Languages – *data description languages*; I.2.2 [**Automatic Programming**]: Program Synthesis.

General Terms
Design, Languages, Theory

Keywords
Containment Modeling Framework. Model-driven development. Automatic code generation. Hypertext data models. Open Hyperbase.

Hypertext'04, August 9–13, 2004, Santa Cruz, California, USA.

1. INTRODUCTION
Like most software development, the process of creating new hypertext systems is expensive: development for commercial systems takes several person-years; maintenance and evolution add to the cost as well. Despite the cost and effort, the resulting hypertext systems usually have the following problems: it is difficult to share resources among these systems because the underlying data models are different and often incommensurable; it is expensive to develop and maintain new hypertext systems because components and patterns are not standardized and reusable; and it is difficult to change existing systems because the refactoring process can be problematic, introducing new bugs and inconsistencies.

In previous work, we have proposed a uniform modeling mechanism, the Containment Modeling Framework (CMF) [12, 26, 27] for describing content management system data models. CMF uses a uniform set of entities, relationships and attributes to model hypertext system repositories. This framework has been successfully applied to model 27 existing hypertext, content management, and software configuration management systems [27]. Based on the CMF, we have developed a model driven methodology and a supporting toolset to automatically generate hypertext system repositories.

Our work has two major contributions: first, we developed a model driven methodology and a toolset, based on CMF, which support rapid prototyping of hypertext system repositories; second, our approach can effectively reduce the cost of extending CMF-based hypertext systems. We implemented an automatic generator called Bamboo to generate hypertext system repositories as well as repository accessor APIs. Bamboo uses template-based code generation. Recurring patterns in hypertext system development are encoded in the templates. Our generator reduces much monotonous work at the repository layer—programmers only need to build models for the repository layer and leave the tedious coding work to the generator. This enables developers to focus on more important development efforts like the domain logic and user interface (UI) design. Currently, we provide no support for automatic generation of domain logic or UI code since every system has domain specific requirements.

By using CMF we also avoid the error-prone manual refactoring process. The generator supports an easy way of code refactoring that adds new features to the systems that were developed using CMF. Users just need to change the model, regenerate the repositories, and make a minimum modification to the client code. We use an experiment to show the addition of version

control capability to a reimplementation of NoteCards [23] based on CMF.

The rest of this paper is structured as follows: Section 2 discusses related work on automatic code generation, its history, existing tools and frameworks; Section 3 introduces the Containment Modeling Framework, its basic building blocks, and the graphical and textual notations with several hypertext system repositories modeled using CMF; and Section 4 describes the details of the generator architecture and code generation process, a general architecture for the generated system is also discussed. Section 5 then uses two experiments to demonstrate use of our generator to produce hypertext system repositories, the construction of a GUI interface upon the generated repository, and an example of model-based repository extension. Section 6 concludes the paper, and illustrates possible future directions.

2. BACKGROUND AND RELATED WORK

Automatic code generation or code synthesis can trace its history back to the invention of compilers. The first FORTRAN compiler was called an automatic programming system rather than a compiler [7]. With the advantages of shorter development cycle, low implementation cost, and fewer code faults, automatic code generation quickly became popular in both industry and academia. More recently, we see various generators for the code synthesis of user interface [14], XML schema (CAIDE) [4], and statistical analysis tools (NASA AutoBayes) [10].

A recent development in modeling and code generation is meta-modeling frameworks that produce domain-specific generators based on particular domain knowledge. Examples include Model Driven Architecture (MDA) [17], GenVoca [5], and NASA's Meta-Amphion [13]. The meta-generator frameworks provide basic modeling abstractions to represent specific domain knowledge and create domain models and generators. The domain-specific generators then produce actual systems based on the specifications. Automatic code generation and component reuse play an important part in these meta-modeling frameworks. Our CMF and Bamboo generator is not a meta-generator framework; it is a domain-specific modeling framework for the content management domain. But we share the same philosophy of design-by-model, automatic code generation and component reuse.

We believe our model driven approach to be the first attempt to automatically generate the repository layer for hypertext systems. The Entities and Relationships on the Web (ERW) project [24] employs an Extended ER (EER) model to specify database schemas and automatically generates complex relational database systems with GUI Web clients. It is one of the research ideas most similar to ours. The objective of ERW is to automatically generate relational databases that can be managed through a Web interface. Our focus is to generate a hypertext capable content management repository layer using a uniform modeling framework for rapid prototyping of hypertext systems. In our modeling specification, we encode not only the repository schema, but also the semantics and operations for the data in the repository. The implementation of Bamboo's repository layer is independent of storage type: we can use file system, relational database, object-oriented database or XML data store as long as they implement the standard accessor API. Our modeling framework also provides capability for feature extension, examples being version control by using hypertext overlays.

3. CONTAINMENT MODELING FRAMEWORK

CMF is the modeling framework for constructing containment models for specific content management domains such as hypertext systems. A containment model is the model definition for the repository schema of a particular system. The actual data stored in the repository is an instance of the containment model for each hypertext system.

Containment models emphasize the containment and storage relationships among entities in a given content management system. Directed by the design principles of simplicity and neutrality, there are only two basic modeling components in CMF, namely *entities* and *relationships*. Using these two building blocks, we can graphically and textually describe and compare hypertext system data models. The actual data in the repository consist of entity instances and relationship instances. The framework can also be extended to provide feature aspects via a feature overlay mechanism.

3.1 Basic Components: Entity and Relationship

Entities represent data abstractions in a hypertext system. There are two basic entity types: *atom* and *container*. Atoms are the lowest level of abstraction. They cannot be divided, and hence atoms cannot contain other entities. Atoms often represent file content, attributes, or metadata items like author, last modified date, etc. Containers are used to form containment structures consisting of atoms or other containers. These structures are often hierarchical, but may have other topologies as well. Examples of containers can be Web pages that have text chunks, images, and hyperlinks, or project folders that contain sub project folders and memos.

Relationships represent the connection between entities and express the constraints between containers and containees (we refer to one end of a containment relationship as the container, and the other end as the containee). There are two types of containment relationships: *referential* and *inclusive*. Referential containment means that the container and containee are physically stored at different locations and the container only stores a reference or pointer to the containee. In contrast, inclusive containment indicates the containee is stored as part of the container, which is similar to composition in UML [16]. The difference between two relationship types is highlighted by examining the typical "delete" semantics on a container—when a container is deleted, the containee is deleted as well for inclusive containment; but for referential containment, only the reference is removed and the containee remains unchanged.

Relationships have inherent modeling properties of *ordering* and *multiplicity* (namely either *membership* or *cardinality*). The ordering property for the relationship identifies whether there is a partial ordering on the relationship instances. As an example: suppose a containment model defines an inclusive relationship from a project folder to several meeting memos. If the ordering property has the value "true", all the meeting memo instances

will have a partial ordering for any project folder instance that contains them. Otherwise, there is no ordering. In our implementation, ordering is realized by using a special index on each relationship instance.

Multiplicity defines the upper and lower bound for the number of occurrences of entity instances on both ends of a relationship. Membership restricts the number of container instances (of a particular type defined on the relationship) that one containee instance can belong to; cardinality restricts the number of containee instances (of a particular type defined on the relationship) that one container can have.

Using the same example of a project folder and meeting memos, we can set the membership to 2, which means one meeting memo instance must be contained by exactly 2 project folder instances, e.g. one project folder instance as the official copy and the other as backup; if we set the cardinality to 1..b, one project folder instance should have at least 1 meeting memo instance, and no more than b memo instances. If the special letter n is used in the place of b, it means the upper bound is an unlimited natural number. For more discussion on the containment modeling framework, please refer to [12, 26, 27].

3.2 Graphical Representation and XML Representation

CMF has a graphical notation for depicting entities, relationships and their corresponding properties. Figure 1 shows the basic graphical notations that are used to describe a containment model. This graphical notation provides a visually straightforward way to represent and compare data models for various hypertext systems. It helps hypertext system designers and users quickly understand the underlying data model without mining through thousands of lines of code or hunting through thick user manuals or research papers. In our previous work, we have used this graphical notation to compare data models of 27 existing content management systems [12, 26, 27]. Among these 27 systems, Figure 2 shows 5 system data models that we are going to use in later sections. They vary from simple models (KMS [1], Figure 2a, 6 entities) to moderately complex models (Chimera [2, 3], Figure 2e, at least 20 entities). Note that those relationships with no membership and cardinality values defined on them use the default value of 1.

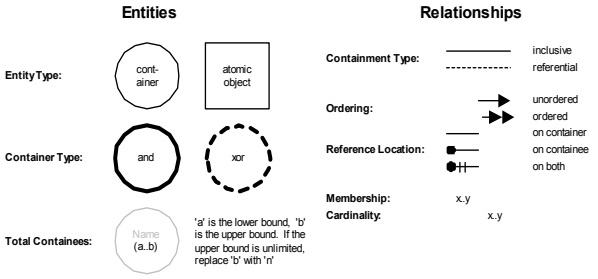
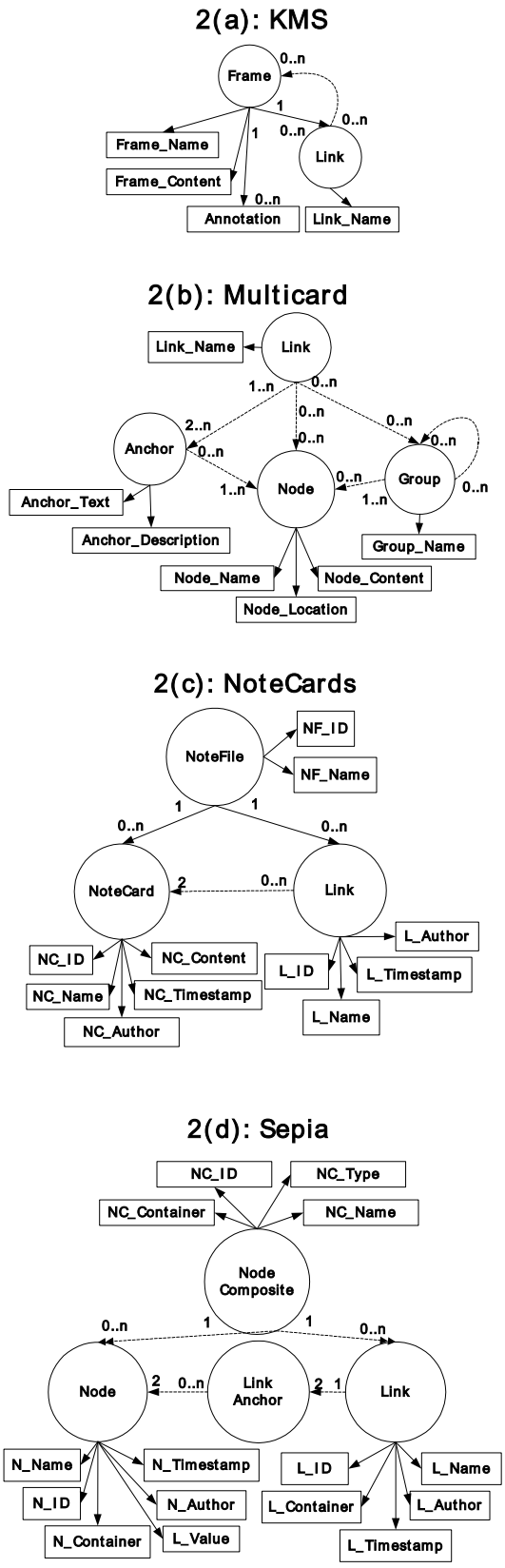


Figure 1. Graphical notations for CMF.



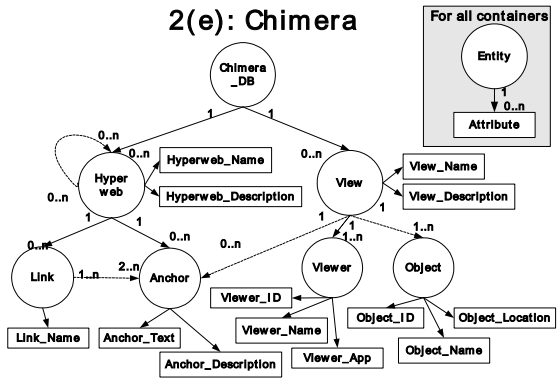


Figure 2. Containment model examples—KMS [1], Multicard [18], NoteCards [23], Sepia [20], and Chimera [2, 3]. Note that those relationships with no membership and cardinality value use the default value of 1. Chimera can have arbitrary attributes for each container entity, so we use the short-hand notation in the gray box.

The containment modeling framework is more than a visual modeling language for hypertext system data models. Because repository layer details are encoded in a uniform way, it provides the ability to automatically generate the repositories for hypertext systems. Our research is based on the assumption that the repository layer code across hypertext systems has similar functions such as reading, writing a database or file system, querying particular entities, etc. Moreover, though many hypertext systems achieve a degree of reuse by building atop database technology, we still find repetition (and hence opportunities for code generation) in the code that maps between a hypertext system specific data model and the database. By manually implementing repositories similar to those of several existing systems from scratch, we found several recurring patterns in the repository handling code. The hypothesis we explore in the work is whether it is possible to use CMF-based code generation to implement these recurring patterns and to find a shortcut for developing hypertext system repositories. Given a generated repository, a developer can build a complete system by developing complex clients that use the generated repository API or by extending the repository via an extension mechanism in HyperDisco [28], to the repository.

Given the containment model graphs, we serialize them with an XML containment model representation. The basic textual model contains two XML documents: one for the containment model definition (*_scm.xml*) and the other for the repository data type definition (**_repos.xml*). Figure 3 gives examples of these files for the NoteCards containment model. The model definition file notecards_scm.xml (Figure 3a) describes the entity and relationship definition with their properties. The repository definition (Figure 3b) carries mapping information between the logical data types used in the model definition and the physical data types used in the physical repository store. In the current implementation, we use a MySQL [15] relational database to store the instances of entities and relationships of the containment models, so the physical data types are mapped from those supported in MySQL.

```
<?xml version="1.0" encoding="UTF-8"?>
<model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://dav.cse.ucsc.edu/svn/bamboo/XML/scm.xsd">
    <entities>
        <container name="NoteFile" hasOrderedChildren="false"
ttlContainerLower="0" ttlContainerUpper="0" ttlContaineeLower="2"
ttlContaineeUpper="INF" type="generic"/>
        ...
        <atom name="NF_ID" ttlContainerLower="1" ttlContainerUpper="1"
logicalType="Integer"/>
        ...
    </entities>
    <relationships>
        <relationship name="Unordered_Referential" isOrdered="false"
type="referential"/>
        <relationship name="Unordered_Inclusive" isOrdered="false"
type="inclusive"/>
    </relationships>
    <er_model>
        <arc refRelName="Unordered_Inclusive" from="NoteFile" to="Link"
membershipLower="1" membershipUpper="1" cardinalityLower="0"
cardinalityUpper="INF" location="on_container"/>
        ...
    </er_model>
</model>
```

3(a) notecards_scm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<repos_mapping xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\repos\XML\repos.xsd">
    <value_type logicalType="Integer" physicalType="INT"/>
    <value_type logicalType="String" physicalType="VARCHAR(50)"/>
    <value_type logicalType="Datetime" physicalType="DATETIME"/>
    <value_type logicalType="Text" physicalType="TEXT"/>
</repos_mapping>
```

3(b) notecards_repos.xml

Figure 3. XML containment model specification (*notecards_scm*.xml) **and repository data type definition** (*notecards_repos*.xml) **for NoteCards.** Note that some parts are omitted to save space.

```
<?xml version="1.0" encoding="UTF-8"?>
<HypertextVersioning xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\repos\XML\hyper_mapping.xsd">
    <StructureContainer
VersionPattern="OrderedRevision">NoteFile</StructureContainer>
    <Work VersionPattern="OrderedRevision">NoteCard</Work>
    <Link VersionPattern="OrderedRevision">Link</Link>
</HypertextVersioning>
```

vernotecards_hyper_mapping.xml

Figure 4. Hypertext mapping for VerNoteCards. This mapping file combines the hypertext overlay and versioning overlay to add domain semantics on top of the two containment modeling files. It defines the feature overlay that we use to extend the core containment models.

3.3 Extending the Basic Containment Model via Feature Overlays

Basic containment models only provide information about the repository: domain specific knowledge for hypertext systems is not expressed so far. For example, by reading the two XML documents describing the model definition and the data type definition, we do not know which entity maps to a hypertext work, which one represents a link, and which one acts as an anchor. That is, we can define a container and give it the name

"link", but it will only support standard container operations such as add a member, delete a member, list members, etc. The names of the entities are, at present, just opaque strings with no semantics. Also, if we want to add new features, like version control, there is no specification in the model definition identifying the entities to which we should apply version control, what kind of version control we want, and which entities should not versioned.

A *role* is defined as a semantically meaningful entity that has specific functions and operations for a given domain or aspect. For example, a link is a role specific to the hypertext system domain that associates hyperlink semantics and link traversal operations to a container entity. As human beings, we can read the entity names on the graphical notation and relate them implicitly with the corresponding roles. The automatic generator has no knowledge of role mapping information unless we provide extra specifications.

So, we extend the basic containment model by adding feature overlays that carry domain specific role mapping information. Each overlay relates semantics specific to one domain or aspect, e.g. hypertext, version control, workspace support, distributed collaboration, with the entity definition in the containment model specification. It is only when we give the "link" container a role as "hyperlink" that the generator knows to generate hypertext operations such as link traversal. This approach to extending the basic containment model via overlays is deliberately designed to separate repository storage from specific domain semantics.

Figure 4 gives an example of a role mapping between entities and hypertext roles for the NoteCards containment model shown in Figure 2c. Specifically, it states that the NoteFile entity plays the role of StructureContainer, the NoteCard entity plays the role of a Work (document), and the Link entity plays the role of a hypertext Link. The role mapping specification also relates version control capabilities with the entities and relationships by applying a "VersionPattern". A VersionPattern is an abstraction of the choices and tradeoffs inherent in providing version control operations. In this Figure 4, the XML attribute VersionPattern is set to OrderedRevision, indicating the desire for an RCS [22] style version history. Additional detail is provided later in Section 5.2.

This flexible overlay architecture separates different aspects of a content management system. It allows the users to customize the system functionality by integrating various overlays; it also provides an open architecture to add new features without changing the existing basic repository model.

4. AUTOMATIC REPOSITORY GENERATOR

A software generator is a program that translates a domain specific language or specification into application source code [8]. Our generator program, called Bamboo, takes as input source code templates, a containment model description, a repository data type description, a hypertext mapping overlay, and other applicable overlays, such as the version control overlay (described further in Section 4.2). The outputs of the generator are source code files that include the repository schema definition and initialization, an entity class definition, an API for repository access, and a simple command line shell client for testing purposes. In the remainder of this section, we will describe the Bamboo generator architecture, its code generation process, and the architecture of the generated repositories. We also give two experiments in Section 5 to show the efficiency of hypertext repository generation, the ease of building actual hypertext systems on top of the generated repositories, and the extensibility of adding new features with overlays.

4.1 Generator Architecture

The Bamboo generator consists of four major parts, as shown in Figure 5: an XML parser, a template file processor, a physical repository schema code creator (with initialization code), and a modeled entity code generator.

The *XML parser* analyzes all the input XML documents, extracting entity and relationship definition, data type definition, hypertext entity mapping, and other overlay information. Currently, we use the Apache Xerces DOM parser [21].

Following common practice for automatic code generation, prior to execution of Bamboo, we have prepared code templates that contain model-independent repository code and special tags as place holders for model-dependent code. These code templates are the same for all the generated systems, and the *template file processor* parses the template files and generates the actual source code by inserting modeling information into the template files.

Based on the containment model and data type description, the *physical repository schema code creator* produces the code that defines the repository schema and initializes the physical repository. In the current implementation, we use the MySQL relational database because it is open source and we have experience with relational database design and optimization. The architecture can be extended to include other repository types like file systems and XML databases as long as they can support the same repository accessor API interface.

The *modeling entity code generator* is responsible for generating the entity classes that correspond to the containment model entity and relationship definition; it also creates operations for specific overlays such as "link traversal" for hypertext systems and "show version history" for version control systems.

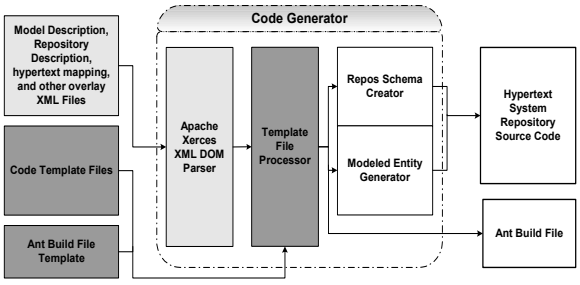


Figure 5. Generator architecture.

4.2 Generator Inputs

To generate a hypertext system repository, Bamboo needs at least four inputs: containment model description, repository data type description, template files, and hypertext mapping overlay. They are described as follows.

The *containment model description* (as shown in the example of Figure 3a) has three required XML elements under the root element *model: entities, relationships,* and *er_model*. Entities define all the possible entities in a hypertext system repository: either *atoms* or *containers*. Entity attributes are defined as XML attributes on atom and container elements. Relationships select the relationship types from four possible combinations of containment type (*referential* or inclusive) and relationship ordering (*ordered* or *unordered*) that will be used in a particular containment model. Er_model specifies all the existing arcs between entities with XML attributes that define multiplicity upper and lower bounds.

The *repository data type description* maps the logical data types in the containment model to the physical repository data types. For example, we can map a logical "String" type to a physical VARCHAR (50) type in a MySQL relational database. With this information, repository schemas can be built and initialized for the generator since it now knows how to map abstract untyped entities into the typed data items in the repository.

The *template files* are the specially tagged source code. Based on our analysis of a manual implementation of the repository of one SCM system called PIE [11], we found that only about 11.4% of the 3183 lines of repository layer code are dependent on the containment model specification. This means the rest of the code can remain static in the template files. For those parts that are dependent on model specifications, we substitute them with special generation tags so that the code generator can replace the tags with model-dependent code at generation time.

As an extension mechanism to basic containment models, Bamboo can also take in overlay specifications. The *hypertext mapping overlay* specifies a mapping from containment entities to hypertext entities, such as structure container, link, work, and anchor (using term described in [25]). It provides information to Bamboo so that it can generate domain specific operations on the entities, e.g. link traversal operations on a link entity. This overlay tells the generator that we are generating entities for a hypertext repository and some entities play specific roles that have particular hypertext operations. We also implemented a simple linear versioning overlay based on the ordered-revision versioning pattern (version control of RCS and CVS style) and integrated it with the hypertext mapping overlay. We provide a detailed description of an experiment of adding the versioning overlay to an existing hypertext system in Section 5.2.

4.3 Generation Process

From a user's perspective, the automatic code generation process is as follows. First, the user defines the required input XML documents as described above in Section 4.2. Second, the XML parser processes these XML documents, checks their consistency (e.g. to make sure the total number of containees does not conflict with the sum of cardinality on each outgoing arc definition), and converts this information into a linked list data structure in memory. After that, the code templates are processed using the information from the input specification files. Finally, the generator creates the repository schema definition code, in our case, code that defines the database tables and does the basic initialization. Entity class definition code is also generated using information extracted from the containment model.

For the purpose of testing, we also generate a command line shell client for repository exploration, providing basic repository exploration functions. Because we use Ant as out build tool for the generated code, a build file is provided as one of the generator outputs as well. At this point, the generator's work is done.

Now, the user can run the Ant build tool and get the hypertext repository layer with the simple command line shell client. If a GUI client is required, the user can start from the existing code and use the generated API to start building the GUI client right away, instead of needing to build every layer from scratch.

One of the major tasks for the generator is to transform the code templates into the desired target source code. We use a simple tag-based approach: each tag has a unique name and semantic meaning. Examples of tags include package import, linked list definition for entity classes, accessor methods based on physical data types, and specific operation methods based on entity types. But not all the tags are expanded into parts of the final code—the generator can decide where to skip the tags based on the entity definition or options in the overlay definition in the input XML files. There are 50 distinct tags in the source code template files.

4.4 Generated System Architecture

The generated hypertext system repository has four layers: physical repository layer, repository API layer, logic layer, and front end. Figure 6 shows the architecture for the generated repository.

The *physical repository layer* stores instances of the entities and relationships. The repository API defines the accessor methods to create, destroy, get and set data values from the physical repository. Our architecture includes a standard repository API and we can have multiple API implementations to accommodate different repository types, e.g. file system, relational database, object oriented database, XML data store, etc. Currently, our implementation is for a MySQL relational database only.

The *logic layer* stores the definition of the entity classes, methods for operation on their containers and containees, and methods specific to hypertext systems. It also executes active constraint checking when creating and destroying entity instances. For example, if a relationship has 2..n cardinality definition from entity A to entity B, then every time we create an instance for entity A, we must provide two instances of entity B as containees. The logic layer checker enforces this constraint and requests user input from the front end layer.

The *front end* is the client layer for using the hypertext system repository. Our generator creates a command line shell program for exploring the repository. Users need to manually build GUI clients using the generated logic layer and repository API.

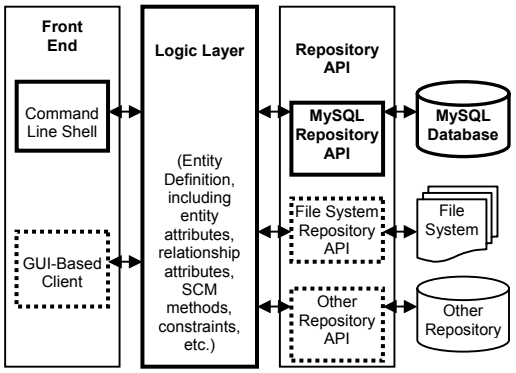


Figure 6. Architecture for the generated hypertext system repository. (Elements with dotted lines are not currently implemented or generated.)

5. TWO EXPERIMENTS WITH GENERATED REPOSITORIES

We have two goals in this section. First, we want to demonstrate that Bamboo is capable of replicating most aspects of the repository layers of interesting existing hypertext systems (Section 5.1). This validates Bamboo's ability to create a wide range of repositories with varying data models. Second, we want to highlight one advantage of our approach, namely that using a model-driven generator makes it much easier to add functionality to existing repositories (Section 5.2) By extending the model, and re-executing the generator, an extended repository can be created. We also wish to validate that we can, in fact, manually write a usable GUI interface on top of the generated repository (also Section 5.2).

5.1 Generating Repositories for KMS, Multicard, NoteCards, Sepia and Chimera

In the first experiment, we generate the repository layers for the five hypertext systems described in Figure 2. These systems represent important monolithic, open hyperbase, and link server data models. The generation process was done on a P4 1.6 GHz IBM Thinkpad R32 laptop with 384MB memory using J2SE_1.4.2_03. The generation was repeated 10 times and the generation time is the average elapsed time. The results are given in Table 1. Note that the reported generation times are for Bamboo's execution only, and do not include compilation time.

Table 1. Metrics for 5 generated repositories.

	KMS	Multicard	NoteCards	Sepia	Chimera
Total Entity Number	6	11	14	19	20
Template Files (LOC)	2848	2848	2848	2848	2848
Generated Files (LOC)	3286	3811	3649	4218	4660
Code Changes (LOC, percentage)	438 (15.4%)	963 (33.8%)	801 (28.1%)	1370 (48.1%)	1812 (63.6%)
Generation Time (s)	0.399	0.511	0.504	0.594	0.628

We draw three conclusions from these results. First, our generator scales well from small models (the 6 entities of KMS) to medium-sized models (the 20 entities of Chimera). Second, that template-based code generation of content management repositories is feasible. We note that the percentage of the newly created code (generated LOC over template LOC) is at most 63.6% of the template code base, with the percentage generally increasing with the size of the containment model. This means a large proportion of the code would have to be manually implemented if we did not use the generation approach. Third, the generator is effective in saving development time and cost. For a developer new to CMF (but with a strong existing background in content management systems), it takes a short period of time (anecdotal evidence suggests several hours) to learn the CMF and prepare all the input specification files. Then, as we can see from Table 1, it takes the generator less than 1 second to produce thousands lines of code. The same amount of work would take days, if not weeks, were to be performed manually.

5.2 NoteCards and VerNoteCards: GUI Clients on Top of the Generated Repository

Our second experiment demonstrates the feasibility of building a prototype GUI-based NoteCards-like system using the generated repository. It also shows the flexibility of system extension using our model-driven approach. By changing the containment model to add versioning definitions for the selected entities we can regenerate the repository code and have an extended system. Additionally, with minimal change to the GUI client, we can create another GUI for the version-control enhanced system, which we call VerNoteCards. We chose NoteCards as the reference system for our implementation because it is a well-known hypertext system with a straightforward data model and set of hypertext capabilities, making it well-suited for use as an example. Note that we are not reimplementing the entirely of the original NoteCards system. Rather, we are borrowing basic concepts in NoteCards and implementing them using our CMF-based generator approach.

Following the same generation steps as in the first experiment, we produced repository code for NoteCards (its containment model is described in Figure 3). Based on that, we built a prototype GUI client for our NoteCards-like repository using Eclipse JFace/SWT [9] (We freely acknowledge that this GUI differs from that of the original NoteCards). The client shows the hierarchical tree structure for the container instances, e.g. notefiles, notecards and links, in the repository. It also shows atomic properties of the selected notecards and links, e.g. author and datetime. Simple operations are also provided to add and delete the entity instances. A simple text viewer/editor is provided to display and edit the content of a notecard. The link is implemented as an external link connecting two notecards. By opening a link, the ends will be opened in two viewers, side by side.

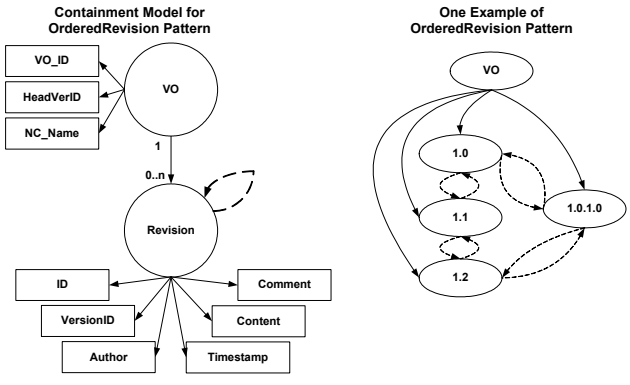
The whole development process was very straightforward because much of the low level repository handling code was produced by the generator. We as developers spent most of the time focusing on the domain logic and interface design. The NoteCards client took us two weeks to design, implement and

test, during which one week was spent on learning Eclipse JFace/SWT toolkit programming.

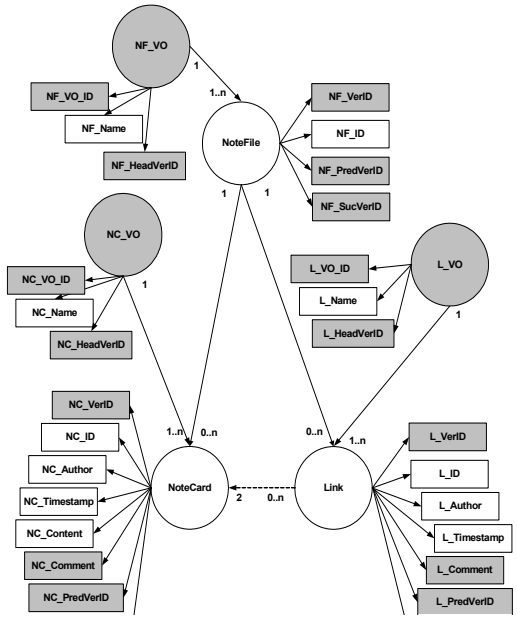
After that, we wanted to add a new feature to the GUI client—versioning support on certain entities. The normal procedure would involve refactoring of the existing code at many places. This task can be time-consuming and error-prone especially when the developers who do the refactoring are not the original developers. However, our model-driven approach makes this process much easier.

First, we changed the hypertext role mapping specification by adding the VersionPattern attribute to the entities that we wish to have versioning capabilities: in this case, NoteFile, NoteCard, and Link. From our previous research, we identified at least 5 versioning patterns from several existing hypertext versioning systems. Here we are using the *OrderedRevision* pattern—the one used by popular SCM systems like RCS [22], SCCS [19], and CVS [6]. Figure 7(a) gives an example of what this versioning pattern looks like. In this pattern, a Version History entity inclusively contains all the revisions of an entity instance. Among entity instances, there are referential relationships connecting predecessor and successor versions. The hypertext mapping specification after adding the VersionPattern attribute is shown in Figure 4; note the only difference between the old and new hypertext mapping specifications is the VersionPattern attribute. We implement the version control overlay in the hypertext mapping overlay because no other complex version control options are implemented so far. Otherwise, we could have used a separate version control specification file.

Second, we developed and used an XML transformation tool that automatically adds Version History entities to the NoteFile, NoteCard, and Link entities in the containment model XML document; also added are atom entities like VersionID, Comment, PredecessorVersionID, and SuccessorVersionID. Figure 7(b) graphically shows the generated containment model that has the OrderedRevision versioning pattern added. Those shaded *_VO containers are the newly added version histories—there should be one version history for each entity that has an OrderedRevision pattern defined on it.



7(a) OrderedRevision versioning pattern.

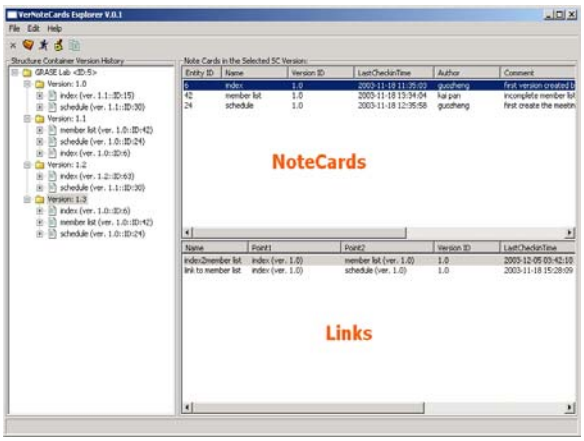


7(b) Containment model for VerNoteCards.

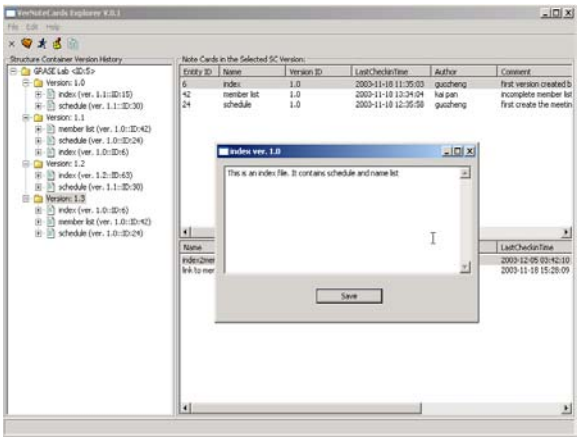
Figure 7. Applying OrderedRevision versioning pattern to the NoteCards containment model and obtaining the VerNoteCards containment model.

For example, in the new hypertext mapping specification in Figure 4, NoteFile, NoteCard, and Link entities are assigned to the OrderedRevision versioning patter, so each of them has a newly added *_VO container and other new entities for recording versioning metadata. It is possible to use other patterns, and then the entities to be added will be changed according to the pattern definition. We call the new model VerNoteCards indicating that now the NoteCards repository supports versioning. Note that only the hypertext mapping and containment model specifications are changed, while data type and code templates are not touched.

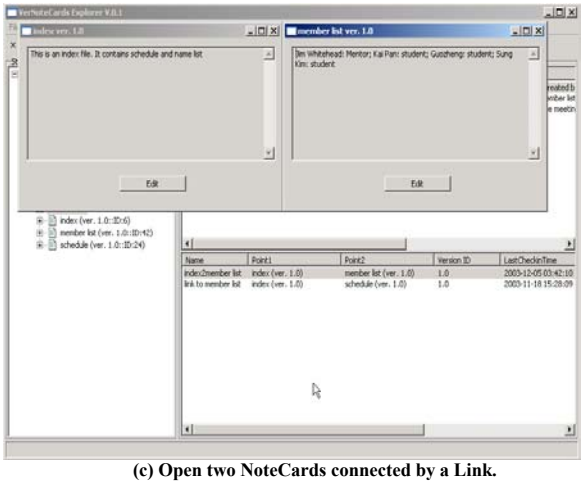
We ran the Bamboo generator with the new set of input files yielding a repository with version control support through the use of the OrderedRevision pattern. Note that *_VO containers support the checkin, checkout, and view version history operations. We then modified the NoteCards GUI client code to add versioning related operations and metadata like version tags, comment, etc. The result was the VerNoteCards client shown in Figure 8, which adds versioning support to NoteCards. Figure 8a shows versions of a NoteFile in the left-hand side tree structure. Versions of NoteCards and Links contained by a selected NoteFile version are displayed in two table panes. Figure 8b shows one NoteCard version opened in a viewer/editor. Figure 8c shows link traversal operation that opens both ends of a link version.



(a) VerNoteCards overview.



(b) VerNoteCards viewer/editor.



(c) Open two NoteCards connected by a Link.

Figure 8. Screenshots of VerNoteCards.

6. CONCLUSIONS AND FUTURE WORK

Automatic code generation for domain-specific software development has attracted much attention in the last two decades. It applies domain knowledge and design patterns to reduce monotonous manual development, increase productivity, reduce errors, and perform code optimization. Code generation is also a sharp tool for rapid prototyping, where different design choices can be tested in a cost-effective way, and design deficiencies can be identified in early stages of development.

The Containment Modeling Framework is a lightweight framework for describing the repository layers of content management systems. Our repository generator for hypertext systems simplifies the development of hypertext systems: users can focus on domain logic and user interface design for the client. The generator has demonstrated to be effective and efficient after being tested on many existing systems. This extensible architecture also allows additional features to be included using feature overlays. Each overlay represents an aspect of interest, such as hypertext and version control, and permits high-level reasoning about cross-cutting features. This avoids dependency between features and provides flexibility for adding new features. Additional information about CMF and Bamboo can be found at the project website: http://www.soe.ucsc.edu/research/labs/grase/bamboo/.

There are several directions for future work. Currently, the generated code is manually tested and validated against the design specification. This is not reliable for safety-critical applications. GenVoca and the NASA Amphion projects both provide validation algorithms and tools for testing and validation. We will investigate approaches to automatically generate test cases and validation tools for the generated code. Although we have implemented a hypertext overlay and version control overlay, we need more overlays to enrich the features of the generated repository, including workspace support, concurrency control, and distributed collaboration. As we implement more overlays, we will need to explore a mechanism for standardizing the interfaces between overlays. Since these generated hypertext repositories share the same abstract model, data exchange between different systems becomes possible. This will require a semantic mapping tool between different systems and overlays. It is also possible to generate domain logic and user interface that build on top of the repository layer. This will require rich domain knowledge about hypertext system applications and development of reusable components for domain logic representation and user interface design.

7. ACKNOWLEDGEMENT

This project is supported by the National Science Foundation under Contract Number NSF CAREER CCR-0133991. We would like to thank Dorrit Gordon and Sung Kim from the Software Engineering Lab at the University of California, Santa Cruz, for their invaluable discussion and careful review of the paper.

8. REFERENCES

[1] R. M. Akscyn, D. L. McCracken, and E. A. Yoder, "KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations," *Communications of the ACM*, Vol. 31, No. 7, pp. 820-835, 1988.

[2] K. M. Anderson, "Data Scalability in Open Hypermedia Systems," *Proc. Hypertext'99*, Darmstadt, Germany, Feb. 21-25, 1999, pp.27-36.

[3] K. M. Anderson, R. N. Taylor, and E. J. Whitehead, Jr., "Chimera: Hypertext for Heterogeneous Software Environment," *Proc. ECHT '94, European Conference on Hypertext Technology*, Edinburgh, Scotland, Sep.19-23, 1994, pp. 94-107.

[4] L. Bahler, F. Caruso, and J. Micallef, "Experience with a model-driven approach for enterprise-wide interface specification and XML schema generation," *Proc. 7th IEEE International Enterprise Distributed Object Computing Conference*, Brisbane, Australia, Sep. 16-19, 2003, pp. 288-295.

[5] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, "The GenVoca model of software-system generators," *IEEE Software*, Vol. 11, No. 5, pp. 89-94, September 1994.

[6] B. Berliner, "CVS II: Parallelizing Software Development," *Proc. the USENIX Winter 1990 Technical Conference*, Washinton DC, Jan. 22-26, 1990, pp.341-351.

[7] W. L. Buntine, "Trends & Controversies: Will Domain-Specific Code Synthesis Become a Silver Bullet?," Vol. 13, No. 2, pp. 9-15, January 1998.

[8] J. C. Cleaveland, "Building application generators," *IEEE Software*, Vol. 5, No. 4, pp. 25-33, July 1988.

[9] Eclipse.org, "Eclipse.org Portal Site," http://www.eclipse.org, last accessed time.

[10] B. Fischer and J. Schumann, "Generating Data Analysis Programs from Statistical Models," *Journal of Functional Programming*, Vol. 13, No. 3, pp. 483-508, May 2003.

[11] I. P. Goldstein and D. G. Bobrow, "A Layered Approach to Software Design," in *Interactive Programming Environment*. New York: McGraw-Hill, 1984, pp. 387-413.

[12] D. Gordon and E. J. Whitehead, Jr., "Containment Modeling of Content Management Systems," *Proc. Metainformatics Symposium 2002 (MIS'02)*, LNCS 2641, Esbjerg, Denmark, Aug 7-10, 2002, pp. 76-89.

[13] M. R. Lowry and J. V. Baalen, "Meta-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems," *Proc. 10th Knowledge-Based Software Engineering Conference*, Boston, Mass, Nov. 12-15, 1995, pp. 2-10.

[14] N. Manohar and M. P. S. Bhatia, "A tool for automated code generation for designing user interfaces on character terminals," *Proc. SoutheastCon 2001*, Clemson, SC, March 30-April 1, 2001, pp. 155-159.

[15] MySQL AB, "MySQL Reference Manual," http://www.mysql.com/doc/en/index.html, last accessed time March 9, 2004.

[16] Object Management Group (OMG), "Introduction to OMG's Unified Modeling Language (UML)," http://www.omg.org/gettingstarted/what_is_uml.htm, last accessed time March 9, 2004.

[17] Object Management Group (OMG), "MDA Guide Version 1.0.1," http://www.omg.org/docs/omg/03-06-01.pdf, last accessed time March 9, 2004.

[18] A. Rizk and L. Sauter, "Multicard: An open hypermedia system," *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp.4-10.

[19] M. J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, Vol. 1, No. 4, pp. 364-370, 1975.

[20] N. Streitz, J. Haake, J. Hannemann, A. Lemke, W. Schuler, H. Schutt, and M. Thuring, "SEPIA: A Cooperative Hypermedia Authoring Environment," *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov.30-Dec.4, 1992, pp. 11-22.

[21] The Apache XML Project, "Xerces2 Java Parser 2.6.2 Release Notes," http://xml.apache.org/xerces2-j/index.html, last accessed time March 9, 2004.

[22] W. F. Tichy, "RCS - A System for Version Control," *Software-Practice and Experience*, Vol. 15, No. 7, pp. 637-654, 1985.

[23] R. Trigg, L. Suchman, and F. Halasz, "Supporting Collaboration in NoteCards," *Proc. Computer Supported Cooperative Work (CSCW'86)*, Austin, Texas, Dec. 3-5, 1986, pp. 147-153.

[24] S. Vigna, "Automatic Generation of Content Mangement Systems from EER-Based Specifications," *Proc. 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, Montreal, Quebec, Canada, Oct. 6-10, 2003, pp. 259-262.

[25] E. J. Whitehead, Jr., "Design Spaces for Link and Structure Versioning," *Proc. Hypertext 2001, The Twelfth ACM Conference on Hypertext and Hypermedia*, Arhus, Denmark, August 14-18, 2001, pp. 195-205.

[26] E. J. Whitehead, Jr., "Uniform Comparison of Data Models Using Containment Modeling," *Proc. Hypertext 2002, The Thirteenth ACM Conference on Hypertext and Hypermedia*, College Park, MD, June 11-15, 2002, pp. 182-191.

[27] E. J. Whitehead, Jr. and D. Gordon, "Uniform Comparison of Configuration Management Data Models," *Proc. 11th International Workshop on Software Configuration Management Systems (SCM-11)*, Portland, Oregon, May 9-10, 2003, pp. 70-85.

[28] U. K. Wiil and J. J. Leggett, "The HyperDisco Approach to Open Hypermedia Systems," *Proc. 7th ACM Conference on Hypertext (Hypertext '96)*, Washinton, DC, March 16-20, 1996, pp. 140-148.