Hypertext Versioning for Embedded Link Models

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

ABSTRACT
In this paper, we describe Chrysant, a hypertext version control system for embedded link models. Chrysant provides general-purpose versioning capability to hypertext systems with an embedded link model. To apply Chrysant for a specific hypertext system, we require the containment model for this system's data model, the containment model of the version repository for this system, the hypertext role definition, the versioning role definition, and the filesystem mapping definition. Additionally, a specific parser that retrieves the link targets from the hypertext resources is needed. Hypertext versioning is different from versioning an individual resource in the traditional way, in that both the content of a hypertext resource and the relationships between it and other resources related by hypertext links are versioned. In Chrysant, the structure container and the content of a hypertext resource are versioned separately. We describe the architecture of Chrysant, and explain the procedure of the check-in and check-out functions. An AF-BTU algorithm is introduced in the paper to check in the hypertext network of a hypertext resource. As a case study, the application of Chrysant for HTML content is introduced. We create necessary definition specifications for the HTML system and a parser to retrieve link targets from a HTML document. Some examples of HTML versioning with Chrysant are shown.

Categories and Subject Descriptors
D.2.2 [Document preparation]: Hypertext/hypermedia;
H.2.1 [Database Management]: Logical design – data models.

General Terms
Design, Documentation

Keywords
Containment model, hypertext versioning, structure versioning, link versioning, HTML versioning, version control system, SCM.

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

1. INTRODUCTION
Hypertext resources evolve just as other unlinked resources do and we need to record their version history for configuration management purposes. Here, a resource is "anything that has identity" [1], such as a bitmap image, a source code file, or a directory. Regular resources are usually versioned individually in version control systems like RCS [2], CVS [3], and SCCS [4]. More advanced SCM systems, such as Subversion [5] and ClearCase [6], support directory versioning or project-level versioning, where the filesystem hierarchy is versioned as well as individual files. The distinction of hypertext is that a hypertext resource is not isolated. The links inside or outside hypertext resources form a hypertext network. So, versioning a hypertext resource goes beyond the individual document boundary and reaches outside to the other resources it links to, and so on. In other words, versioning a hypertext resource means versioning the whole hypertext network. That is, not only the content of a hypertext resource is versioned, but also its relationships with other resources as referenced by hypertext links.

Why should we version the hypertext network as a whole? Here are some scenarios that show the necessity of tracking the evolution of the whole hypertext network.

- An online reference manual for a database system has an index of chapters in the root hypertext document, with each chapter saved in an individual document. Every time changes are made to some of the chapter documents, the whole reference manual should have a new version, though the root document is unchanged. In this case, all the documents should be versioned as a whole, because all the documents related by the links are in fact treated as one document logically.
- An HTML page uses an external CSS (cascading style sheet) file to define the page presentation. When the CSS file is changed, the HTML page will have a new presentation, though the HTML page itself will not have a new version. Ideally we would like to version the two documents together to effectively record the evolution of the presentation of the document.
- An HTML page has multiple frames, and each frame links to another HTML document. Now, the presentation history of this HTML page will depend on the history of the linked-to HTML documents.

Links are treated differently across hypertext systems [7], with some systems, like HTML(WWW), Hypercard [8], and KMS [9] having links embedded in the content of the hypertext document,

while other systems, such as Chimera [10], Aquanet [11], and HyperDisco [12] have external links.

This paper only focuses on the approach to version those hypertext models with embedded links. Embedded links eliminate the need to handle the issues raised by Østerbye concerning the immutability of versions and the versioning of links [13], since the embedded links are versioned together with the content of the hypertext resource that contains them, and link modification will definitely cause a change to the containing hypertext resource in embedded link models. But, at the same time, embedded links introduce the difficulty of retrieving links from the content of the hypertext resources, since a hypertext content parser is required to identify links and then discover link targets. It is thus somewhat more difficult to reason about embedded link structures.

Previous hypertext versioning work has focused almost exclusively on external link versioning, treating embedded links as a trivial case [14]. This work focuses on embedded link versioning, but with an approach that gains the advantages of both embedded and external link models.

This paper presents Chrysant, a system that provides general-purpose hypertext versioning capability to hypertext systems with embedded link models. That is, the check-in, check-out and other version control functions in Chrysant work for any embedded link hypertext model. Several necessary model definitions for a specific hypertext system are needed for Chrysant to work for this system. These definition specifications include the containment model for this system's data model, the containment model of the version repository for this system, the hypertext role definition, the versioning role definition, and the filesystem mapping definition. A hypertext parser for a specific hypertext system is also required as an external module to Chrysant to retrieve the link targets from the hypertext resources in this hypertext system.

To version the hypertext network, the hypertext link structure and the content of a hypertext resource (document/node) are versioned separately in our approach. Using terminology defined in [14], a hypertext resource has two roles in the Chrysant system: one is *structure container*, since a hypertext resource contains a set of embedded links; the other is the resource *content*, which needs to be versioned as regular files. Both of the structure container and the resource content have their own version histories.

There are several unique contributions made by our work. First, Chrysant provides hypertext versioning capability for hypertext systems with embedded link models. Second, a versioning mechanism is introduced to version the structure container and the content separately for a hypertext resource. Third, an algorithm, AF-BTU, is developed to version a hypertext network as a whole. Last, as an application case, HTML versioning is implemented with Chrysant, which has practical usage.

In the next section we discuss the containment model and other definitions which are used to generalize hypertext systems. In Section 3, we explain the architecture, design, and core algorithm of Chrysant. In Section 4, we present the practical application of hypertext versioning for HTML. Last, we discuss our future work and conclude in Section 5.

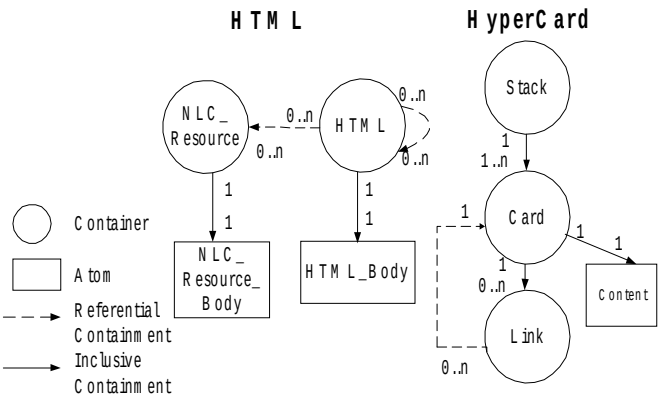

Figure 1. Containment Model of HTML and HyperCard

2. MODELING HYPERTEXT AND VERSIONED HYPERTEXT

To create a hypertext version control system that works for various hypertext systems, a general-purpose modeling approach should be applied to the data model of hypertext systems. This section introduces all the definitions needed by this hypertext version control system. These definitions include the containment model of a hypertext system, the containment model of the version repository of this system, the hypertext role definition, the versioning role definition, and the filesystem mapping definition. All these 5 definitions are represented in XML format, so that Chrysant can read them into memory when running.

2.1 Containment Modeling of Hypertext

Containment modeling [7, 15] is a uniform approach to represent system data models. A containment model is a specialized form of entity-relationship model in which entities and relationships are two primitives, and the only valid relationship is of type '*contains*'. Two kinds of containment are allowed, *referential* or *inclusive*. The entities in contentment models can be of type *container* or *atom*. A *container* can referentially or inclusively contain other entities, while an *atom* cannot. An atom can hold content data, while a container only holds (references to) its containees. Compared to entity-relationship models, which allow all kinds of user-defined relationships, containment models only capture the containment (or aggregation) relationships in data models. The simplicity and the focus on containment relationships make containment models a suitable model to represent the repository data models of a wide range of content management systems. To date, containment modeling has been used to model the data models of 13 configuration management systems [7, 15, 16] and 15 hypertext systems [7, 15].

Two example containment models for HTML (WWW) and HyperCard are shown in Figure 1.

In the HTML example, the *HTML* entity represents a logical HTML document and the *HTML_Body* entity represents the content of the HTML document, so the relationship between the *HTML* entity and the *HTML_Body* entity is *inclusive containment*. The *NLC_Resource* entity represents a different logical document, those resources without links in them such as bitmap image documents. The *NLC* here stands for *Non-Link Containing*. The

HTML entity has a many-to-many *referential* relationship with both the *NLC_Resource* entity and the *HTML* entity, since an HTML document can link to other HTML documents and non-HTML documents through its embedded links. For example, an HTML document can embed an external JPEG resource in its text through a <img> tag. In this example, the HTML document is an instance of the *HTML* (and inclusively contained *HTML_Body*) entity, and the JPEG file is an instance of the *NLC_Resource* (and inclusively contained *NLC_Resource_body*) entity.

In order for a program to process the containment models, we create an XML document type to represent the containment model. There are two major elements, *entities* and *er_model*, defined in the XML containment model document. The *entities* element defines all the entities in the data repository of a system; these can have sub-elements *container* or *atom*. The *er_model* element defines all the relationships (arcs) between the entities in a data repository. Taking the HTML containment model in Figure 1 as an example, there are 4 sub-elements under the *entities* element in the XML containment model document for HTML. They are *HTML*, *NLC_Resource*, *HTML_Body*, and *NLC_Resource_Body*. The former two are *containers*, and the latter two are *atoms*. Accordingly, there are 4 sub-elements under the *er_model* element in the XML document to represent the relationships between those entities, which are *HTML* to *HTML*, *HTML* to *HTML_Body*, *HTML* to *NLC_Resource*, and *NLC_Resource* to *NLC_Resource_Body*.

Containment models provide the basic entity-relationship information for the data model of a hypertext system, but they do not carry any versioning or hypertext information, which are required for general-purpose hypertext versioning tasks. These definitions are introduced in the following subsections.

2.2 Hypertext Role Definition

The hypertext definition specification defines the hypertext roles for the entities in the containment model, and hence adds hypertext semantics to them. Though there could be many kinds of entities defined in a containment model for a hypertext system, there are only a few predefined hypertext roles these entities can be mapped to. Previous research [14] has summarized the hypertext semantics for entities in hypertext version control systems. For a general-purpose hypertext version control system for embedded link models, we define the following hypertext roles based on [14].

Non-link Containing (NLC) Resource: An artifact that does not contain any links, such as an image file, a song file, or a pure text document.

Structure Container: A container that contains (references to) a set of links, NLC resources, or other structure containers.

Structure Container Content: The body content of an artifact which also serves as a structure container. Structure container content always maps to an atom in the containment model and is always inclusively contained by a structure container.

Link: An association among a set of structure containers and non-link containing resources.

As an example, we map the entities in the HTML containment model in Figure 1 to the hypertext roles as follows.

HTML ←→ *Structure Container*

HTML_Body ←→ *Structure Container Content*

NLC_ Resource ←→ *NLC Resource*

In XML format, this mapping can be represented as follows.

```
<hypertextMapping>
  <mapping containmentClass="html"
           hypertextRole="h_structure_container"/>
  <mapping containmentClass="html_Body"
           hypertextRole="h_sc_content"/>
  <mapping containmentClass="NLC_resource"
           hypertextRole="h_NLC_resource"/>
</hypertextMapping>
```

The *HTML* entity is mapped to *structure container*, since a HTML resource has embedded links in it. The *HTML_Body* entity is mapped to *structure container content*. Last, the *NLC_Resource* entity is mapped to *NLC resource*.

We should notice that the link role is not explicitly represented in the HTML containment model. Links in a version control system for embedded link models can be ignored, since links do not have their own version history, and we only care about the relationships caused by links. We should also notice that a hypertext resource in reality has two roles in our modeling approach, one is *structure container*, and the other is *structure container content*. Both of them have their own version history.

2.3 Containment Model of the Version Repository

The containment model of a hypertext system records the original data model for the hypertext resources in that system. Adding version control support to existing hypertext systems involves a process of extending the original data models. The added versioning information includes the version history object that stores all checked-in versions of the resource, the author who checks in the resource, the time the resource is checked in, the check-in comments, etc.

The entities with different hypertext roles have different representations in the version repository. The *Structure container* has a version history on its structure to record the evolution of the structure of a hypertext resource. The *Structure container content* has a version history on its content to record the evolution of the content of a resource. *NLC resources* have a version history on their content to keep track of the evolution to the content of a *NLC resource*.

The benefit of this means of versioning embedded links is that it separately versions the link structure and body content. Since they are independently versioned, we can version link structure as a first class entity.

Let's take the HTML containment model as an example. The containment model of the version repository for HTML is shown in Figure 2.

After comparing the containment model for versioned HTML (Figure 2) with the original HTML containment model (Figure 1), we see that there is a *VO* (versioned object) entity (*HTML_VO*, *HTML_Body_VO*, *NLC_Resource_VO*) added for every entity with the hypertext role of *structure container*, *structure container content*, or *NLC resource* in the original model. A *versioned*

object is an object containing the version history of a logical entity. Every *VO* entity has a one-to-many relationship with its contained version entity, which means a version history object can contain many versions of a resource. In Figure 2 these version entities are *HTML_Version*, *HTML_Body_Version*, or *NLC_Resource_Version*, which correspond to the *HTML*, *HTML_Body*, and *NLC_Resource* entities in the original HTML containment model respectively.

Some atomic entities are not shown in Figure 2 due to limited space. In fact, each *VO* entity has a *fileName* entity, which records the filename of the resource when checked out to the filesystem, and a *head* entity, which stores the last version number of the resource. Every *version* entity contains several atomic entities which are not shown in Figure 2, namely *checkin_timestamp*, *author*, *comments, version_identifier* and *is_root*. The *checkin_timestamp*, *author*, and *comments* entities record the check-in time, author, and comments metadata when checking in a resource. The *version_identifier* entity records the version number of a version. The *is_root* entity indicates whether this version of the resource is a root hypertext resource.

2.4 Versioning Role Definition

The containment model of the version repository for a hypertext system has many entities added to represent versioning functionality (e.g. HTML_VO in Fig. 2), but the containment model itself does not carry any versioning information. That is, the versioning semantics in the containment model is unknown to the program. For example, without any additional information, it is impossible to differentiate between normal containers and those intended to act as VOs. VOs support additional operations for check-in, check-out, and retrieving version history. For a version control system to understand the versioning semantics in a containment model, it is necessary to define several versioning roles and then map the entities in the containment model to these versioning roles, similar to the mapping performed for hypertext roles. Table 1 shows the versioning roles defined, and their according mapping entities in the containment model of the version repository for HTML.

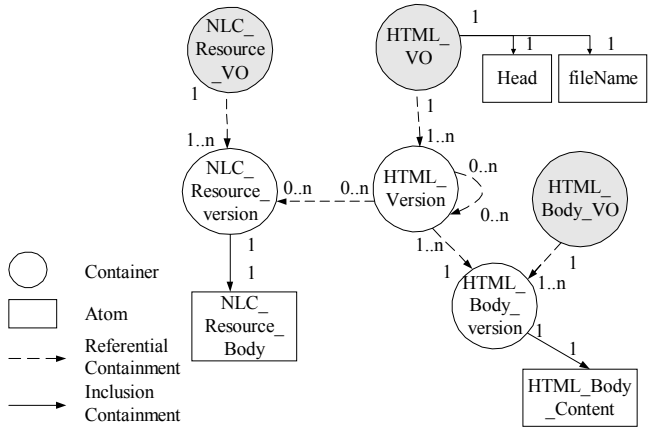

Figure 2. Containment Model of Version Repository for HTML.

Table 1. Versioning Roles and the Mapping to the Entities in the Containment Model of the Version Repository for HTML.

Versioning Role	Meaning and Usage	Mapping Entity in Version Repository for HTML
s_revision	version entity in versioned HTML containment model.	HTML_Version, HTML_Body_version NLC_Resource_Version
s_rev_content	textual or binary content for version entity	HTML_Body_Content NLC_Resource_Body
s_rev_identifier	unique identifier for each version, usually the version number	The *version_identifier* entity under each *version* entity.
s_rev_checkin_timestamp	timestamp for each version when it is checked in	The *checkin_timestamp* entities under each *version* entity.
s_rev_comment	log information for each version	The *comment* entities under each *version* entity.
s_rev_author	author that checks in a version	The *author* entities under each *version* entity.
s_rev_history	version history entity, usually the container for version	HTML_VO, HTML_Body_VO, NLC_Resource_VO
s_rev_history_identifier	handler to locate version history	The *fileName* entities under each *VO* entity.
s_rev_trunktip	latest version in the main branch	The *head* entities under each *VO* entity.

These versioning roles are understandable to the program and general to all the version repositories for hypertext systems, so they make it possible to develop a hypertext version control system that provides general-purpose version control functions.

2.5 File System Mapping Definition

Embedded-link hypertext systems tend to (but don't always) use the filesystem to store hypertext resources – examples include KMS [9], HyperCard [8] and the Web. We assume that users edit their hypertext resources in an isolated local disk area, called a workspace. After editing is complete, a check-in function can be used to store the changed resources into the hypertext version repository as new versions. Check-out is the reverse procedure to check-in, in which a certain version of a resource is retrieved from the hypertext versioning repository and saved into the workspace in the filesystem. So, in this approach the hypertext version control system always interacts with the filesystem.

Hypertext versioning involves many kinds of files, and treats them in different ways according to their entity type in the containment model. A filesystem mapping is hence necessary for hypertext versioning. In our approach, we map the filename suffixes to the entities of the containment model. For example, in the HTML model, the filename suffixes, *.htm* and *.html*, are mapped to the *HTML* entity, and the other filename suffixes are mapped to the *NLC_Resource* entity.

Having this information, the hypertext version control system versions a *.htm* or *.html* file as a *structure container*, where the structure versioning is considered, and versions other files as an individual *NLC resource* file, where only the evolution of the resource content is tracked.

3. HYPERTEXT VERSIONING

This section introduces our approach for structure versioning, the design and implementation of Chrysant, and the core algorithm for versioning a hypertext network.

3.1 Structure Versioning

Structure versioning is a key problem in hypertext versioning, since the objective of structure versioning is to keep track of the state of the entire hypertext network by creating snapshot images for both the data and structure of the hypertext network over time [13, 17]. Backtracking to a previous state of a structure means to find all the node resources that were newest at some point in time [13]. The structure here is not the internal structure of a document, but the structure formed by links in hypertext resources. [13] and [17] have discussed the structure versioning problem.

A hypertext network is a DCG (directed cyclic graph) structure. For an arbitrary unversioned structure, a change in structure means a new node was added to the structure, or a node was deleted. But, for a versioned structure, the change to the structure is caused by changes to the nodes this structure contains. That is, besides deleting a node or adding a new node, a structure has a new version when any node in the structure has a new version.

In our approach, as discussed in the previous section, the structure and the content of a hypertext resource are versioned separately, since a logical document with embedded links has two roles, one as *structure container* (just the links), and the other is the resource content, *structure container content* (the content with embedded links). Hypertext versioning in this case involves structure versioning plus content versioning. Figure 3 shows an example of the version history of a HTML document in the repository.

The smaller circles in Figure 3 stand for the instances of the *HTML_Version* entity as described in Figure 2. The larger circles stand for the *HTML_Body_Version* entity, such as the versions of *a.htm Body*, *b.htm Body*, and c*.htm Body*, except the circle for *d.gif resource Version 1,* which is an instance of the *NLC_Resource_Version* entity. The shaded circles indicate the changed instances at each time instance. The arrows in Figure 3 represent the instances of the relationships among the entity instances in this example.

Figure 3 shows that, in hypertext versioning, changes propagate from the bottom of the hypertext network to the top. If a node in the network changes, it will cause the structure container containing it to have a new version too. This impact will be imposed recursively from bottom to top.

3.2 Architecture of Chrysant

Chrysant is a general-purpose hypertext version control system for hypertext systems using the embedded link model. That is, the check-in, check-out and other version control functions in Chrysant work for any embedded link hypertext model without changing its source code. Since the underlying data model for embedded link hypertext systems can vary, our system must be capable of adapting to a range of data models. We accomplish this by explicitly representing the data model using containment modeling, and then identifying the roles (e.g. Structure Container, s_revision, etc.) played by specific entities.

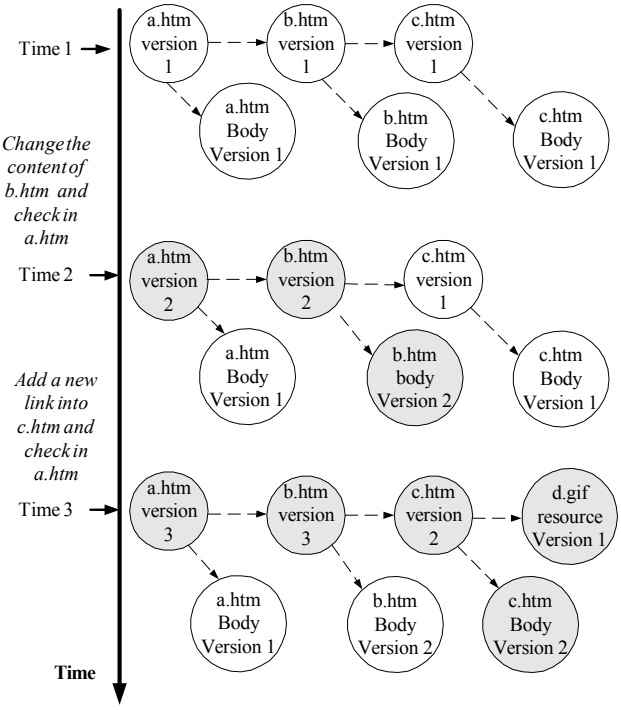

Figure 3. Version History of an HTML document, a.htm.

To version a specific hypertext system, Chrysant requires a containment model specification of the hypertext system without version control, the containment model of the repository for this system with version control, the hypertext role definition specification, the versioning role definition specification, and the filesystem mapping specification. These 5 definition specifications serve as configuration files for versioning a specific hypertext system, and Chrysant will read these specification documents into memory structures when executing. Besides, an external module is needed by Chrysant to retrieve the link targets from the hypertext resources. This module has to be model-dependent, since it carries knowledge of how to parse the document structure and extract embedded links.

In Figure 4 we show the system architecture of Chrysant. The major parts of Chrysant's functionality are the modules that implement general-purpose version control functions, such as *check-in*, *check-out*, *log*, which displays the version history of a structure container or regular resource, and *scdiff*, which shows the structure difference of two versions of a structure container.

Chrysant also has a module, the *hypertext parser interface*, which interacts with an external hypertext parser for a specific hypertext model through system call. This *specific hypertext parser* is responsible for retrieving all the link targets in a hypertext document and returning them to the hypertext parser interface. In this way, we are able to accommodate the range of document formats across embedded link systems by isolating link extraction (which requires deep knowledge of document format) to a single, plugable module. The hypertext parser takes in the filename of a hypertext file as a parameter, and returns a string list that contains all the filenames of the link targets this hypertext file contains.

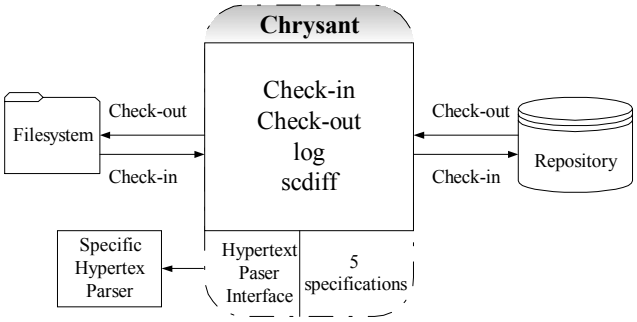

Figure 4. The Architecture of Chrysant.

Chrysant has a repository to store the version history of the hypertext resources, a MySQL database with 2 tables, *entity* and *relationship*. The *entity* table stores the instances of all the entities in the containment model, and the *relationship* table stores all the instances of the relationships among the entity instances.

The 5 specification documents for a specific hypertext system define the containment model and other properties of this system. These documents are in XML format, which will be read by Chrysant into memory structures when Chrysant begins to run.

Chrysant resides on the machine where the hypertext resources are stored. The Chrysant repository database can be on the same or different machine as Chrysant.

3.3 Check-in and Check-out

The check-in function in Chrysant checks in a hypertext resource with the according hypertext network from the filesystem to the repository. Usually Chrysant checks in a root hypertext document and its hypertext network. If the resource the user specifies is not a root hypertext root, Chrysant will find this resource's root hypertext resources in the version repository and check in these root hypertext resources instead. The following procedure explains how Chrysant checks in a root hypertext resource and how the 5 definition specifications are used in the check-in procedure.

1. Chrysant uses the hypertext parser to get all the resource filenames of the link targets from the content of the hypertext resource. (The hypertext parser is used in this step.)
2. For every resource identified in step 1, Chrysant finds its hypertext role by looking up its filename suffix in the filesystem mapping definition and hypertext role definition. For those resources whose hypertext role is *structure container*, Chrysant finds all the resources linked by these resources. This is a recursive procedure. After that, Chrysant gets a hypertext network rooted at the resource to be checked in. The hypertext network is an instance of the containment model of this hypertext system. (The containment model specification, the filesystem mapping definition, the hypertext role definition, and the hypertext parser are used in this step.)
3. For every changed node in the hypertext network, Chrysant checks in that node to the repository. For the node resource whose hypertext role is a *structure container*, Chrysant checks whether its structure has changed. (The hypertext role specification is used in this step.)
4. Versioning information according to the containment model of the version repository is added to the repository when a resource is checked in. Relationships are also built among the newly created resource versions based on the containment model of the version repository. (The versioning role specification and the containment model specification of the version repository for this hypertext system are used in the step.)
5. If it is the first time this root hypertext resource is checked in, Chrysant marks this resource version as 'root' in the repository by setting its atom entity, *is_root*, to 'true'.

From the dataflow perspective, the check-in function transforms the data in this way: *filesystem files* → *instances of containment model* → *instances of containment model for the version repository*. Check-out functions transforms the data in the reversed way: *instances of containment model for the version repository* → *instances of containment model* → *filesystem files*. Checking out a resource in Chrysant has the following process steps.

1. Based on its filename and version number, Chrysant finds the resource version to be checked out in the repository.
2. Chrysant saves this resource version to the filesystem. If the checked-out resource is a *NLC resource*, the check-out procedure is done.
3. For a resource with structure container role, Chrysant finds all the containee resource versions contained (linked) by this resource version in the repository through the relationships between resource versions.
4. Chrysant checks out all the containee resource versions to the filesystem. This step is a recursive procedure.

The main purpose of versioning a structure container is to version a collection of related resources as a whole so that if one resource changes, a new version gets created for that resource and a new version is also created for the structure container that contains it.

3.4 AF-BTU Algorithm

Checking in a hypertext resource is much more difficult than checking in a regular resource, since it involves checking in the whole hypertext network rooted at the resource. In check-in, Chrysant creates a new version only for those changed nodes in a hypertext network. For a node with a hypertext role of *NLC resource* or *structure container content*, its current content is compared to the content of its previous version in the repository to determine whether this node has changed. For a node with a *structure container* role, the structure change has to be checked since it holds a hypertext network. Now, the difficulty lies in how to detect structure changes. As we have discussed in section 3.1, changes to a structure container depend on the changes to the containee nodes contained by this structure container. But, the containee nodes could be structure containers too. So, it is a recursive procedure to check for structure change. In essence, changes propagate from the bottom to the top in a hypertext network. Since a hypertext network is a directed cyclic graph, several special cases of the graph topology of a hypertext network should receive special study before we design the check-in

algorithm. Figure 5 shows two study cases of the hypertext network.

From the two cases in Figure 5, we can see that there can be cycles in the graph of a hypertext network. In case 1, the cycle is B → C → B. In case 2, the cycle is A → B → C → A. When a cycle appears, the nodes in the cycle depend on each other. For example, in case 1, changes to B depend on changes to C. At the same time, changes to C depend on changes to B.

Another characteristic of the hypertext network is that the *structure container content* node and *NLC resource* node, called *atom nodes*, will not appear in a cycle, since they don't depend on any other nodes and they are always leaf nodes in the network graph. Changes to atom nodes always cause change to the *structure container* above them. Due to these characteristics of the hypertext network, an atom-node-first, bottom-to-up (AF-BTU) algorithm is used in Chrysant to detect changes and check in changed nodes in the hypertext network. This algorithm also maintains a list of the nodes that have been checked in to avoid checking in a node for more than once or circular check-in. The detailed AF-BTU algorithm is described as follows.

1. For the hypertext resource to check in, retrieve the hypertext network of this hypertext resource, i.e., the resources reachable by following the links from the resource, and save this hypertext network in a memory structure.
2. For every *atom node* (*structure container content* or *NLC resource*) in the hypertext network, check whether they have changed by comparing them with their previous version in the version repository.
3. Check in the changed *atom nodes* found in step 2.
4. From every changed *atom node*, traverse upwards through its parent containers, the parent containers of its parent containers, and so on. For every structure container node in the upward path, check it in and add it to the checked-in list if it has not been checked in before. Every bottom-to-up traversal procedure will stop whenever it meets a node that has been checked in or a node that has not any parents.

Here is an example of applying the AF-BTU algorithm. Suppose the hypertext network is like the one in Figure 6, and the *atom* nodes *Content of B* and *NLC Resource D* have changed. The execution steps of the AF-BTU algorithm to check in the hypertext network rooted at *A* are as follows.

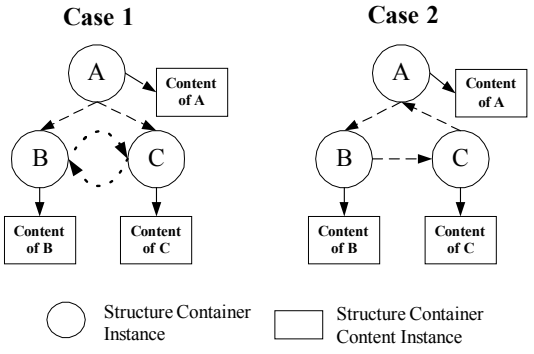

Figure 5. Study Cases of Hypertext Network.

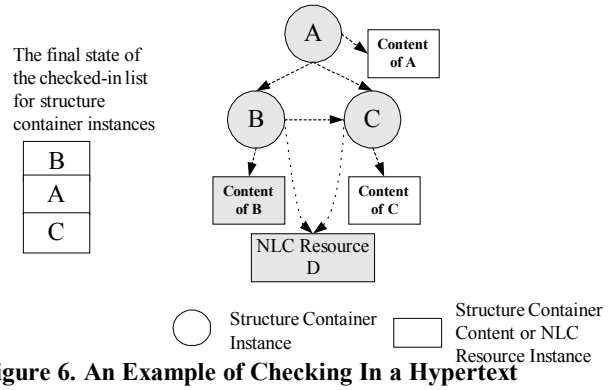

Figure 6. An Example of Checking In a Hypertext Resource. The shaded nodes stand for those will have a new version.

1. Nodes *Content of B* and *NLC Resource D* are checked in.
2. From the node *content of B*, the upward traversal path is *Content of B* → *B* → *A*.
3. Check in *B* and *A*, and add them to the checked-in list.
4. From the node *NLC Resource D*, there are three upward traversal paths. One is *NLC Resource D* → *B* → *A*. The second one is *NLC Resource D* → *C* → *B* → *A*. The third one is *NLC Resource D* → *C* → *A*.
5. For the first path, *NLC Resource D* → *B* → *A*, the algorithm stops when *B* is met, since it is already in the checked-in list.
6. For the second path, *NLC Resource D* → *C* → *B* → *A*, the algorithm checks in *C* and adds it to the checked-in list, and stops when *B* is met, since *B* is already in the checked-in list.
7. For the third path, *NLC Resource D* → *C* → *A*, the algorithm stops when *C* is met, since *C* is already in the checked-in list.

After executing the algorithm to check in all changed nodes in the hypertext network, one more step will be taken to reconstruct the relationships between the new versions of the changed nodes in the repository, since the check-in actions in the AF-BTU algorithm only create new version instances for changed node in the repository, and the relations between the newly created entity instance versions and their containers, which could be newly created versions too, need to be built in the repository.

The algorithm for checking out a hypertext network is much easier than check-in. Only a top-to-bottom traversal is enough. That is, for a *structure container* node in the hypertext network, find the resource versions it contains, and check out all these containees recursively. For checking out an *atom node*, create a file on the filesystem according to its *fileName* property, and retrieve the content of the *atom node* from the repository and put it to the created file. Also, a *checked-out list* needs to be maintained during the check-out process to avoid circular check-out.

4. VERSIONING HTML

As a case study, we apply Chrysant for HTML to version HTML documents. The 5 definition specifications for HTML used by Chrysant have already been introduced in section 2. This section will introduce the HTML parser for Chrysant and the application examples of Chrysant for HTML documents.

4.1 HTML Parser for Chrysant

An HTML parser is required by Chrysant to version HTML documents. This HTML parser should be able to retrieve the link targets from a HTML document. We wrote a parser using an open source HTML parser library, El Kabong [18].

In the parser, we added call back functions to monitor several tags when scanning a HTML document. These tags may contain some parameters which indicate the resources this HTML document links to. Table 2 shows the HTML tags and their according parameters that indicate the link target.

The parser will output the URLs contained in the tags and parameters listed in Table 2 during parsing. The full URLs, e.g. those starting with *http://*, are ignored, and only local URLs are retrieved. Also the section after '#' or '?' in the URL will be discarded. Chrysant checks the existence of the files indicated by these retrieved local URLs and discards those local URLs that don't actually have a file mapping to them in the local filesystem. The *base* tag is also considered, but it has to be a local URL. Special treatment is given to the *@import url* and the *background: url* sections in the comment text of a HTML document, which may also contain local link targets.

4.2 Examples

For an experiment, we used Chrysant for HTML to version the contents of the home page for the Bamboo project (http://www.soe.ucsc.edu/research/labs/grase/bamboo/index.htm). We simulated the process used when we wrote the Bamboo Web pages, and used Chrysant to version them. We took four steps to finish editing the Bamboo Web pages. After each step we edited the Bamboo web pages at the local filesystem, and checked them in using Chrysant for HTML. Figure 7 shows the state of the Bamboo Web page structure at each step and the sequence of editing for Bamboo Web pages.

Table 2. HTML Tags and Parameters Containing Link Targets.

HTML Tag	Parameter
a	href
img	dynsrc, src
bgsound	src
body	background
area	href
link	href
form	action
input	src
frame	src
embed	src
object	object, data
param	value
script	src

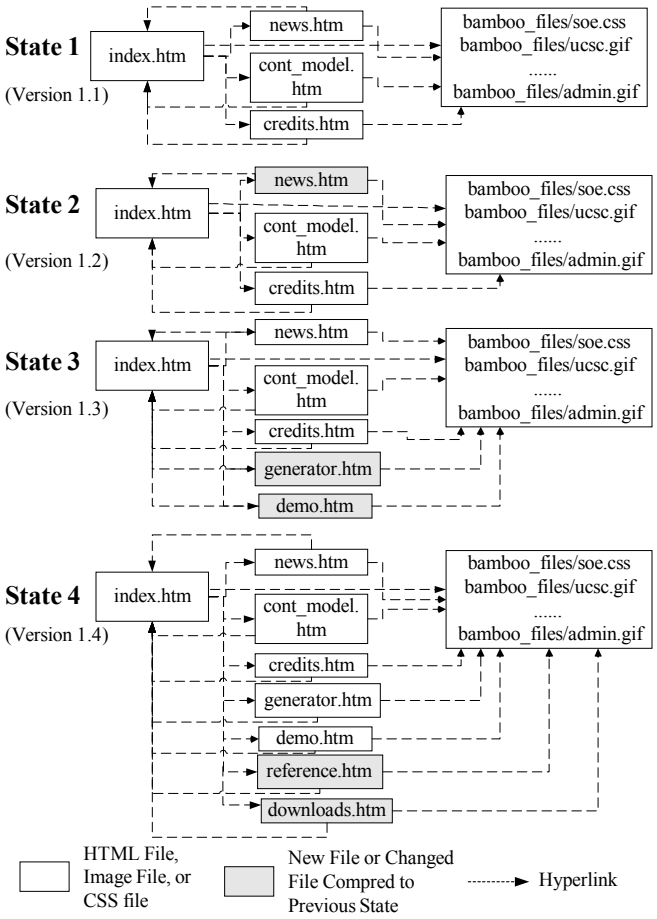

Figure 7. Versioning the Web Pages for the Bamboo Project. These Web pages are rooted by index.htm.

In this example, *index.htm* is the root HTML document for the Bamboo Web page. *Index.htm* contains links to other HTML files which are sections of content of the Bamboo Web pages. Each of these HTML files for section content also contains a link that points to *index.htm*. After each step in Figure 7, we checked in the Bamboo Web pages using the command: *chrysant ci index.htm*, so that the structure container of *index.htm* got a new version, as well as the content of the changed files in the hypertext network and affected other structure containers.

We should note that the structure container of *index.htm* has 4 versions, from version 1.1 to 1.4, though the content of *index.htm* has not changed since version 1.1. That is how we treat the version history of a hypertext resource in Chrysant: we version the structure container and the content of a hypertext resource separately. In this example, the content of *index.htm* always has version 1.1. We use the command, *chrysant log index.htm*, to display the version history of the structure container of *index.htm*. For example, the version 1.4 of *index.htm* is shown as follows.

index.htm, rev 1.4: pankai | 2004-03-06 20:53:49

~~~index.htm(Content), Version:1.1, pankai, 2004-03-06 20:51:51

~~~bamboo_files/soe.css Version:1.1, pankai, 2004-03-06 20:51:50

~~~bamboo_files/ucsc.gif, Version:1.1, pankai, 2004-03-06 20:51:50

~~~bamboo_files/admin.gif, Version:1.1, pankai, 2004-03-06 20:51:51

~~~news.htm(SC), Version:1.4, pankai, 2004-03-06 20:53:49

~~~cont_model.htm(SC), Version:1.4, pankai, 2004-03-06 20:53:49

~~~generator.htm(SC), Version:1.2, pankai, 2004-03-06 20:53:49

~~~demo.htm(SC), Version:1.2, pankai, 2004-03-06 20:53:50

~~~reference.htm(SC), Version:1.1, pankai, 2004-03-06 20:53:50

~~~downloads.htm(SC), Version:1.1, pankai, 2004-03-06 20:53:50

~~~credits.htm(SC), Version:1.4, pankai, 2004-03-06 20:53:50

The results displayed above show all the resources linked by version 1.4 of the *structure container* of *index.htm* and their version information. In the results, the *(content)* after a resource name indicates it is the content of a hypertext resource, and the *(SC)* after a resource name indicates it is a structure container of a hypertext resource. In this example, we can see that version 1.4 of the structure container of *index.htm* contains version 1.1 of the *index.htm* content, and it links to the structure containers of other hypertext resources at version 1.1, 1.2, 1.4 respectively.

Using the command, *chrysant sdiff index.htm 1.1 1.4*, we can see the structure difference between version 1.1 and version 1.4 of *index.htm*. The results of this command are as follows:

index.htm(content): 1.1 <--> 1.1

bamboo_files/soe.css: 1.1 <--> 1.1

bamboo_files/ucsc.gif: 1.1 <--> 1.1

……

bamboo_files/admin.gif: 1.1 <--> 1.1

**news.htm(SC): 1.1 <--> 1.4*

**cont_model.htm(SC): 1.1 <--> 1.4*

**credits.htm(SC): 1.1 <--> 1.4*

+generator.htm(SC): NULL <--> 1.2

+demo.htm(SC): NULL <--> 1.2

+reference.htm(SC): NULL <--> 1.1

+downloads.htm(SC): NULL <--> 1.1

The '+' symbol before the resource name in the results means this node appears in the latter version of the structure container, but not in the former version. The '-' symbol, not appearing in this example, has the contrary meaning to the '+' symbol. The '*' symbol indicates that this node has different versions in the two versions of the structure container.

4.3 Discussion

4.3.1 Full URLs in HTML

In our implementation, the URLs that point to resources on another site, e.g. the URLs starting with *http://*, are ignored by the HTML parser, since remote resources are out of the versioning scope. There are two issues here. First, some of the full URLs may actually indicate local resources. In this case, those local resources will not be versioned, since Chrysant does not know the base URL of the resources it is managing, and hence cannot determine if the base of a URL corresponds to the local server. Second, web pages can be distributed across multiple web sites for an enterprise. By far, Chrysant can only version local resources on one server. One possible solution to these two issues is that a mapping list can be created in the Chrysant repository. This list maintains the mappings between the full URLs, which point to local resources or resources on the web servers in the enterprise, and the local resource paths on each web server. When Chrysant checks in an HTML document, the full URLs in it will be used to locate the target resources on local machines by looking up the mapping list, so that the whole hypertext network can be checked in.

4.3.2 Too Many Versions Due to Change Propagation

An issue in the Chrysant approach is that, since changes to nodes will propagate through the hypertext network, there will be many versions generated. Some versions are necessary, e.g., those versioning the relationships between the main web page and its subsection pages, but some are not, e.g., those links created for reference purpose or convenience for navigation purpose. Applying version selection rules in Chrysant is a solution to this issue. For those links for reference or navigation purpose, version selection rules can be used on it instead of versioning them. For example, through the version selection rule on a link, the parent node can select the latest version of its child node, so that when the child node has a new version, the structure container of the parent node does not have to have a new version created. How to decide which links should use version selection rules, which links should be versioned is another issue. One solution to this issue is the user's interference. The user will decide which approach should be used for every links. The other solution is that a 'linkType' parameter can be added to the tags that link to other resource, and Chrysant will know which versioning approach to use based on the 'linkType' for every link.

4.3.3 Dynamic Pages

A limitation of Chrysant is that it can not handle the dynamic web pages, e.g., those web pages that contain the Javascript or those web pages generated by CGI programs. A HTML parser does not work well for them, since links will be generated dynamically. The possible solution can be the user's help to build the relationship between those web pages with the pages they actually link to. Or the relationships can be found out via dynamic analysis on the running history of the web application.

5. CONCLUSION AND FUTURE WORK

In this paper, we present Chrysant, a system that provides general-purpose hypertext versioning capability to hypertext systems with embedded links. We describe the architecture of Chrysant and a mechanism for versioning the structure container and the content separately for a hypertext resource. An algorithm, AF-BTU, is introduced in this paper to version a hypertext network as a whole. As an application case, HTML versioning is implemented with Chrysant, which has practical usage.

Our future work will include more design considerations for structure versioning, including metadata and directory versioning. It is interesting to version an environment mixed with directories, metadata, and more than one type of hypertext documents with all of them under version control.

Chrysant should support version selection rules and external link versioning in the future. A general link representation in the filesystem and in the repository will be designed.

In the future, we anticipate adding features for workspaces, locks and branching to make Chrysant a complete version control

system. Chrysant will also need to support versioning of web pages that are distributed across multiple web sites.

Link properties will also be studied. Links should be classified by the degree of cohesiveness to which they relate two resources. Based on it, a finer-gained hypertext versioning policy can be designed and applied.

6. ACKNOWLEDGMENTS

This project is supported by the National Science Foundation under Contract Number NSF CAREER CCR-0133991. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We also want to thank Sung Kim for his help on the HTML parser.

7. REFERENCES

[1] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," in *Network Working Group Request for Comments: 2396 Standards Track*, 1999.

[2] W. F. Tichy, "RCS - A System for Version Control," *Software-Practice and Experience*, vol. 15, pp. 637-654, 1985.

[3] S. Dreilinger, "CVS Version Control for Web Site Projects," vol. 1999, 1998.

[4] A. L. Glasser, "The Evolution of a Source Code Control System," presented at Software Quality and Assurance Workshop, 1978.

[5] Subversion, "Subversion project home page," vol. 2001, 2004, http://subversion.tigris.org.

[6] D. Leblang, "The CM Challenge: Configuration Management that Works," in *Configuration Management*, W. F. Tichy, Ed. New York: Wiley, 1994, pp. 1-38.

[7] D. Gordon and E. J. Whitehead, Jr., "Containment Modeling of Content Management Systems," presented at Metainformatics Symposium 2002 (MIS'02), Esbjerg, Denmark, 2002.

[8] Apple Computer, *HyperCard Script Language Guide*. Reading, MA: Addison-Wesley, 1988.

[9] 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, pp. 820-835, 1988.

[10] K. M. Anderson, R. N. Taylor, and E. J. Whitehead, Jr., "Chimera: Hypertext for Heterogeneous Software Development Environments," *ACM Transactions on Information Systems*, vol. 18, pp. 211-245, 2000.

[11] C. C. Marshall, F. G. Halasz, R. A. Rogers, and W. C. Janssen, Jr., "Aquanet: a hypertext tool to hold your knowledge in place," presented at Third ACM Conference on Hypertext (Hypertext'91), San Antonio, Texas, 1991.

[12] U. K. Wiil and J. J. Leggett, "The HyperDisco Approach to Open Hypermedia Systems," presented at Seventh ACM Conference on Hypertext (Hypertext '96), Washington, DC, 1996.

[13] K. Østerbye, "Structural and Cognitive Problems in Providing Version Control for Hypertext," presented at Fourth ACM Conference on Hypertext (ECHT'92), Milano, Italy, 1992.

[14] E. J. Whitehead, Jr., "Design Spaces for Link and Structure Versioning," presented at Hypertext 2001, The Twelfth ACM Conference on Hypertext and Hypermedia, Arhus, Denmark, 2001.

[15] E. J. Whitehead, Jr., "Uniform Comparison of Data Models Using Containment Modeling," presented at Hypertext 2002, The Thirteenth ACM Conference on Hypertext and Hypermedia, College Park, MD, 2002.

[16] E. J. Whitehead, Jr. and D. Gordon, "Uniform Comparison of Configuration Management Data Models," presented at 11th International Workshop on Software Configuration Management (SCM-11), Portland, Oregon, 2003.

[17] D. L. Hicks, J. J. Leggett, P. J. Nurnberg, and J. L. Schnase, "A Hypermedia Version Control Framework," *ACM Transactions on Information Systems*, vol. 16, pp. 127-160, 1998.

[18] J. Travis, "El-Kabong HTML Project Homepage," 2004, http://ekhtml.sourceforge.net.