Version Management as Hypertext Application: Referent Tracking Documents

W. Eliot Kimber
Senior Consulting Engineer
ISOGEN International, A DataChannel Company
email eliot@isogen.com
web www.isogen.com, www.datachannel.com

Steve Newcomb
President
ISOGEN International, A DataChannel Company
email srn@techno.com
web www.isogen.com

Peter Newcomb
President
TechnoTeacher, Inc.
email peter@techno.com
web www.techno.com

ABSTRACT

Presents a methodology for managing and tracking change of information objects over time that enables efficient and accurate maintenance and management of links among information objects. This methodology moves most of the burden of version management from the storage management layer (e.g., the "RCS" Revision Control System) to the semantic layer (link management), recognizing that, ultimately, the decision that a set of information objects represent different versions of the same logical or abstract object can only be made by a human as a matter of opinion. RCS (or its equivalent) is still used to maintain version knowledge, but only on the version management data, not the storage objects that are themselves versioned. Except as explained below, something like RCS is just another storage manager.

The purpose of this paper is to outline the basic concepts of a system design that will guarantee that all references and their valid referents will remain validly connected together, both mechanically and semantically, across all versions of all resources in a revision-control-managed technical information set, at any scale, and at the lowest possible cost over the long term.

While the implementation approach described uses the facilities of the HyTime standard, the general approach can be implemented using any equivalent mechanism, such as the extended form of "XLink" (as currently drafted).

KEYWORDS

version management; hypertext; change; referent tracking document; HyTime; XLink; XPointer

PROBLEM STATEMENT

Many large and complex technical information sets are subject to constant maintenance, which includes updating, enhancement, improvement, correction, reorganization, and expansion. In order to manage constant change in such a way as to maintain quality, rationalize workflow, and control costs, such information sets are usually placed under a software-enforced management policy often called ''Revision Control''.

Revision control systems do many different things in many different contexts, but most revision control systems, at a minimum, provide a means whereby the evolution of individual information assets can be seen as a chronological sequence of numbered versions. In other words, a revision control system allows its users to access any version of any asset, by means of the name of the whole sequence of revisions, in combination with the ''version number'' of the specific desired revision of that asset. The ''individuality'' of assets under revision control inheres in the fact that the whole sequence of revisions of a ''single'' asset is addressed by means of the name of the sequence.

The fact that revision control systems make many revisions appear to be a single abstract asset causes confusion for owners and maintainers of complex technical information sets in which there are many cross-references, traversable hyperlinks, and re-used components. All such references necessarily refer to information in particular revisions, not to all revisions in general. Furthermore, at any given moment in the revision history of a set of resources, a given referent may jump from one abstract asset to another. Keeping all references in working order from one release to the next has become one of the most perplexing, embarrassing, and expensive problems in the information management industry. Up to the present time, no out-of-the-box document management solution has addressed the problem adequately, much less optimally.

The act of creating information necessarily involves the change over time of the information as it is revised and updated over its life. This is the general problem of change management or version management: the maintenance of multiple versions of a given abstract information object and the maintenance of the version-to-version relationships of those objects. (For the purposes of this discussion, the term ''information object'' means any uniquely-addressable bit of information, from a single character to an entire storage object [e.g., XML document]. This paper limits its scope to data stored using XML syntax to keep the discussion as simple and focused as possible. However, nothing in this paper is necessarily limited to the management of XML-based data—it can be applied to data of any sort stored in any manner, given appropriate addressing facilities.)

Before computers, version management was merely a matter of keeping good files, that is, keeping each new physical copy, labeling it properly, and storing it in such a way that its relationship to other parts was clear (or at least determinable). ''Documents'' were well-defined sets of pieces of paper with reasonably obvious boundaries and relationships to each other. Various simple and obvious binding and collecting techniques could be used to keep the parts of a document together and clearly distinguish them from the parts of other documents. To lose a version of a document one had to physically destroy or misplace the paper itself, otherwise the version would persist until the paper (or sheep skin or mud tablets) decayed into oblivion. Version-to-version relationships were normally only maintained at the document level—determining and tracking changes at the component level was limited largely to scholarly analysis and legal documents.

With the advent of computers and computer-based document processing, the version management problem appeared to get much more difficult. While computer files are physical objects (magnetic states in metal particles, pits in aluminum films, etc.), they are much easier to destroy than paper documents. The typical use of computers sees one overwriting old versions of files with new versions whenever the new version is saved from the editor. The ability of computers to represent and manipulate complex structures made it possible to track changes in the details of these structures in a way that had huge potential benefit. The advent of hypertext turned documents from singular, informal things into complex systems of interconnected and interdependent information objects that must be formal and precise in order to function properly. The advent of hypertext also raised the specter of creating references to objects that may not exist in their current form forever. The ability to relate individual components of different versions together brought with it the overhead of tracking the relationships themselves.

Essentially, computers created a situation in which it is necessary to distinguish the management of storage objects from the management of the abstract semantic objects they contain. In a purely paper-based environment, there is no useful distinction between the physical and abstract because the physical is all there is. In a computer, there is a clear and vital distinction between the physical storage (e.g., a stream of bytes or string of characters) and the abstractions they represent (e.g., XML elements). To manage computer-based data completely it is necessary to manage both the physical instantiations and the abstractions.

To date, no system of which the authors are aware has solved the version management problem in a completely satisfactory way. In particular, there has been no satisfactory way to manage relationships along with storage objects. The approach defined in this paper attempts to satisfy both base storage object version management and relationship version management as completely as possible by adding to existing storage object management a technique for managing the relationships among the semantic (abstract) objects inherent in the storage objects. The approach uses and depends on storage-object versioning systems like RCS to manage the identity of and access to logical objects. It does not require the use of systems like RCS to manage versions of other storage objects but it does not preclude their use either. The only requirement is that every version of a storage object be maintained and be separately addressable. Whether this requirement is satisfied by creating a new file name for each storage object version or through the use of a system like RCS is irrelevant to our approach, although it may have profound practical implications for implementations.

The traditional approaches to version management can be grouped into the following classifications:

Latest version only In this approach, only the latest version of a given storage object (and thus the components of the storage object) are maintained formally (although older versions may be maintained for some period of time as a side effect of backup policies, for example). This is the normal operating mode for most computer users, who blithely overwrite their old versions at the end of editing sessions. This approach clearly does not solve the version management problem. This approach produces a system that has no past, only a present. This type of system is typified by personal computer office systems and Web sites, where there is no standardized or conventionalized way of storing and referring to older versions of objects.

Storage object name discipline In this approach, new versions of documents are stored with new file names, usually using some sort of naming discipline that defines or implies the time-dependent relationship of one file to another (e.g., mydoca.xml, mydocb.xml, etc.). This approach has the advantage of not throwing away the past and therefore maintaining old versions indefinitely. It has the disadvantage that it requires knowledge of the naming discipline (if there is one) and consistent application of the discipline. It is also at the mercy of human error and laziness. This approach cannot always clearly represent the division or combination of storage objects except through the use of fairly sophisticated naming disciplines. It is incapable of representing the version relationships of components of storage objects. Data managed this way may not be interchangeable across different operating systems or file systems due to limitations in the file naming facilities of different systems. For example, a Unix-style discipline like mydoc.sgm.0.1 cannot be reliably translated to an MSDOS system. This approach is satisfactory for managing relatively stable systems of storage objects, as demonstrated by its use in managing UNIX storage objects for many years, but is not adequate by itself to solve the version management problem.

Storage-object delta-chain managers In this approach each new version of a given storage object is differenced against the previous version and only the differences are stored. This type of system is typified by the GNU Revision Control System (RCS). Systems of this type do an excellent job of storing storage objects so that different time-specific versions can be easily accessed, but they do nothing to relate different versions of their components together (except to the degree that such relations can be inferred by examining the differences between two versions). These systems are essentially optimized storage managers that minimize the amount of space used to store different time-specific versions of the same storage object. This is a very useful facility, and, as will be discussed later, critical to efficient version management, but it does not, by itself, solve the version tracking problem, especially with respect to relationships among data components. For example, RCS (and Concurrent Versioning System [CVS], a more complete data management system built on top of RCS) cannot track the fact that one storage object at time T(0) is split into two new storage objects at time T(1). It also cannot track that elements A and B at time T(0) are combined into single element C at time T(1).

Element managers In this approach XML documents are managed at the element level by making each element an object. This type of system is typified by Chrystal's "Astoria" product. These systems provide version tracking by maintaining versions of each object within the repository. They do essentially the same thing as delta-chain storage object managers, but at the element level rather than at the document (storage object) level (or, if you prefer, they treat each element as a separate storage object). This approach works as long as all modification of documents is done through the repository so that it has knowledge of the modifications. However, if an export-change-import model is used, there is no way, in the general case, for the system to reliably relate the elements in one version of a document to the elements in the new version. Even if the system embeds some sort of markers into the data on export, there is nothing to prevent their modification or removal during the editing process because it occurs outside the control of the repository. Thus, these repositories are essentially persistent editors (persistent in the sense that their internal representation of the original XML documents is made persistent, with the data never reconstituted into XML except for export out of the repository). This type of repository also has the problem that the overhead for each object is high and must be present even when many, if not most, of the elements are of no interest from a version management perspective (either because they have not changed or because their changes are not significant to the data owner). One solution, which is to limit the level at which elements are decomposed, also limits the ability to track element-level versions. While systems of this type are useful as persistent editors, they also fail to solve the version management problem.

In-line version representation In this approach, the representation of different versions of an information object is done within the information object itself, that is, using some syntactic feature of the data representation syntax, such as marked sections (in SGML), paired processing instructions, or elements. This approach has the advantage that it is not dependent on specific software or hardware and, because its use is controlled by the author, only those changes that are significant and at the granularity that is most useful or appropriate need be reflected. This approach has the disadvantage that it adds significant overhead to the data itself, often swamping the volume of core data with version representation information. It also increases the processing overhead for all users of the information, regardless of which version or versions they are interested in.

This approach is fundamentally a special case of the more general problem of applicability, which is the definition of the conditions to which a particular information object applies. While many applicability management approaches can also solve the version management problem, version management is usually seen as being a separate domain from other uses of applicability, such as relating information objects to specific operating platforms, part numbers, or user categories. In general, the version applicability of an information object is orthogonal to any other applicability it may have. Another flaw is that, like delta management, most in-line version representation approaches (certainly those that are easily authorable) cannot represent the splitting of a single storage object into multiple objects.

Thus, no existing XML version management approach succeeds in completely solving the version management problem, although many, if not all, solve parts of the problem well enough to be useful in particular circumstances. To be completely satisfactory, a version management system must do the following:

1 Maintain all versions of storage objects such that no data is ever destroyed except by explicit choice of the data owner.
2 Provide a facility for identifying and referring to abstract things independent of their locations in particular storage objects.
3 For each abstract thing, relate the versions in time of the thing so that any or all versions can be accessed

ASSUMPTIONS AND GIVENS

The following are assumptions or givens on which our version management approach depends.

The storage object (e.g., file, document entity) is the fundamental unit of storage. All data access is through storage objects, i.e., the names of storage objects constitute the global name space that is the basis of all physical addressing.

Once created, a storage object is static and immutable. That is, there is no sense in which a storage object can change, any more than a printed page can change. This assumption does raise certain existential questions about what constitutes a "change" and what does "the same" mean, but for the practical purposes of version management, we choose to leave those existential questions to philosophers. For our purposes here, one perfectly usable definition of "change" is "any activity whose effect alters the checksum of the storage object." (In some situations, of course, this definition may be insufficiently sophisticated.) The assumption of storage object immutability makes the system simpler and makes version management tractable.

Storage objects have identity in space and time (they must be unalterably bound to their creation times). This is a recognition of the fact that storage objects are, ultimately, physical things and therefore have all the qualities of physical things, then most important of which are identity in space and time. Storage objects are physical things because they exist on storage media as physical things, such as magnetic field domains or pits in aluminum film or holes in long strips of paper. This means that a given storage object can exist in only one place at any given point in time. If you copy a storage object, you have two distinct storage objects. That you may consider them to be "the same" is a policy that you impose. Thus, when people ask "Do you have the file?" they are really asking "Do you have a copy of the file?".

This idea of "one thing one place" can be taken a bit farther to say that one important aspect of a storage management system is that it provides the complete and dependable illusion that a given storage object really does exist in only place, even though it may actually be copied or mirrored in several places. In the ideal world, every storage object would have exactly one storage location that was always reachable from anywhere in "the network'' (where ''the network'' is whatever scope of access is of interest, from the network of computers in your home to the Internet). The World Wide Web provides a taste of this ideal where it would be possible for every computer (and thus every storage device) to be on the net and every file to exist in exactly one place. Of course practical realities make this naive view impossible. Huge effort has gone into solving the problem of making distributed systems appear to be single computers and this effort shows that it is possible to build a system in which the practical details of data storage can be hidden so that it can appear to users of the system that things exist in exactly one place even though they may actually be copied on many physical systems. For the purposes of this paper, we do not distinguish between a simple single-computer file system on a single hard disk and a highly sophisticated distributed file system such as are often used for high volume, high traffic Web sites. Our approach can be used on the former without difficulty and depends on the latter to provide scalability. That is, our approach treats file systems, however implemented or distributed, as consistent services that provide access to files that are, as far we can tell, single physical objects, even if there are more layers of abstraction under the covers that we cannot see.

CHANGE IS AN ILLUSION

When we use an editing tool like Emacs or Notepad or Microsoft Word™ we normally say that we are ''changing'' the document we are editing. However, this is a fiction very much like the fiction that movies actually consist of moving pictures.

When you create a storage object (e.g., a file), it is static and unchanging. It has identity in that you can distinguish it from all other files on your system (or any other system anywhere in the universe). If you bring that file into an editor, you are really copying the file's data into the local storage space of the editor (its ''memory'') and manipulating that copy. If you save the data using the filename of the original file you have not changed the original file. Rather, you have destroyed the original and replaced it with the new copy. This is exactly analogous to pulling a paper document from a file, retyping it onto new pieces of paper, burning the original, and putting the new copy back in the file—there is no evidence that the original copy ever existed and no hope of getting it back.

If, by contrast, you saved the new copy with a new file name, the original copy is undisturbed and continues to exist. As long as you don't explicitly delete the original copy it will always be available, in its exact current state until the end of time. By the same token, the new copy, once saved to a file, becomes constant and immutable.

Note that, at the level of pure data storage mechanics, there is absolutely no difference between creating a copy of a file that is considered to be a new version in time and creating a copy of a file that is considered to be a distinct ''document''. It is impossible, in the general case, to distinguish these two forms of copy simply by looking at the storage objects themselves—there must be some separate indicator of the version relationships, whether it is the use of particular filenames, the storage of the new copy in a particular RCS archive, or the use of an explicit hyperlink that relates the two versions.

Note further that the idea of ''document'' as a logical thing (as opposed to ''document'' as defined by the XML and SGML standards) is a subjectively applied classification of information objects. For example, the paper you are reading now is one instantiation (of many) of the idea of a paper on the subject of using hyperlinks to represent versions. The paper you are reading is no more ''the document'' than any of the various earlier drafts were. Even more, it is a matter of differing opinion as to what physical things constitute versions or components of this paper over time. Clearly, the idea of ''document'' is a highly perspective-driven thing. No existing version management approach is capable of fully representing the complexity of a document like this paper that was developed in many media by different people in different ways.

Every storage object has identity—this is a fundamental and necessary property of storage objects (and objects in general). This also means that every storage object exists in exactly one place at exactly one time—if you create a copy of a storage object and all you do is change its storage location (but change no other properties, including its recorded creation date), it is still a different storage object, even if you consider it to be ''the same'' as the original.

Thus, change is an illusion or fiction that obscures the true nature of the way we tend to manage data, which is to throw away old versions. It also means that change, or rather, the representation of change over time, is a function of assertions made about sequences of storage objects over time and not an inherent property of the storage objects themselves. Operating systems could provide built-in facilities for relating storage objects together to assert version relationships, but few do. In any case, such facilities would not address the requirement to assert version relationships among the semantic components of storage objects (e.g., between elements in different documents).

MANAGING HYPERLINKS OVER TIME

When documents are standalone objects (like pieces of paper), the full difficulty of version management is not encountered. However, when documents are actively linked to each other, they become an interdependent system of objects and serious version and information management problems are exposed.

One commonly-cited problem with hypertext is the "broken link" problem. This is the problem whereby a link that worked at time T(0) no longer works at time T(1) because one or more anchors of the link no longer exist at their former locations.

From the foregoing discussion of storage objects and versions it should be clear that this problem is caused by throwing away old versions of documents. If the versions created and linked at time T(0) continue to be accessible, then the links to their components cannot be broken. If, at time T(1) you throw away the original documents and replace them with new copies, you have committed a fraud, substituting a false, unexpected, unwanted thing (the T(1) versions) for the original, true, known things (the T(0) versions). This type of behavior can only lead to pain and frustration, when attempting to maintain hyperlinks to the components of the T(0) versions.

Link breakage is a problem for which the solution is easy: never throw anything away. This solves the physical problem of maintaining referential integrity within the data set (once established). However it doesn't solve the practical problem of how to react to change.

The real problem is that for every reference within a system of information objects, when a new version of any storage object in the system is created a decision must be made as to how to react to this change. For each reference to a given information object for which there is a new version there are three possible decisions:

1. The new version is of no interest and no reaction is necessary; the reference is unchanged.
2. Create a new version of the reference that only points to the new version
3. Create a new version of the reference that points both to the original version and to the new version

All three decisions are equally reasonable and will be appropriate depending on the nature and use of the information and the general policies in effect. For example, a brief for a person accused of a crime committed 20 years in the past must point to the version of the law in effect at the time of the crime, whereas a police policy related to such cases must point to the latest version of the law, and scholarly commentary on the relevant body of law must point to both (or, potentially, all) versions in order to explain how and why they have changed over time.

Clearly no generally useful version management system can impose one of these three policies—it must allow any choice. It would probably be useful if it optimized access to the latest version as that is usually (but not always) the most common policy.

This problem is compounded by the use of meaningful names for storage objects where the name itself becomes imbued with value and thus becomes more important than the data it happens to identify at any given time. For example, the name "http://www.techno.com/index.html" is a reference to the index.html file that is the root of the Techno.com Web site. If the "never throw anything away" policy was instituted on the Techno.com Web site, when you went to the above name you'd get the first index.html file they ever created—by necessity each later version would have a different name. This suggests that there is a class of information objects that need only have a present and not a past or that, rather than giving the new version a new name, the old version is given a new name and the new version gets the old name. This is roughly analogous to recording a broadcast television program: the original broadcast can only be accessed once: it exists at a precise point in time that, barring time travel, can only be seen once. However, a recording of the broadcast can be accessed many times; the act of recording the program is equivalent to copying a file to a new location (and thus giving it a new name).

REPRESENTING VERSION-TO-VERSION RELATIONSHIPS

If the fact that two information objects represent versions in time of some abstract, logical object is an opinion, then it follows that the representation of version relationships is appropriately done with hyperlinks, because the primary purpose of hyperlinks is to assert and represent opinions about the relationships among things. Furthermore, these links must, by necessity, be completely separate from the storage objects and data elements they relate; otherwise new opinions about versions could not be stated without also changing the storage objects about which the opinions were being stated. While the benefits of this approach are significant even when applied to a single data type such as XML, given an addressing mechanism that can address anything, the approach can be applied to data of any and all types.

The HyTime architecture defined in the HyTime Standard (ISO/IEC 10744:1997) provides just such a hyperlink and addressing mechanism. The methodology presented here takes advantage of these facilities as an implementation approach. However, the approach can be implemented using any linking and addressing mechanism that has the same characteristics. XLink and XPointer could be used, although the potential solution would not necessarily be as flexible or all-encompassing (but it would be sufficient to implement this approach at least within an XML-only environment). One limitation of XLink today is that the XPointer addressing method is only defined for the addressing of XML documents—it cannot be used in any standards-conforming way to address components of non-XML data. This is a deficiency in the way that the XPointer semantics are specified, not an inherent limitation in the XPointer mechanism. That is, if XPointers were defined in terms of operations on a generic, specializable data model, they could then be applied to any data type described using that generic model. Because the HyTime semantics are defined in terms of such a generic data model, HyTime syntax and semantics can be applied to data of any type.

The key problem is the creation of representations of abstract things that are the intended targets of references such that a reference to a single, unchanging name over time enables the addressing of the set of semantic objects that make up the thing at any point in time, present or past.

The abstract things are called "referents" in this design. Each referent is represented by a single storage object (file) which is a single XML document consisting of a single hyperlink. This hyperlink points to all the semantic components that make up the referent. These documents are called "referent tracking documents" (RTDs) because they bind time-specific versions of referents to their component objects and track them as they change over time. The filename of the RTD serves as the persistent name of the referent over time. Each RTD is managed over time as a single revision chain in RCS or its functional equivalent. Any time-specific version of a referent can be determined by retrieving the appropriate time-specific version of the referent's RTD document from the RCS archive.

To illustrate the mechanism, we start with the following two XML documents, doc001.xml and doc002.xml, which are documents being developed by some set of human authors. Document doc001.xml:

```
<?xml version="1.0"?>
<!DOCTYPE doc SYSTEM "doc.dtd">
<doc><p>See thing one.</p></doc>
```

Document doc002.xml:

```
<?xml version="1.0"?>
<!DOCTYPE doc SYSTEM "doc.dtd" []>
<doc>
<p id="e001">This is all about thing one.</p>
</doc>
```

At time T(0), the two documents are not explicitly connected, although the author of doc001.xml appears to intend a reference to doc002.xml.

At time T(1), the author of doc001.xml creates a new version, doc001-1.xml, and adds an explicit hyperlink to the element "e001" in document doc002.xml:

```
<?xml version="1.0"?>
<!DOCTYPE doc SYSTEM "doc.dtd" [
<!NOTATION xml SYSTEM
  "http://www.w3.org/TR/1998/REC-xml-19980210">
<!ENTITY doc002 SYSTEM "doc002.xml" NDATA xml>
]>
<doc>
<p>
See <xref target="e001" doc="doc002">thing
one</xref>.
</p>
</doc>
```

The storage object that contains the target element has been declared as an external unparsed entity. The attributes of the "xref" element bind the element ID ("e001") to the entity (doc002) to establish the complete address of the desired target element. (This could also have been done with a single URL ala XLink. The HyTime-defined syntax is used here in part to emphasize the distinction between the storage object part of the address and the element ID part of the address. There is no functional difference between the syntax used here and the equivalent XPointer-based URL.)

Note that this example uses a direct, hardened reference from the first document to the second.

At time T(2) a new version of doc002.xml, doc002-1.xml, is created:

```
<?xml version="1.0"?>
<!DOCTYPE doc SYSTEM "doc.dtd" [
]>
<doc>
<p id="e001">
This is all about thing one.
</p>
<p id="e002">
This is all about thing two.
</p>
</doc>
```

Note that the reference made from document doc001-1.xml is still pointing to doc002.xml and does not reflect in any way the new document doc002-1.xml. At this point the author doc001-1.xml has to make a choice: do nothing and continue to point to doc002.xml, create a new version that points to doc002-1.xml, or create a new version that points to both doc002.xml and doc002-1.xml. At time T(3) the author decides on the second option, creating document doc001-2.xml:

```
<?xml version="1.0"?>
<!DOCTYPE doc SYSTEM "doc.dtd" [
<!NOTATION xml SYSTEM
  "http://www.w3.org/TR/1998/REC-xml-19980210">
<!ENTITY doc002-1 SYSTEM "doc002-1.xml" NDATA xml>
]>
<doc>
<p>
See <xref target="e001" doc="doc002-1">thing
one</xref>.
```

```
</p>
</doc>
```

At this point, the original referencing document has been updated to react to the change in its target. Both versions of document doc001.xml maintain referential integrity. However, there are two problems. First, there is nothing in this system of documents that relates any of these documents together as being versions of each other. Second, there is nothing that formally establishes the relationship of the element "e001" in doc002.xml and element "e001" in document doc002-1.xml. The fact that they have the same ID value is mere coincidence and is not a necessary condition. For example, at time T(4) the author of doc002-1.xml could make this change, creating new document doc002-2.xml:

```
<?xml version="1.0"?>
<!DOCTYPE doc SYSTEM "doc.dtd" [
]>
<doc>
<p id="e002">
This is all about thing one.
</p>
<p id="e001">
This is all about thing two.
</p>
</doc>
```

The IDs of the two paragraphs have been swapped (perhaps because the author started to rearrange the order of the paragraphs by swapping the contents of the two paragraphs and then decided to put things back by simply reordering the paragraphs, forgetting about the original ID assignments, an easy thing to do when editing tools hide details like element IDs). The author of doc001-2.xml now has to make the same decision as before and the mechanics of reacting to it are the same—the change in the element IDs does not affect the process (after all, what's important is the content of the thing referenced, not its arbitrary identifier).

Again, there is no explicit relation between the different versions of the referent of the cross reference. While the referential integrity of the information has been maintained, the tracking and management of it over time has not been enabled.

Replaying the same scenario using referent tracking documents produces the following result. We start at the same time T(0) with two documents that are not yet explicitly linked.

At time T(1), the author of doc001.xml creates a new version, doc001-1.xml, and adds an explicit hyperlink to the element "e001" in document doc002.xml, but instead of creating a direct reference, the system creates a referent tracking document to represent the intended target of the reference. This reference tracking document, rtd001.xml, looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE referent SYSTEM "rtd.dtd" [
<!ENTITY doc002 SYSTEM "doc002.xml" NDATA xml>
]>
<referent members="m001">
<title>Thing one</title>
<nmsploc id="m001" locsrc="doc002"
        namespc="elements">e001</nmsploc>
</referent>
```

The document element is a hyperlink pointing to element "e001" in document "doc002.xml". The "nmsploc" element is a HyTime indirect location address that translates the local ID "m001" (first member) to the element "e001" in the document "doc002". This indirection is introduced here because it is absolutely required when the referents are in separate documents or the referent is addressed in a different way. (The XLink equivalent would be an extended link with two locator subelements that have the same value for their "role" attributes.)

The rtd001.xml document is checked into its RCS repository, becoming version 1.1 according to RCS' versioning numbering scheme. The "-u" option of the RCS "ci" command is used to maintain the latest copy of the file on the file system for convenience.

The document doc001-1.xml now looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE doc SYSTEM "doc.dtd" [
<!NOTATION xml SYSTEM
  "http://www.w3.org/TR/1998/REC-xml-19980210">
<!ENTITY rtd001 SYSTEM "rtd001.xml" NDATA xml>
]>
<doc>
<p>
See <xref target="rtd001">thing one</xref>.
</p>
</doc>
```

Now, instead of pointing directly to the doc002.xml document, the cross reference points to the referent tracking document "rtd001.xml", which, by the rules of HyTime, is interpreted as a pointer to the "referent" element itself. Being a hyperlink, the referent element then enables navigation to the members of the referent, in this case the single element "e001" in document "doc002.xml". The key here is that the pointing is to the referent tracking document by file name, which will be persistent for the life of our information system.

The referent tracking document "rtd001.xml" now formally represents the logical or abstract thing the author of document doc001-1.xml intended the cross ref-

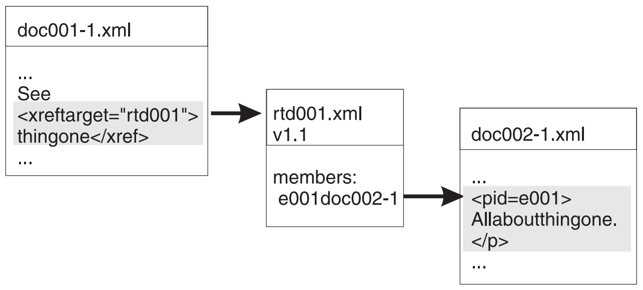

Figure 1 Connection between Source and Target of Link Through Referent Tracking Document

erence to point to. It essentially provides a physical proxy for the idea of, in this case, ''stuff about thing one''. The RTD shown is very simple, but it could of course be made more sophisticated, for example, containing descriptive metadata about the referent as an abstract idea.

To continue the scenario: at time T(3) the author of doc002.xml creates a new version, doc002-1.xml, to add information about thing two:

```
<?xml version="1.0"?>
<!DOCTYPE doc SYSTEM "doc.dtd" [
]>
<doc>
<p id="e001">
This is all about thing one.
</p>
<p id="e002">
This is all about thing two.
</p>
</doc>
```

By some process not yet defined, the system determines that document doc002-1.xml is in fact a new version of doc002.xml and determines that the element ''e001'' should be used as the member of this version of the referent rtd001.xml. The system creates a new version of rtd001.xml to reflect this:

```
<?xml version="1.0"?>
<!DOCTYPE referent SYSTEM "rtd.dtd" [
<!ENTITY doc002-1 SYSTEM "doc002-1.xml" NDATA xml>
]>
<referent members="m001">
<title>Thing one</title>
<nmsploc id="m001" locsrc="doc002-1"
namespc="elements">e001</nmsploc>
</referent>
```

This document is checked into the rtd001.xml archive, becoming version 1.2. Again this latest version is maintained on the file system for convenience.

At this point, the author of document doc001-1.xml must make a decision: do nothing and, by default, point to the new version of the referent (element ''e001'' in document doc002-1.xml) or refer to a specific version or versions of the referent tracking document in order to continue to point to a specific version or versions of the referent.

At this point, the value of the referent tracking document should be clear: it protects references from the creation of new versions of referents when the desired

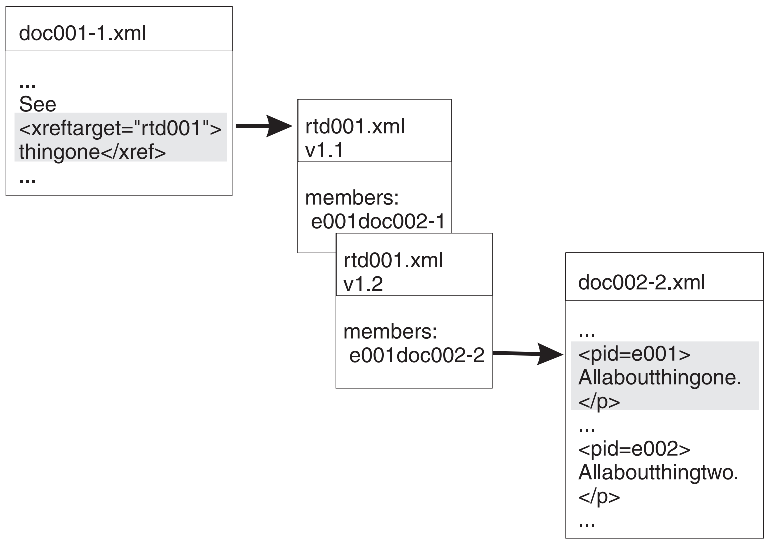

Figure 2 New Version of RTD Reflecting New Version of Referent

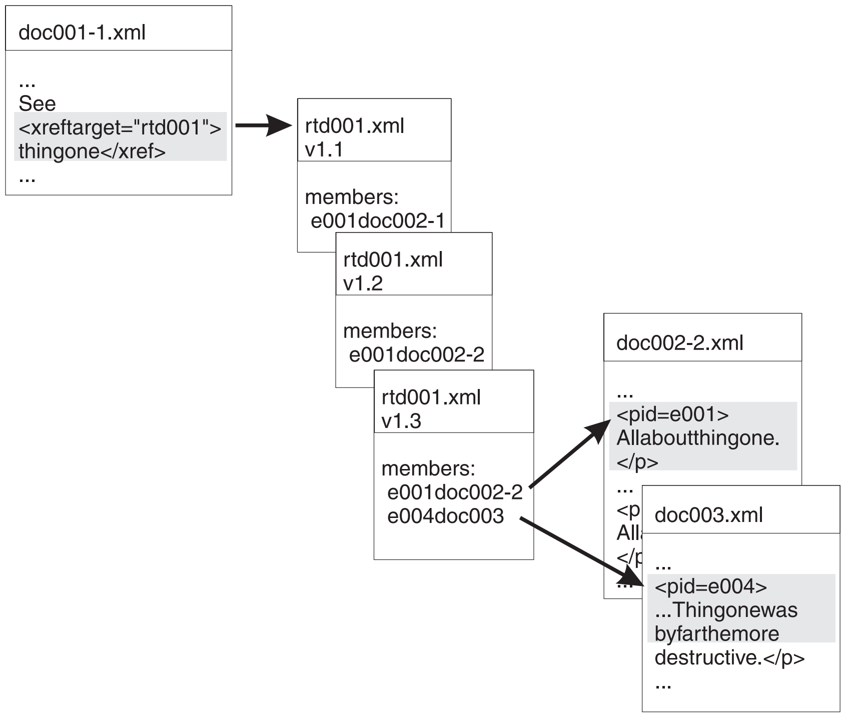

Figure 3 RTD Describing Referent Distributed Across Multiple Documents

result is to point to the latest version. It also serves to explicitly bind different versions of the referents together through the use of RCS to both maintain the different versions of the RTD document and to formally relate them in time.

The use of RCS for the referent tracking documents has another important aspect: it enables accurate recreation of the state of the system at any given point in time. Assuming that all storage objects are accurately bound to their creation time, the system can be queried to return only those storage objects in existence at a particular point in time, including the corresponding versions of referent tracking documents. Essentially, the time to be recreated is specified as a parameter to the RCS check out command.

The scenario as developed so far formally relates the two versions of the ‘‘thing one’’ referent but does not formally relate the different versions of the referencing document. This is easy to do simply by creating an RTD for document doc001, rtd002.xml. The initial version of this RTD document would be:

```
<?xml version="1.0"?>
<!DOCTYPE referent SYSTEM "rtd.dtd" [
<!ENTITY doc001 SYSTEM "doc001.xml" NDATA xml>
]>
<referent members="m001">
<title>doc001</title>
<nmsploc id="m001" namespc="entities">
doc001</nmsploc>
</referent>
```

Version 1.2 of this RTD document would be:

```
<?xml version="1.0"?>
<!DOCTYPE referent SYSTEM "rtd.dtd" [
<!ENTITY doc001-1 SYSTEM "doc001-1.xml" NDATA xml>
]>
<referent members="m001">
<title>doc001</title>
<nmsploc id="m001" namespc="entities">
doc001-1
</nmsploc>
</referent>
```

Likewise, the same would be done for document doc002.

At this point, we now have a complete system in which every ‘‘important’’ object (that is, objects we do or might want to reference) is described by a reference tracking document, providing a single, consistent mechanism for reference and access to arbitrary versions of documents. However, the scenario has not yet presented a case that would not have been satisfied by simply using RCS for all three documents. This case is the one in which the original referent is split across two or more storage objects.

At time T(4) the author of doc002 decides to split the document into two separate documents and spread the discussion of thing one between them. This creates a new document, doc003:

```
<?xml version="1.0"?>
<!DOCTYPE doc SYSTEM "doc.dtd" [
]>
<doc>
<p id="e004">
The Cat brought some friends,
Thing One and Thing Two. Thing
One was by far the more destructive.
</p>
</doc>
```

The RTD for thing one is revised to reflect this change:

```
<?xml version="1.0"?>
<!DOCTYPE referent SYSTEM "rtd.dtd" [
<!ENTITY doc002-1 SYSTEM "doc002-1.xml" NDATA xml>
<!ENTITY doc003   SYSTEM "doc003.xml" NDATA xml>
]>
<referent members="m001 m002">
<title>Thing one</title>
<nmsploc id="m001" locsrc="doc002-1"
namespc="elements">e001</nmsploc>
<nmsploc id="m002" locsrc="doc003"
namespc="elements">e004</nmsploc>
</referent>
```

At this point, the RTD is able to represent the distribution of the members of referent thing one across two documents. Neither RCS nor any single-file version management system could represent this distribution. It is also important to remember that the RTD is relating semantic components together, while the storage object versioning system is relating storage objects together. You need both forms of relation in a complete system.

THE PROBLEM WITH EDITING

In the foregoing scenario, the mechanism by which the system knows what the different versions of the referent ‘‘thing one’’ are is intentionally left undefined. This is because there is no reliable way to do this that does not involve humans explicitly stating what things are versions of what. This is because of what might be called the ‘‘editing problem.’’

The basic problem is that of tracking the details of changes made during an editing session. There does not appear to be a general solution to this problem, although the fact that SGML and XML documents have rich structure does hold some promise for the use of structure-aware differencing that can make reasonably accurate inferences about how things have changed and what the correlation between information elements in two versions are. Unless every user action is journaled through an editing session, there is no reliable way to know how some original thing was transformed through the editing process. Even when actions are journaled, it may still be impossible for a computer, lacking understanding of the meaning of the information, to accurately deduce a true change. This means that in the general case it will always be necessary for the humans involved to state what the version-to-version relationships are.

Note also that there is a difference between simply identifying things that are new or changed and relating objects in one version to specific objects in another version. Identifying changes is well solved by existing differencing technology (although there are subtleties having to do with the distinction between purely syntactic changes, many of which may have no semantic significance, and changes that are truly significant). Relating changes across versions requires either sophisticated tracking of editing actions through an editing session or human identification of changes.

From a practical standpoint, there are several things that can be done to brighten this picture a little. First, element identification conventions can be enforced such that the same logical thing should have the same ID in all versions of the document that contains it. This allows computers to make a reasonably accurate and reliable guess about versions, although it cannot solve the problem completely. This approach can be further augmented by using indirect addressing within documents to explicitly state relations and maintain ‘‘persistent’’ IDs for things as they are reorganized over time. This adds another level of indirection beyond that provided by the RTDs themselves.

Another potential aid is to require each author to examine the RTDs that refer to things in the documents that author controls and is editing and to adjust them as necessary.

Note also that it is not generally necessary to create RTDs for all things that might be referenced. It is only necessary to have RTDs for those things that are actually referenced. It is also possible to create RTDs for older versions after the fact by synthesizing RCS archives for the RTDs that reflect the appropriate creation times.

FORMAL DEFINITION OF THE REFERENT TRACKING DOCUMENT VERSION MANAGEMENT SYSTEM

This section formally defines the components and conventions of the referent tracking document version management system. In this description, the term "information system" means a system of related information objects that are maintained as a unit for some non-trivial period of time. The RTD mechanism does not define or constrain how these information objects are individually stored or managed.

The essential components of the RTD system are:

1. For each "thing of interest", a single referent tracking document (RTD), managed as a single sequence of storage object versions using RCS or its equivalent. The storage object identifier for the RTD is constant with respect to the "thing of interest" for the life of the information system.
2. The content of the referent tracking document is a single hyperlink that serves to aggregate the semantic objects that make up a time-specific version of a single thing of interest. In HyTime terms, this hyperlink is an "aggregation link" (agglink). Because this link may aggregate multiple semantic objects, in XLink it must be represented using an extended link where the link element is one resource role and all other resources are the same role (e.g., "members", following the definition of the HyTime agglink link type). The RTD's hyperlink may have any content or attributes desired beyond those required to establish the base aggregation relationship. Good practice is to have some sort of title or descriptor that describes the thing of interest for the benefit of humans using the information system.
3. For each new version of the thing of interest, a new version of the thing's RTD is created reflecting the new constituents of the thing.
4. By default, unqualified references to RTDs are references to the latest version of the RTD, and therefore to the latest version of the thing of interest the RTD represents.
5. The system must provide facilities for referring to time-specific versions of RTD (and therefore, to time-specific versions of the things of interest that the RTDs represent). This could be through the use of formal system identifiers or, in an XLink context, through some form of URL that addresses time-specific versions of RTD documents (possibly using facilities based on IETF WebDAV).
6. The system should provide facilities for extracting time-specific views of the information system. Note that this facility is inherent in the use of software systems like CVS to manage all the information objects in the information system.

The only indirection required by the RTD approach is that represented by the referent aggregation links. This means that RTD-based information systems can be implemented using the facilities of the XLink and XPointer specifications.

SUMMARY

The referent tracking document version management system relies on the following key components:

1. The creation and management of a referent tracking document for each "thing of interest" in a system of documents where the referent tracking document has the same filename for the life of the information system.
2. The use of RCS or its equivalent to manage versions of the RTD documents over time.
3. The consistent use of RTD documents as the initial target of all references to things of interest.

The mechanism is not dependent on any particular hyperlinking or addressing mechanism as long as the mechanism used is equivalent to the HyTime facilities used in the examples in this paper.

The problem of tracking changes through the process of editing is probably unsolvable in the general case and is therefore simply a fact of life. System implementors must carefully analyze the accuracy requirements and work habits and patterns of systems in order to develop appropriate authoring conventions and practices for version identification and tracking. In particular, it is important to distinguish the need to simply identify changes from the need to relate different versions of the same logical information object. The former can be satisfied by existing differencing tools. The latter requires a system like that described in this paper.

While the authors have demonstrated that the system described here satisfies the requirements of complete version management with respect to the integrity of references, more experimentation is needed to understand how this type of approach works in practice and how existing tools can be adapted to support this approach.

APPENDIX: FULL DECLARATIONS FOR ALL EXAMPLES

The following declarations conform to the HyTime standard as corrected through TC1, still under development as ISO/IEC JTC1/SC34 document N1957. In particular, the location type "ENTLOC" was omitted from the standard as published. The PI-form of architecture use declaration is defined in Amendment 1 to 10744:1997.

doc.dtd declaration set for non-RTD scenario:

```
<?IS10744 arch
  name="HyTime"
  public-id="ISO/IEC 10744:1997//NOTATION AFDR
       ARCBASE Hypermedia/Time-based Structuring
       Language (HyTime)//EN"
  options="hylink refloc loctype"
  doc-elem-form="HyDoc"
  arch-bridge-form="HyBrid"
?>
<!ELEMENT doc
 (p+)
>
<!ATTLIST doc
 HyTime
   NAME
   #FIXED "HyDoc"
>
<!ELEMENT p
 (#PCDATA |
  xref)*
>
<!ATTLIST p
 id
   ID
   #IMPLIED
 HyTime
   NAME
   #FIXED "HyBrid"
>
<!ELEMENT xref
 (#PCDATA)
>
<!ATTLIST xref
 target
   CDATA
   #REQUIRED
 doc
   CDATA
   #REQUIRED
 loctype
   CDATA
   #FIXED "target IDLOC
           doc    ENTLOC"
 rflocsrc
   CDATA
   #FIXED "target doc"
 HyTime
   NAME
   #FIXED "hylink"
 anchrole
   CDATA
   #FIXED "refmark target"
 anchcstr
   CDATA
   #FIXED "self required"
>
```

doc.dtd declaration set for RTD scenario:

```
<?IS10744 arch
 name="HyTime"
 public-id="ISO/IEC 10744:1997//NOTATION AFDR
      ARCBASE Hypermedia/Time-based Structuring
      Language (HyTime)//EN"
 options="hylink refloc loctype"
 doc-elem-form="HyDoc"
 arch-bridge-form="HyBrid"
?>
<!ELEMENT doc
 (p+)
>
<!ATTLIST doc
 HyTime
   NAME
   #FIXED "HyDoc"
>
<!ELEMENT p
 (#PCDATA |
  xref)*
>
<!ATTLIST p
 id
   ID
   #IMPLIED
 HyTime
   NAME
   #FIXED "HyBrid"
>
<!ELEMENT xref
 (#PCDATA)
>
<!ATTLIST xref
 target
   CDATA
   #REQUIRED
 loctype
   CDATA
   #FIXED "target ENTLOC"
 HyTime
   NAME
   #FIXED "hylink"
 anchrole
   CDATA
   #FIXED "refmark target"
 anchcstr
   CDATA
   #FIXED "self required"
>
```

DTD declaration set for rtd.dtd:

```
<?IS10744 arch
 name="HyTime"
 public-id="ISO/IEC 10744:1997//NOTATION AFDR
      ARCBASE Hypermedia/Time-based Structuring
      Language (HyTime)//EN"
 options="hylink refloc loctype nmsploc"
 doc-elem-form="agglink"
 arch-bridge-form="HyBrid"
?>
<!ELEMENT referent
 (title,
  nmsploc+)
>
<!ATTLIST referent
 members
   CDATA
   #REQUIRED
 loctype
   CDATA
   #FIXED "members IDLOC"
 HyTime
   NAME
   #FIXED "agglink"
>
<!ELEMENT title
 (#PCDATA)
>
<!ELEMENT nmsploc
 (#PCDATA)
>
<!ATTLIST nmsploc
 id
   ID
   #REQUIRED
 locsrc
   CDATA
   #IMPLIED
 namespc
   (elements |
    entities)
   entities
 loctype
   CDATA
   #FIXED "locsrc ENTLOC"
 HyTime
   NAME
   #FIXED "nmsploc"
>
```

BIOGRAPHY

Eliot Kimber has been doing generic markup in one form or another for going on twenty years. His first exposure to SGML was IBM's dear departed Dialog Tag Language, for which he wrote a crude SGML parser in REXX by reverse engineering document instances. It was only later he discovered that not only was there a formal standard for this pointy stuff but IBM supplied software for processing it. Since that humble beginning, Eliot has helped define the state of the generalized markup art, first as an architect of IBM's Information Development Document Type (IBMIDDoc), as a co-editor of the HyTime standard (with Charles Goldfarb, Steve Newcomb, and Peter Newcomb), and as a founding member of the XML Working Group. Since 1994 Eliot has worked as a systems integration consultant focusing on the application of SGML, XML, HyTime, and related standards to various industrial information management problems. Eliot is also involved in the STEP and SGML harmonization effort, an attempt to define a robust and useful bridge between the worlds of industrial product data and SGML/XML-based information. On those rare occasions when he is not trying to teach brain-dead software to understand impossibly abstract standards, he is a devoted husband and dog owner. In his spare time, Eliot enjoys body boarding, hiking, and exploring the joys of open-source software.

Steve Newcomb has been a lifelong champion of the rights of data owners. Before embarking on a quixotic but ultimately successful attempt to actually implement the HyTime standard, Steve was a professor of music education at Florida State University, where he got involved with an attempt to define an SGML language for the representation of music, which became the Standard Music Description Language, ISO/IEC 10743, which led to the development of the HyTime standard. Steve has long been involved in standards development, first as an editor of the SMDL and HyTime standards, as a participant in the various MID and IETM efforts over the years, and then as a champion for and co-editor of the Topic Map standard (ISO/IEC 13250:1999). Steve is the President and founder of TechnoTeacher, Inc., suppliers of the GroveMinder™ engine, an industrial-strength grove-based information management system. When not struggling to help the great unwashed understand the power of the abstractions at his command, Steve is a devoted husband and father. In his spare time, Steve enjoys playing the piano, Star Trek™ reruns, and swimming.

Peter Newcomb is the youngest member of the HyTime editorial team and the principal architect of much of the machinery defined in the HyTime standard. He is principal architect and implementor of the GroveMinder™ system and one of the few people on the planet who actually understands the SP source code. While less visible than his father within the standards community, Pete has had a tremendous and profound influence on the shape and details of a number of standards. When not trying to make groves and HyTime safe for normal humans, Pete is a devoted husband and reef tank enthusiast. In his spare time, Pete enjoys in-line skating, wind surfing, and Descent.