# Efficient Self-Versioning Documents*

Tim A. Wagner
University of California
Berkeley

Susan L. Graham
University of California
Berkeley

## Abstract

*We describe methods to produce software and multimedia documents that are* self-versioning—*they efficiently capture changes as the document is modified, providing access to every version with extremely fine granularity. The approach uses an object-based spatial indexing scheme that combines fast access with very low storage overhead. Multiple tools can extract change reports from these documents without requiring their queries to be synchronized. We describe and evaluate a working implementation of these ideas, suitable for use in software development environments, multimedia authoring systems, and non-traditional databases.*

## 1. Introduction

Documents based on linked, hierarchical data structures with media content in their leaves are at the core of software development environments, multimedia authoring systems, and various types of non-traditional ('engineering') database implementations [5]. In each of these applications, interest in history services—particularly support for versioning—is tremendous and growing. We build on theoretical results developed for persistent linked data structures to create *self-versioning documents*. These documents cache their current contents, just as conventional representations do, but also provide access to previous versions by transparently incorporating modifications as they occur. The approach is designed to support high-bandwidth, fine-grained recording: individual textual and structural modifications as well as complex transformations (e.g., incremental compilation) can be treated as atomic updates.

We describe an implementation of lightweight versioned objects and a document representation that utilizes these objects to provide fine-grained versioning for main memory history logs. Our design is object-based: the primary index for updates is spatial, with each object encapsulating its own history by recording the association between modification times and values. This is in contrast to the usual design of editor 'undo' logs, where the primary indexing method is temporal (and often limited to a single entry). This work augments techniques for object caching and off-line storage developed for object-oriented databases [1], multi-user locking, and nested transaction support [6].

Recording modifications at this level of granularity requires attention to bandwidth constraints and representation issues: a typical document has many nodes, each containing several versioned fields. Existing methods for capturing updates are too slow and heavyweight to support fine-grained capture and lack crucial multimedia support. Our approach is based on efficient methods for producing persistent linked data structures: current values are provided in $O(1)$ time and arbitrary values in $O(\lg |\text{local modifications}|)$. A prototype environment supporting both incremental software development tools and multimedia authoring capabilities was used to evaluation our approach. Measured overhead is minimal: less than 4% of the running time is attributable to versioning. Recording a change to most versioned datatypes requires only 34 additional bits of storage. Caching policies and inlined query methods enable the current document content to be accessed as quickly as in unversioned documents. Lazy history log instantiation optimizes storage for unmodified objects.

The interface provided by the versioning scheme is both simple and largely media-independent.[1] Objects transparently record modifications made to them, stamping changes with the current 'global' version number. Each object is fully persistent (can provide its value for any given time).

*This research has been sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under Grant MDA972-92-J-1028. The content of this paper does not necessarily reflect the position or policy of the U. S. Government.
Authors' addresses: Tim A. Wagner, 573 Soda Hall and Susan L. Graham, 771 Soda Hall; Department of EECS, Computer Science Division, University of California, Berkeley, CA 94720-1776.
email: `twagner@cs.berkeley.edu`, `graham@cs.berkeley.edu`
URL: `http://http.cs.berkeley.edu/~twagner`, `http://http.cs.berkeley.edu/~graham`

[1] We do not describe multimedia encoding methods; formats such as MPEG are well-known, and Section 5 describes data structures for versioning datatypes such as links, booleans, and large text buffers.

Access to the current value is optimized. Changes to versioned objects during some time interval can also be determined efficiently. Self-versioning documents are built simply by using versioned objects for the link fields in document nodes. Data fields can be versioned as well; the nodes themselves retain their identity to simplify reference maintenance. Any version of the document can serve as the basis for a new version.

In addition to a low-level versioning approach that enables us to record modifications and answer queries about the history of a document component, we provide a flexible *change reporting* framework that allows clients to efficiently discover the regions of a document modified during some time period. The interface supports unrestricted queries, since different clients will have different needs. (For example, the presentation services will need to update the on-screen image of a program after every keystroke, but the user may wish to make several changes before incrementally recompiling the executable image.) Only a single versioned boolean field per internal document node is required to enable change reporting.

Our primary applications are highly interactive, requiring incremental algorithms and efficient access paths to all regions of a document. We support these needs by exploiting the document's natural structure. Additional structure is imposed by representing sequences of items as balanced binary trees. This representation ensures that any node in the document can be reached in logarithmic time. It also enables virtually any document type, including free-form text and graphic object lists, to be uniformly stored and accessed efficiently independent of the type, location, frequency, and nature of the modifications applied to it.

The remainder of this paper is organized as follows. In Section 2 we describe our document model and the client interface to versioned objects in greater detail. In Section 3 we examine two concrete implementations of self-versioning documents: programs and essays. Section 4 describes the run-time representation of the version hierarchy and its role in grouping updates. The implementation of versioned objects is described in Section 5. Further discussion of these ideas and related incremental algorithms may be found in the first author's dissertation.

## 2. Document Representation and Services

Although versioned objects can be instantiated individually, they most commonly occur as slots within document nodes. When a linked data structure is constructed using versioned slots, the data structure as a whole will be versioned. (Additional versioned data can also be stored in document fields.) In our prototype implementation, nodes are instances of C++ classes containing both versioned and unversioned fields; (permanent) sentinel node(s) provide clients with access to the root(s) of a linked data structure.

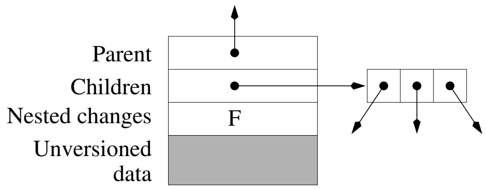


**Figure 1.** Document node representation. Each node typically contains both versioned and unversioned fields. The latter include read-only data and any value whose lifetime is within a single update or is global (e.g., cumulative edit counts).

For the applications of interest to us, documents are primarily represented as trees. Interior nodes provide structure while terminal nodes typically contain media-specific content (text, graphics, etc.). (Figure 1 illustrates an interior document node.) Information associated with a document node can be represented as slot values, annotations, or entries in a separate database. Such values may be links to other nodes, enabling general graphs to be modeled. The relationship between structure, content, annotations, and database entries may be maintained in whole or in part by automatic mechanisms. In the case of a program, for example, incremental analysis and transformation mechanisms preserve the relationship between the abstract syntactic structure, the textual content, and the binary representation of the compiled program. Although multilingual/multimedia documents are supported both by the model and by our prototype, in this paper we will assume a single language for the entire document (the structure can thus be described by a context-free grammar) and primarily text-based content. We will also assume a single current (cached) version for each document.

To provide high-performance and interactive response times, algorithms that query and update the document must be *incremental*, requiring efficient access to any component of a document at any time. To achieve efficiency, *sequences* of document components, such as paragraphs in an essay or statements in a program, have additional structure imposed on them in the form of a binary tree. These sequence trees are incrementally balanced after every modification and, together with the document's own hierarchical structure, enable logarithmic access times for every node in a document.

In a typical scenario, a document is modified repeatedly using one or more tools. Modifications are grouped by the client into atomic updates using begin/end edit methods of the global version tree (Section 4). Our approach makes no assumption regarding the frequency of updates; in fact, the finest level of granularity consistent with user expectations and interactive performance should be provided. Both trans-

```
<T> get ()
void set (<T> value)
bool changed ()
bool changed (FromVersion, ToVersion)
void alter_version (TargetVersion)
bool had_value (<T> value, FromVersion, ToVersion)
```

**Table 1.** Interface to versioned objects.

formation and analysis tools will routinely query the document to ascertain which regions have been modified and in what way. (This may lead to additional updates in the same or subsequent versions, or may be responsible for updating independent data structures such as an on-screen representation or an incrementally compiled object file.) When combined with high-performance object-oriented database technology, the techniques described here enable a seamless integration of source code control, editor undo logging, and filesystem caching for software documents [4], and a similar degree of support for natural language documents (for which commercial application programs rarely provide useful history-related services).

The document's own structure (along with any imposed tree structure for sequences) is used as an 'implicit' spatial indexing scheme for many version-related operations. This includes answering client change queries (see below) and altering the version of the entire document, for which an internal form of change reporting is required. In addition to link fields, other data fields of document nodes may be versioned, thus synchronizing their values with the version of the document as a whole.

Analysis and transformation tools discover document changes by performing a tree traversal that is restricted to only those areas that have been modified (along with additional regions dependent upon the changed material according to tool-specific semantics). Note that different tools need not be synchronized with one another. An individual node is generally considered changed whenever any of its versioned slots have been modified since the tool's previous interrogation. To enable efficient traversal of only the updated regions of a document, tools must also know whether there are additional changes within the subtree rooted at a given interior node. This *nested change* information is summarized by a versioned boolean slot in each interior document node that indicates whether the subtree the node roots (excluding itself) was changed. These bits provide a 'trail' to all the local changes for every version of the document. Nested change bits represent document *transitions* rather than document values *per se*, and thus require slightly different query methods. Nested change bits are ignored during local change queries and are cleared prior to each document update. As with other versioned objects, a value that duplicates the parent version's value need not be recorded; storage for change bits is also naturally minimized by locality in editing. Multiple edit categories can be supported by using several nested change bits per node.

**One of the following pairs for each versioned field**

```
<T> getX ()
setX (<T> value)
```

**The following three refer to the node as a whole**

```
bool changed ()
bool changed (FromVersion, ToVersion)
void alter_version (TargetVersion)
```

**Subtree update queries (excludes node itself)**

```
bool nested_changes ()
bool nested_changes (FromVersion, ToVersion)
```

**Table 2.** History-related interface to document nodes.

Table 1 contains the basic interface to a versioned object; this API will be described in greater detail in subsequent sections. The portion of the document node interface relevant to history-based queries and versioning is shown in Table 2. (Deletion and other miscellaneous state management functions are omitted.)

## 3. Applications

We consider two case studies from our prototype environment to illustrate self-versioning documents: programs and simple natural language documents ('essays') composed of text and graphics.

### 3.1. Programs

For programs, a formal language specification is compiled into a set of run-time libraries that are dynamically loaded to provide incremental analysis and translation services. Each program module is represented as a document whose structure corresponds to its abstract syntax. Terminal nodes are tokens that collectively represent the program text. (Stream-style comments and whitespace are integrated into the structure of the program [8].) Associative sequences (such as declaration and statement lists) are identified in the grammar using regular expression sequence operators and are instantiated as binary trees. (As with all documents, such trees are incrementally rebalanced after every atomic modification.)

Programs represent a stress case for fine-grained versioning because the structure itself is both large and dense (a high ratio of nodes to textual content). Incremental analysis and transformation tools that operate on the program (including such 'tools' as editing and presentation services) require precise change reporting in order to avoid time-consuming recomputation in their own analysis and to avoid triggering unnecessary work during subsequent analyses [7].

Program-based tools such as incremental parsing and semantic analysis take advantage of the ability of versioned objects to determine whether they have been modified during some time interval. In our prototype, consistency of the program representation is restored on user demand, integrating any textual and/or structural modifications performed since the previous analysis. Versions of the program created by analysis tools (as opposed to user edits) are *synchronized*—their content, structure, and database entries have been restored to consistency. The incremental lexer and parser use the synchronized version created by their most recent application as a reference point; any changes between that version and the current version are used to determine the regions requiring re-analysis.

In our prototype, semantic attributes are cached in transient (unversioned) slots. The user may request that semantic consistency be established for any point in the version hierarchy, resulting in a semantic reference version that is potentially different than that for syntactic analysis. In restoring semantic consistency, any changes since the reference version, including changes by the incremental lexer and parser, collectively determine the set of outdated attributes. Both syntactic and semantic analyzers use the query methods provided by the document node interface (Table 2) and have no knowledge of the internal details of document nodes or versioned objects.

### 3.2. Essays

In this case study, a relatively 'flat' natural language essay represents a typical word processor document. Once again a formal grammar (albeit a simple one) describes the permissible structure of such documents to the environment. As with programs, balanced binary trees are used to represent long sequences, guaranteeing incremental tool performance even though this document type lacks any intrinsic hierarchical structure. Each keystroke or collection of related keystrokes can be an atomic update.

Essays differ from programs by containing multiple media and text strings of non-trivial length: In a program, tokens average only a few characters in length [9]. In an essay, however, each 'token' represents an entire paragraph. The string storage method must therefore record substring modifications to avoid wasting space and to support fine-grained change reporting. Such *differential* storage differs from the state-based 'snapshot' recording used for small datatypes such as links and integers. Note that two-dimensional graphics can be handled by the techniques already described, since the attributes of graphic objects (location, radius, angle, etc.) can be represented as versioned real numbers. Sequences of graphic items (e.g., display lists) are treated like any other sequence.

## 4. The Global Version Tree

The relationship between the versions of a document is described by the *global version tree* (GVT). Each atomic update creates a new version, derived from its parent in this hierarchy.[2] The GVT is materialized in our approach, both to provide a version naming service to document clients and as an intrinsic part of the implementation of versioned objects. Modifications are grouped into updates by explicit begin/end edit methods provided by the GVT. Between updates, any version may be selected as the current version. (To simplify the exposition here, we assume a single GVT, a single document, and at most one version in progress at any time.)

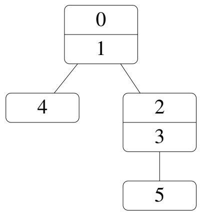


**Figure 2.** Global version tree. The version hierarchy is defined by the ancestor relationships between the nodes. Versions are named with integers indicating the order of their creation. Each node contains a linear 'run' of contiguous versions.

Within the GVT chains—linear 'runs' of successive versions—are compressed into single nodes. (These should not be confused with document nodes.) Figure 2 shows a sample GVT with six versions. Each version is named by a unique integer, which indexes an array to map an integer name to its GVT node. Collapsing chains minimizes the size of the GVT and speeds the calculation of non-current values in versioned objects. New GVT nodes are added as the *leftmost* sibling of their parent node (see Section 5). The GVT is also maintained as both pre- and post-order lists to support 'ancestor-of' queries.[3] Each versioned object's private history is a subset of the global version tree; an efficient array-based representation for these local version 'trees' is described in Section 5.

## 5. Implementing Versioned Objects

In this section we sketch the data structures and algorithms required to implement the versioned object interface described in Section 2, while meeting the time and space requirements for interactive applications. The theoretical basis for the design is a corrected version of the 'fat node' approach to persistent linked data structures developed by Driscoll, et al. [3].

[2] We do not consider the merging of different versions here; for coarse-grained merging the individual objects rarely conflict, and in cases where they do the resolution is media-specific.

[3] The pre-order sort of the GVT corresponds to the 'version list' in Driscoll, et al. [3].

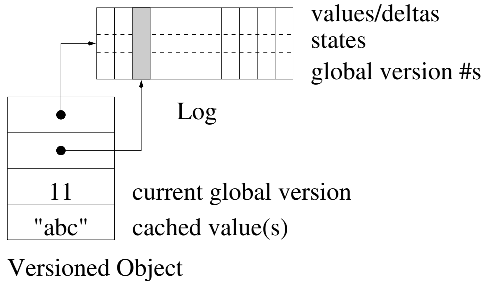


**Figure 3.** Representation of a versioned object. Usually the `current global version` field can be omitted, and the `cached value` field(s) are required only for differential objects.

**Map a global version to a local version (log index).**

```
int VObject::project (int version) const {
  int l = 0, r = RightmostLogIndex;
  const GvtNode *gd1, *gd2;
  gd1 = gvt->VersionToGvtNode(version);
  do {
    int x = (l + r) / 2, result;
    gd2 = gvt->VersionToGvtNode(versions[x]);
    // Use integer comparisons within a GvtNode
    if (gd1 == gd2) result = version - versions[x];
    else result = gvt->PreorderCompare(gd1, gd2);
    if (result < 0) r = x - 1;
    else if (result > 0) l = x + 1;
    else return x;
  } while (l <= r);
  return r;
}
```

**Figure 4.** Projection algorithm.

### 5.1. Full State Storage

Many of the elements of a versioned document, such as the links between the nodes themselves, are small datatypes that are treated as intrinsic values. These datatypes are versioned by recording the value itself at each checkpoint; the historical representation is conceptually an array of $<$*version*, *value*$>$ pairs. The versioned object itself is implemented as a pointer to this log, coupled with one or more log indices that denote the current (cached) values of the object. In addition to the global version identifiers and the values themselves, the log also contains state information, allowing deleted and undefined object states to be supported. Figure 3 illustrates the log data structure; the 'cached value' field is not present in objects that record their full state in the individual log entries.

The version log can be implemented in a number of ways, such as balanced binary trees. Our prototype uses dynamically-sized arrays to minimize space overhead. For additional time efficiency, the log can be maintained as a pair of arrays stored in contiguous memory with an 'edit gap' between them. This arrangement is consistent with typical application requirements: documents are frequently changed at multiple points, but it is less common for the user to back up and begin modifications from a previous version. To alter the current version of a full state object, the index slot of the object is simply changed to point to a different value in the log. (Section 5.3 describes the mapping of global versions to local log indices in further detail.)

Recall from Section 4 that the structure of the global version tree is partially defined by the order of version construction, since each version must be a child of its parent. The Driscoll algorithm produces a complete ordering by arranging sibling GVT nodes left-to-right by *decreasing* version numbers. The log entries are sorted by their global version fields, ordered according to the pre-order linearization of the nodes in the the global version tree. When a new value is recorded in the object, an *additional* entry is needed if the version following the current global version in the the pre-order sort is not already represented in the local log.

### 5.2. Differential Storage

As mentioned in Section 3.2, some datatypes, such as lengthy text strings, audio and video recordings, etc., are too large to store in full at each checkpoint. Even if this were feasible, sufficiently precise change reporting would typically demand a differential strategy. Although we do not describe media-specific differential representation formats here, the framework for storing and accessing such objects is independent of their representation.

As with full-state storage, the fundamental storage model is a log, but it stores *deltas* rather than complete values. Each entry describes how to build the local value for that version by applying the stored delta to the object's value in the parent version. (Deltas must be reversible—they are also used to build the parent version's value starting from a child version.) Since log entries are updates, not values, the current value is cached in the versioned object itself (the 'cached value' slot shown in Figure 3). To retrieve the object's value at a different point in time, a temporary copy of the object is made and then transitioned to the target time by applying deltas from the log.

### 5.3 Projection

The most frequently requested value for a versioned object is its current (cached) value. To produce the value corresponding to a different time requires mapping a name for the target global version (which is simply its ordinal number) into a name for the local version, which represents a slot index in the object's history log. This mapping thus *projects* the global version onto the object's log. The projection is the rightmost index such that the global version recorded for that entry is not to the right of the target version in the pre-order linearization of the GVT. The algorithm in Figure 4 computes this result efficiently using a logarithmic search through the array.

The compact form of GVT nodes, which collapse linear runs in the GVT, allows the projection algorithm to use integer comparisons to implement its comparison function in the common case, making the lookup extremely fast. When two global versions belong to different GVT nodes, the pre-order comparison can be made efficient by representing the GVT's pre-order linearization as a data structure that supports $O(1)$ order queries [2].

Datatypes that store their entire contents at each checkpoint can simply return the corresponding entry of their value array using the result of the projection. Objects stored differentially require additional computation, since multiple deltas must be applied to the currently cached value to produce the target value. Repeated calls to `project` are made as the value computation traverses the path through the local version tree corresponding to the path in the GVT between the current and target versions.

### 5.4. Lazy Log Construction

In a typical application, the contents of the document will be loaded into main memory from a disk or network image and subsequently modified by the user. For all but the smallest documents, however, many versioned objects will remain unmodified—their initial value will be maintained until the document is destroyed. This suggests that a log representation, while appropriate for the general case, adds unnecessary space and time overhead for the majority of versioned values.

To address this, we instantiate all versioned objects as single-valued *temporary* containers. These are one-entry logs, but without the arrays or administrative overhead. If a temporary object is subsequently modified, it is first transformed into a log containing the sole value; the temporary container is then destroyed and the modification continues as if a log had been in use all along. The use of a temporary representation for a versioned object can be detected by setting its index field to a negative value.

### 5.5. Relating Versioned Objects to a Global Version Tree

When the user is editing several documents with distinct histories, each local object conceptually possesses a pointer to a corresponding *global object*, which identifies both the global version tree and a specific version within it as the current version. (The current version is needed to process queries that use it as an implicit argument). However, this arrangement usually results in a great many copies of identical pointers. When the design requires that multiple versioned objects maintain a synchronized current version, the pointer to the corresponding global object can be shared. The document node level is a logical choice for this sharing, since its versioned fields always represent the current version of the node. (In many cases the entire document is constrained to be in a single, consistent version, in which case only one copy of the global object pointer is needed.)

## 6. Conclusion

We have described self-versioning documents that support fine-grained recording of their modification history using an object-based mechanism. The approach is useful for representing programs in software development environments, online hypermedia documents, and time-varying data within a database. The techniques are based on a proven and efficient theoretical model, augmented with special support to make the common cases fast and to minimize overall space consumption. Highly efficient change reporting is provided by versioning a nested change summary bit in document nodes. The combination of intrinsic document structure and a balanced binary tree representation of sequences enables clients of this representation to achieve incremental performance.

## References


[1] R. Bretl et al. The GemStone data management system. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 283–308, 1989.

[2] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symp. on Theory of Computing*, pages 365–372, 1987.

[3] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, Feb. 1989.

[4] C. W. Fraser and E. W. Myers. An editor for revision control. *Software—Practice & Experience*, 9(2):277–295, 1987.

[5] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):275–408, 1990.

[6] B. Magnusson, U. Asklund, and S. Minör. Fine-grained revision control for collaborative software development. In *First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 33–41, New York, 1993. ACM Press.

[7] T. A. Wagner and S. L. Graham. Integrating incremental analysis with version management. In *5th European Softw. Eng. Conf.*, number 989 in LNCS, pages 205–218, Berlin, Sep. 1995. Springer-Verlag.

[8] T. A. Wagner and S. L. Graham. Modeling explicit whitespace in an incremental SDE, 1996. In preparation.

[9] W. M. Waite. The cost of lexical analysis. *Software—Practice & Experience*, 16(5):473–488, May 1986.