Implementation Techniques for Integral Version Management

Ernst Lippe
Software Engineering Research Centre, University of Utrecht
Gert Florijn
Software Engineering Research Centre
P.O. Box 424
3500 AK Utrecht
The Netherlands
e-mail: lippe@serc.nl, florijn@serc.nl

Abstract

Version management services have traditionally focussed on versioning individual objects, and especially text files. This approach ignores the fact that (versions of) different objects are not independent from each other, and introduces the problem of finding consistent version combinations. One way to alleviate these problems is by expanding the unit of versioning, i.e. by applying integral version management to collections of objects.

This paper describes implementation techniques for integral version control. The techniques are applied to an object model which is characteristic for modern (engineering) object management systems, i.e. a model in which data is represented through objects and relationships. The techniques we propose support for both linear development and general, branching history. Furthermore, the techniques are incremental: they only store the difference with respect to the previous version.

1 Introduction

Object management systems (OMSes) have an increasing popularity, especially in engineering environments [PCT, BCG*87, BMO*89, F*88, KL89]. These object management systems provide mechanisms for storage of typed objects, plus mechanisms to handle references between objects. It is expected that they will replace traditional file systems for many applications, especially as a basis for engineering support environments.

Object management systems give new possibilities and new problems for version control. Traditional version control systems apply version control to individual files. The equivalent in object management systems would be that version control is applied to individual objects. We argue that version control should not be limited to individual objects but instead be applied to larger units (see section 2). We refer to this approach as *integral version management*. Integral version management forms the basis of the CAMERA system [LFB89, LF91], a version control system aimed at supporting (software) developers that use loosely coupled distributed networks. In CAMERA integral version management is applied to the entire contents of an OMS.

Efficient implementation of versioning is hard. Therefore, the main part of this paper is concerned with possible implementations of integral version management when applied to a data model consisting of objects and relationships. We will discuss data structures for storing and retrieving versions that can be used for linear (section 5) and non-linear (section 6) version history.

2 Version management

Version management is concerned with handling versions of objects in a systematic way. The main applications of version management are the recording of the history of developments, and support and coordination of parallel activities[1].

Versions (or revisions) are used to record the evolution of data. In software development for instance it is important to trace the modifications that were made to a set of programs. Modifications often introduce new bugs; and in such a case it is very important to know exactly what was changed. Another application for historical versions is release management. All entities that are needed to recreate a specific release of a software product are stored on a safe medium, so that this release can be recreated (and possibly corrected) at a later point in time. Historical version management can also be used to "undo" unwanted modifications, if versions are created sufficiently often. When a user has made a mistake, (e.g. deleted an important object) an old version can be used to continue work.

Version management can be used to support parallel activity, where a group of users is working on (a version of) a set of objects at the same time. In such situations users often want to have their private workspace, which they can modify without interference from others. If we look at a multi-user hypertext system, for instance, users may want to add new links without seeing the links that are added by others. However if each user has a private version of all data, it is very hard to discover when new versions of objects are created, and what the contribution of a single line of development is to the overall result. Version management systems can help in managing and tracing these parallel developments.

Units of version management

Traditionally, version control is applied to individual objects. In this section we provide some arguments for the claim that this is not sufficient (see also [LFB89]). Instead version control should be applied to collections of objects and their mutual references.

Versions of one object depend on specific versions of other objects, since a new version is developed using specific versions for the other objects. In general it is not possible to combine arbitrary versions of some objects, since these versions need not be mutually compatible. For example, in an object oriented environment, a version of a method in one class may depend on the existence of a method in second class. Inconsistency problems arise if a version of the second class has been selected that does not contain this method.

Ideally, a version management system should only allow "consistent" version combinations. This is a difficult problem since the number of possible combinations grows exponentially. Furthermore, it is in general impossible to determine automatically which combinations are consistent, because the definition of consistency depends on the semantics of the objects that are involved.

However, there seems to be a reasonable choice for obtaining consistent combinations. It is likely that the set of versions that were used to develop a new version of some object will form a consistent selection. This can be supported by recording which versions of related objects were used in the creation of a particular version of an object. One particular way to do so is by shifting the unit of versioning up to collections of objects, and the references between them.

This approach has an intuitive appeal. Often changes that are seen as a conceptual unit by a user will involve modifications to a group of related objects. For example, adding a new feature to a program may involve changes to several program modules. The changes to all these modules are of course closely related and can be seen as one compound modification to the group as a unit. Again, it seems more appropriate to apply version management to the group as a whole (see [NSE88, BGW89]).

[1] Traditionally variant management is also seen as a form of version management. Variants are different objects that share functionality. This notion is mainly used in program development were different variants of a program (e.g. for different machine architectures) may share a large portion of their source code. In our opinion variant management is orthogonal to history management, and should be solved by the modeling capabilities of the OMS.

Versioning larger units also provides a direct solution to the problems of versioning references between objects. Object systems provide mechanisms to store and manipulate references between objects. References can occur in two forms: as instance variables that act as pointers to other objects (most object-oriented languages), and in the form of relationships (some object-oriented databases and engineering object management systems). Since references constitute important information, they must be subject to version control, too. A problem here is, how version control on references is related to version control on the individual objects. For example, what must happen if a new version is created of an object that is referenced by another object. Must a new version of the referring object be created as well? This problem can be solved by putting both objects (and the reference) in one group and creating a new version of this group. Thus we version graphs of objects.

Certain object oriented systems use the notion of *composite* or *complex* object to define the sub-graphs that are subject to version management (e.g. [KBC*87, PAC89]). A disadvantage of this approach is that it makes it impossible to apply version control to references *between* objects that are not part of the same composite object. This places a rather severe restriction on the use of references. This is especially important since it is not always easy to define a particular composition structure, in advance. In hypertext systems, for instance, one often finds overlapping views, each of which could be a complex object. In some cases composition structures are even determined by the operations that are performed [Rum88]. Furthermore, the same combination problems that we saw for individual objects reoccur, but now on a different level. I.e. combining versions of several complex objects may give rise to consistency problems.

Consequently, a reasonable solution appears to be *integral version control*: version control is simultaneously applied to a self-contained collection of objects and the references among them.

Systems for integral version management

There are several systems that provide some form of integral version management. Systems like NSE [NSE88], and Plan 9 [PPTT90] store versions of (a part of) a file system, see also [Hum89]. Other examples of systems that provide integral version management on a graph of objects are Gandalf [BGW89], and [LCM*89]. Postgres [SR87] also offers a form of integral version management in a relational database. By tagging each tuple with the time at which it was created, the system supports linear history recording for a complete database.

In a certain sense it can be said that Smalltalk gives a form of integral version management in the Smalltalk images. But because images are large, it is highly impractical to store many of them. PIE [GB86] attempts to solve some of these problems by adding a notion of layers, a (sub)-graph of related objects that is treated as a unit, and that can be exchanged between users. A similar notion of layers can be found in [GMS89] for a software development system that is built on top of a traditional file system, Sun's Translucent File System [Hen88] where the notion of layer is incorporated into the file system itself, and [Pre90] for hypertext systems.

The CAMERA system, currently under development at SERC [LF91], also incorporates integral version management. CAMERA is aimed at supporting teams of (software) developers who cooperate across loosely-coupled networks. CAMERA has a two-level architecture. Integral version control is applied to object worlds containing the data (e.g. the design, the software) that is under development. Versions of these object worlds (called snapshots) are stored in a higher level object management system (called the Album), which supports and records the development process.

In the remainder of this paper we focus on implementation techniques for integral version management that were developed for the CAMERA system. This emphasis means that we will not discuss usage aspects of versioning service, such as whether version creation is implicit or explicit (via checkout/checkin), or policies that determine when new versions should be created. The reader is referred to [LF91] for further background on this.

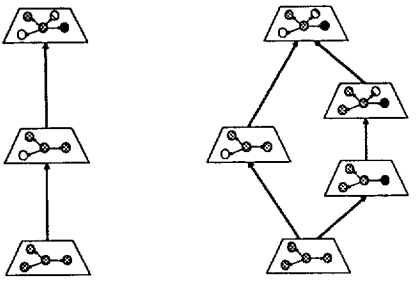

Figure 1: Examples of linear and branching history

3 A model for integral version management

In this section we will give a model for integral version management which is used in the description of the implementation techniques.

The example object model

This section describes the sample object model that will be used in the rest of this paper. We assume that data is stored in an object management system that is based on a hybrid data model, i.e. a model providing both objects and relations.

Objects are identified by an *object identifier* OID. The object identifier of an object remains the same when the object is changed. Every object has a *value*. The internal structure of the objects is not relevant for the rest of this paper, but we should point out that the internal structure cannot contain references to other objects.

References are handled via *relations*. Relations are sets of *relationships*, n-tuples of object identifiers or primitive values. Each relation is identified by a *relation identifier* RELID. We can use relations to describe how version control on inter-object references can be implemented. Relations are also treated separately because references between objects could change faster or slower than the contents of the object. This makes it more efficient to separate the version control of references from the version control of the contents.

Snapshots

Under integral version management versions of the entire contents of an object management system are recorded. Versions of the OMS are called *snapshots*. Since snapshots are used to record history we will assume that old snapshots (i.e. snapshots for which a successor has been created) are immutable. If errors are discovered in an old snapshot, a new snapshot with the modifications must be created. Possibly, the erroneous snapshot could be deleted.

Every snapshot is uniquely identified by a *time stamp*, also known as version identifier. The time stamps are unique labels, that need not have any relation to actual clock times. Time stamps have an ordering imposed on them, that describes the successor relation between the versions.

Two different types of history can be distinguished: *linear* and *non-linear* version history. In linear version history every version (except for the most recent one) has a single successor. This happens if there is one central version that is developed. The ordering that corresponds with linear history is a total ordering of the time stamps.

In a non-linear version history versions can have more than one successor, and more than one predecessor. This version history is characterized by branches and merges. This occurs during parallel development,

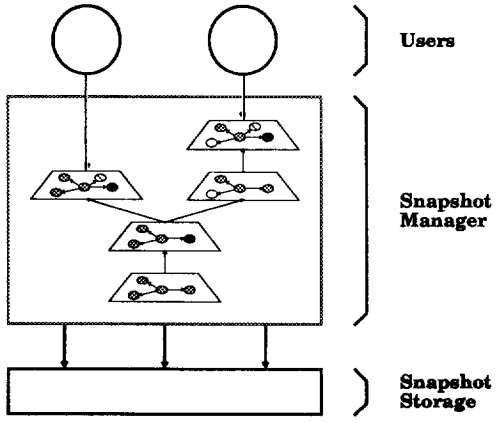

Figure 2: Architecture for integral version management

when several persons use the same version as basis for their work. The ordering in a non-linear version history is a partial ordering of the time stamps, i.e. the successor relation forms a directed acyclic graph.

As we have seen above inside the OMS every object has a unique object identifier (OID) and likewise every relation is identified by a relation identifier (RELID). Thus a version of an object (relation) can be identified by the combination of its object (relation) identifier and the time stamp of the snapshot to which this version belongs.

The two most important operations on snapshots, are creation and retrieval. Users can retrieve any existing snapshot, and use that as the contents of their workspace. At certain points in time a new snapshot is created of the entire contents of this workspace. Users can also examine parts of existing snapshots, e.g. to determine the values of objects and relations inside the snapshot.

4 The implementation model

An overview of the model for integral version management is given in figure 2. Versions of the entire contents of an OMS are stored by the system, and one such version is called a *snapshot*. Users operate on specific snapshots, each of which contains a particular state of the OMS. The snapshot manager handles operations on these snapshots, and maintains the successor relation between snapshots.

All snapshots are stored in one global snapshot storage, and this paper describes data structures for the efficient implementation of this global store. Different techniques can be used for objects and relations, and the following sections will introduce the functionality of these data structures. This section will introduce a global framework that will be used to describe the specific data-structures for linear and non-linear history, in sections 5 and 6, respectively.

Storing versions of objects

At an abstract level the functionality of an OMS can be described as a data structure that associates an object identifier of an object with the value (the contents) of this object, i.e.:

$$OMS : OID \rightarrow value$$

If we add integral version management on objects, this can be described as:

$$Index : OID \times TimeStamp \rightarrow value$$

i.e. this data structure maps a combination of an object identifier and a time stamp to the value of this object in the snapshot.

A straightforward approach for implementing integral version management is to store a complete copy of the entire snapshot. This is effectively what is done in Smalltalk, where the equivalent of a snapshot is a Smalltalk image. It is obvious that for virtually all practical applications this will soon become prohibitively expensive, unless the number of snapshots remains very small.

A somewhat more advanced solution is based on the observation that successive snapshots will tend to be very similar, since most objects and relations will not change between snapshots. It is therefore wasteful to make separate copies of all objects — especially when objects are large — for all snapshots in which the object occurs, since the value of the object will be the same in most of these snapshots.

An obvious implementation technique is to let snapshots share common values of objects. This can be done by adding an extra level of indirection. Every *different* value of an object is stored in a separate data structure, and can be identified by a value identifier (VALID). The data structure now consist of two parts:

$$VersionIndex : OID \times TimeStamp \rightarrow VALID$$

and

$$ValueIndex : VALID \rightarrow value$$

The ValueIndex contains for each object all values of this object. In order to retrieve a specific version, first the VersionIndex is searched to find the corresponding VALID, which is then used as key in ValueIndex to retrieve the value of this version. In a functional notation this can be described as

$$Index(oid, timestamp) = ValueIndex(VersionIndex(oid, timestamp))$$

The implementation of the ValueIndex will be dependent on the type of the values. With text files, for example, compression techniques such as delta compression ([Tic85]) can be used. For objects that contain fixed length records the techniques from [KL84] might be used. A data structure that can be used to store different versions of a set in a compressed way is described in sections 5.2 (linear history) and 6.2 (non-linear history).

For objects that are small or that have very dissimilar values, these compression techniques cannot be used: in this case the different values will be stored completely. Very small values (e.g. integers) can even be stored in the VersionIndex itself, instead of their VALID, thereby avoiding the extra overhead of using the ValueIndex.

In the rest of this paper we will ignore the ValueIndex, and concentrate on implementations for VersionIndex. These will be described in sections 5.1 and 6.1.

Storing versions of relations

It is of course possible to treat a relation as an object that has a set (of relationships) as its value, and then to use the techniques from the previous section. Special algorithms would be needed to compress the values of relations. Since relations are sets, we could use the techniques like the ones described in sections 5.2 and 6.2. However, the speed of access for relations can be greatly improved by defining special index structures, which reflect the common use of the relation.

A standard method for the implementation of relations is as one or more indexes on search keys of the relation. Such an index has the following functionality:

$$RELID \times Key \rightarrow \mathsf{P}\ Relationship$$

The index takes a relation identifier plus a search key, and returns the set of relationships with this key. These indexes are commonly found in databases but can also be used for the navigational behaviour found in most OMS applications, by using object identifiers as search keys.

A modified form of these indexes can be used to store old versions of relations. We define a two level index:

$$Relindex : RELID \times Key \rightarrow RVALID$$

$$RelVersionIndex : RVALID \times TimeStamp \rightarrow \mathbf{P}\ Relationship$$

RelVersionIndex implements versions of sets (of relationships). Specialized data structures for storing versions of sets will be described sections 5.2 (linear history) and 6.2 (non-linear history).

Relindex can be implemented using traditional indexing techniques, e.g. B-trees [Knu73].

Because these indexes have a key, fast access to the tuples is possible.

5 Linear version history

An important special type of history is the linear version history in which each time stamp (except for the last one) has precisely one successor. The restricted nature of the linear version history makes it possible to use more efficient data structures than those for the general non-linear version history, which will be described in section 6.

This section, that describes data structures for linear version history, is divided in two parts. The first part (section 5.1) concentrates on implementations for VersionIndex, the data structure that is used for integral version management on objects, while section 5.2 describes a data structure for storing versions of sets that can be used for integral version management on relations.

5.1 Sparse index

An obvious implementation for VersionIndex is to use a B-tree [Knu73], with the combination of OID and TimeStamp as key. If the index is sorted lexicographically on ⟨OID, TimeStamp⟩, all entries with the same OID will be adjacent in the index, and ordered by increasing TimeStamp.

It can be expected that successive entries in the index will tend to have the same value, since typically an object will not change between two snapshots. In a certain sense these repeated entries are redundant, and can be omitted.

A more precise definition of when an entry can be omitted is the following. We define that an entry $\langle oid, t\rangle$ *immediately precedes* an entry $\langle oid, t'\rangle$ in the index if there is no entry $\langle oid, t''\rangle$ in the index with $t < t'' < t'$. An entry $\langle oid, t\rangle$ can be omitted from the index if there is an entry $\langle oid, t'\rangle$ that would immediately precede $\langle oid, t\rangle$ if it were present in the index, such that $indexval(\langle oid, t\rangle) = indexval(\langle oid, t'\rangle)$.

A *sparse index* is an index where the value of each entry is different from the immediately preceding entry. In order to handle deletions of objects correctly, the domain of object values is extended with the special object value null, i.e. an entry $(\langle oid, t\rangle \mapsto \mathsf{null})$ would indicate that at time t, object identifier oid was not referring to an existing object.

The insertion algorithm for a new entry $(\langle oid, t\rangle \mapsto val)$ is:

if the index contains an entry $\langle oid, t'\rangle$ that immediately precedes $\langle oid, t\rangle$ such that $indexval(\langle oid, t'\rangle) = val$
then do nothing
else insert $(\langle oid, t\rangle \mapsto val)$ in the index

Obviously, the deletion of an object *oid* at time t is indicated by inserting an entry $(\langle oid, t\rangle \mapsto \mathsf{null})$ in the index.

The algorithm to retrieve the value of an object *oid* at time t is:

```
if   there is an entry for ⟨oid, t⟩
then return indexval(⟨oid, t⟩)
else if   there is an immediately preceding entry ⟨oid, t'⟩
     then return indexval(⟨oid, t'⟩)
     else return null
```

This sparse index can be implemented using traditional data structures, e.g. the ubiquitous B-trees.

Performance

The index will only contain an entry for an object, if that object has been changed with respect to the previous snapshot. In general, it is to be expected that most objects will not change between snapshots. Therefore, a sparse index will be much smaller than the corresponding full index that stores all object values for all time stamps.

Searching the sparse index can be somewhat faster than with the full index due to the smaller size of the index. In a similar way the creation of a new snapshot can be faster because only objects that have been changed since the previous snapshot must be entered into the index.

5.2 Versions of sets

This section describes a data structure that can be used to handle versions of sets for a linear version history, that can be used to store versions of relations. This data structure is called *version list*. A version list consists of a list of 3-tuples $\langle el, birth, death\rangle$, where el is an element of the set, *birth* and *death* are both time stamps. The meaning of a 3-tuple is that el is a member of the set at all times t such that $birth \leq t < death$. All elements that are in the set at the most recent time stamp have a death time stamp of $+\infty$.

The 3-tuples in the version list are sorted on decreasing death time stamp. This means that all the entries that are currently alive are located near the head of the list. When a new element r is added to the set at time t, a new entry $\langle r, t, +\infty\rangle$ is added to the version list. For a deletion of a element el at time t the death time of the corresponding 3-tuple is changed to t. In order to keep the list sorted this 3-tuple may have to be moved to the rear of the list, this can be arranged by swapping this entry with the last 3-tuple that is currently alive.

The algorithm for reconstructing the set of elements for a given time t from a version list v is:

```
i := 1;
result := ∅
while i ≤ size(v) and v[i].death > t do
      if v[i].birth ≤ t
           then result := result ∪ v[i].el
      i:= i+1
```

Observe that we can terminate the search loop when an entry is found for which v[i].death < t because the list is sorted on decreasing death time. So we know that for all j > i, v[i].death < t.

Performance

Because old snapshots are immutable it is only possible to insert and delete elements for the most recent snapshot. Therefore, insertion and deletion operations will be $O(s)$ where s is the size of the current version of the set.

From the algorithm that retrieves old versions of the set, it can be seen that newer versions can be retrieved faster than older versions. This is desirable, because it can be expected that recent versions will be used much more frequently than older versions. The time needed to retrieve the most recent version is $O(s)$, where s is the size of the current version of the set, and this is of course the best achievable.

6 Non-linear version history

In the general case, version histories are not linear — i.e. timestamps can have more than one successor/predecessor — due to branching and merging. The techniques that were used in the previous section cannot be applied in this case. This section will describe some extensions that can be used in the non-linear case as well.

Like the previous section, this section is split into two parts. The first part describes techniques for storing objects. The second part describes data structures for storing versions of sets that can be used to implement versioned relations, as was explained in section 4.

6.1 Storing versions of objects

The sparse index technique, that was described in section 5.1, cannot be used immediately in the case of non-linear history. The most important problem is that in this case time stamps are not completely ordered. Therefore, they cannot be used as key in the sparse index. This problem can be solved by adding yet another indirection.

The procedure works as follows. There is one global mapping table: $\mathsf{map} : \mathit{TimeStamp} \rightarrow \mathbb{N}$, that is used to convert a TimeStamp to a (totally ordered) natural number. The resulting number is used to index the modified sparse index

$$\mathit{VersionIndex}' : \mathit{OID} \times \mathbb{N} \rightarrow \mathit{VALID}$$

This mapping table will contain one entry per snapshot. It will thus be relatively small, and could be kept in core. It can be implemented e.g. as a hash table.

Example

An example sparse index for the data in figure 3 is shown in figure 4. In this example, we have two objects OID_1 and OID_2, and a set of snapshots with timestamps $T_a \ldots T_f$. The initial snapshot is T_a, that only contains the object OID_1 with value A. In one line of development (T_b, T_d, T_e), the object OID_2 is created and modified. In the other development line (T_c) object OID_1 is modified. The final snapshot T_f merges the results of both lines of development.

Insertion algorithms

The size of the index VersionIndex' depends on the mapping table in the following way : for all TimeStamps A and B let $d(A, B)$ be the number of objects that are different between snapshot A and snapshot B. The TimeStamps in the index are sorted by the increasing value of the mapping function. If we arrange all TimeStamps in a sequence $t_1 \ldots t_n$ such that $\mathsf{map}(t_i) < \mathsf{map}(t_{i+1})$, the total size of the index is

$$l = \sum_{i=1}^{n-1} d(t_i, t_{i+1})$$

Snapshot contents

T	OID_1	OID_2
T_a	A	–
T_b	A	B
T_c	D	–
T_d	A	B
T_e	A	C
T_f	D	C

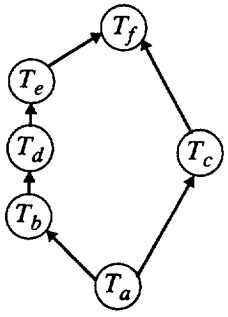

Figure 3: Example data for sparse index. This figure shows a partially ordered set of time stamps, and the contents of the corresponding snapshots.

map function

T	map(T)
T_a	1
T_b	2
T_c	6
T_d	3
T_e	4
T_f	5

VersionIndex'

OID	map(T)	Object-Version
OID_1	1	A
OID_1	5	D
OID_2	2	B
OID_2	4	C
OID_2	6	null

Figure 4: A sparse index

This table shows an example sparse index and the corresponding mapping function. null indicates a deleted entry.

A bad choice for the mapping table will give a very large index. Unfortunately it is not very easy to discover the optimal mapping table for a given set of snapshots, that gives a minimal value for l. Appendix A contains a proof that this problem is NP-complete. Because the problem of finding an optimal mapping table is NP-complete, for a practical insertion algorithm heuristic methods must be used. Which particular method is most suited depends very much on local branching patterns. The index will be relatively small if every snapshot is similar to its neighbors in the index. It can be expected that every snapshot is similar to its predecessor, thus a simple heuristic would be to insert every snapshot as a neighbor in the index of its predecessor.

It is not always possible to insert a successor of a snapshot with time stamp t as a neighbor, this happens if $\mathsf{map}(t)+1$ and $\mathsf{map}(t)-1$ are already used in the mapping table. In this case the new snapshot must be inserted elsewhere. A useful (greedy) heuristic, is to insert this snapshot at a place in the index where it causes the smallest increase in the total index size, i.e. between two snapshots to which it is very similar. Simulation experiments with this heuristic indicate that the size of the produced index is normally very close to the size with the optimal mapping table.

Performance

The performance of retrieval operations on this data-structure can be very similar to that of the standard sparse matrix for linear history. The only extra overhead is one lookup in the mapping table. When the relevant part of the mapping table is kept in core (as it probably will be) the additional overhead is negligible.

6.2 Implementations for versioned sets

This section describes implementation techniques that can be used to store versions of sets, in the case of non-linear version history. As was explained in section 4 this data structure can be used to store versions of relations.

Several standard data structures like lists and hash tables can be used to represent sets. But these data structures cannot be used to store multiple versions of the set in an efficient way. Two data structures, delta lists and modified AVL-trees, that are more space efficient will be described in the next sections.

Delta lists

A first data structure for the implementation of versioned sets makes use of *delta lists*. This data structure consists of two parts. Certain versions of the set are stored in full as *base versions*, using a traditional data structure (hash tables are suitable). Other versions are stored as sets of changes to a base version in the delta lists. A delta list is a list of *deltas*, every delta contains the differences (additions/deletions of elements) with respect to the previous version of the set. Every delta in a delta list is marked with a time stamp. The first delta in every delta list contains the modification with respect to a fully stored base version.

To retrieve a specific version of the set with a specific time stamp T, first the corresponding base version must be retrieved, and then all delta's that have time stamps between that of the base version and T must be applied. Thus the average access time will depend on the size of the delta lists. If base versions are made more often access time will get shorter, but storage demands will increase.

The average amount of storage per set with this data structure is:

$$c_1 \cdot f \cdot n + c_2 \cdot (1-f) \cdot k$$

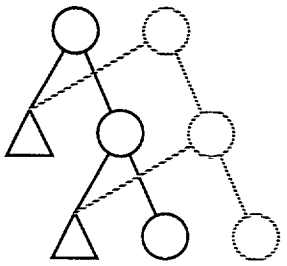

Figure 5: Shared subtree implementation for sets

The black tree and the shaded tree represent two different versions of the same set that share common subtrees.

where f = proportion of the versions that are stored as base versions
c_1 = storage size per element in a base version
n = average number of elements in a set
c_2 = storage size per element in a delta
k = average number of elements in a delta

The average access time is:

$$\tfrac{1}{2}d_1 \cdot (1-f) \cdot k + d_2$$

where d_1 = time to access one element in a delta
d_2 = time to access one element in a base version

The proportion of versions that are stored as base versions (f) is an important parameter when implementing this data structure. It represents a classical space/time tradeoff, a low value for f gives a compact data structure while a high value gives good access performance.

Trees

Another standard way to implement sets is by using trees, e.g. AVL trees (see e.g. [Knu73]). This data structure can be used in a slightly modified form to store different versions of sets in an efficient way, by *sharing* common subtrees. In large trees successive different versions of a set, that differ in only a few elements, will have large subtrees in common, and these can be shared between the different versions, see figure 5.

The procedure for inserting/deleting elements is almost the same as that for a standard AVL tree, but instead of modifying existing nodes in the tree, a new copy of the node is made that contains the new value. When a node is copied, all of its ancestors must be copied as well. Each copied ancestor will point to the copies of its children, and if a child is not modified, the copied parent will use the original child node, which will then be shared between the old and the new version of the tree. If, for example, two successive sets differ only in one element, they will only differ in the nodes that form the path from the root to that element. All other nodes can be shared between the two versions.

Every individual tree is a normal balanced AVL-tree, so average access-times are logarithmic in the size of the tree. Also the worst case access-times are logarithmic, since the maximum depth of an AVL-tree of size N is $1.4404 \log_2(N+2) - 0.328$ [Knu73].

For single deletions/insertions the storage requirement to store the new version is logarithmic in the total size of the set, since all elements of the path to the root must be copied. However when multiple additions/deletions are made the overhead per individual modification is less than log n.

Analysis of AVL-trees for versioned sets

An important question about the implementation of versioned sets with AVL-trees is the amount of storage that will be consumed by this data structure. This section attempts to give some approximate answers.

The storage requirements are determined by the overlap between successive versions. If they can share many common subtrees, they will require less storage. So the question is: how many nodes will on the average be shared between a tree and its successors?

The key observation for this analysis is that a node will only be shared with the next tree if the subtree that is rooted at this node, is left completely unchanged. Thus, the expected number of nodes that can be shared with the successor of tree T is:

$$\sigma(T) = \sum_{s \in T} p(s)$$

where $p(s)$ is the probability that subtree s will not be modified.

We now split the newly created nodes into two different groups: primary and secondary nodes; primary nodes are those that would also be changed in a normal AVL tree, while secondary nodes are those nodes which are not primary, but are created because one of their subtrees was modified. We assume that the possibility that a node does not undergo a primary change is the same for all nodes and equal to c, and that the probabilities for the different nodes are independent from one another. Under these assumptions the expected number of shared nodes becomes

$$\sigma(T) = \sum_{s \in T} c^{|s|}$$

where $|s|$ denotes the number of nodes in the tree s.

The value of $\sigma(T)$ depends on the shape of T. Therefore, we will analyze the two extreme shapes that an AVL tree can have: a fully balanced tree (best case) and the Fibonacci tree (worst case). These are the most balanced and most unbalanced shapes that an AVL-tree can have [Knu73].

Balanced trees

A fully balanced tree can be defined as a tree in which every node has either no children or two children that have the same height. In a fully balanced tree of height h there are 2^{h-n} subtrees of height n, each of which contains $2^{n+1} - 1$ nodes. Thus in this case:

$$\sigma(T) = \sum_{n=0}^{h} 2^{h-n} c^{2^{n+1}-1}$$

Fibonacci trees

Fibonacci trees are the most unbalanced AVL trees. The Fibonacci trees $\mathcal{F}_0, \mathcal{F}_1, \ldots$ can be defined in the following way:

$\mathcal{F}_0$ is the empty tree, $\mathcal{F}_1$ is the tree consisting of one node, and $\mathcal{F}_n$ is the tree that has $\mathcal{F}_{n-1}$ and $\mathcal{F}_{n-2}$ as children.

It can easily be proved by induction that $|\mathcal{F}_n| = Fib(n+1) - 1$ where Fib is the Fibonacci function. Thus we have

$$s(n) \stackrel{\text{def}}{=} \sigma(\mathcal{F}_n) = s(n-1) + s(n-2) + c^{Fib(n+1)-1}$$

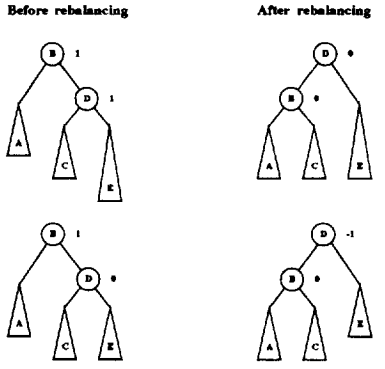

Figure 6: Rebalancing an AVL tree

In these 2 cases (and their mirror images) additional rebalancing can be performed without destroying the AVL property. The balance factors are indicated at the individual nodes.

This recurrence relation can be transformed to:

$$s(n) = \sum_{i=0}^{n} Fib(n-i) \cdot c^{Fib(n+1)-1}$$

This could be further transformed using the identity

$$Fib(n) = \frac{1}{\sqrt{5}}(\Phi^n - \overline{\Phi}^n)$$

where $\Phi = \frac{1}{2}(1+\sqrt{5})$ and $\overline{\Phi} = \frac{1}{2}(1-\sqrt{5})$:

$$s(n) = \sum_{i=0}^{n} \frac{1}{c\sqrt{5}}(\Phi^{n-i} - \overline{\Phi}^{n-i}) \cdot c^{\frac{1}{\sqrt{5}}\left(\Phi^{n+1}-\overline{\Phi}^{n+1}\right)}$$

Rebalancing

The algorithm can be further enhanced by performing some extra rebalancing to get a better weight balance. This will make the creation of new sets somewhat slower, but can speed up access to the new set, without incurring additional storage overheads. The enhancement is based on the fact that at certain places in an AVL tree it is possible to rebalance subtrees without loosing the AVL property. These cases are shown in figure 6 (the mirror images are not shown). In these cases rebalancing obviously does not disturb the AVL-property: the balance factor[2] for the individual nodes still remains within [-1,1]. However, the weight-balance for these nodes could be improved by this rebalancing operation. If we denote by $\#X$ the total number of nodes in subtree X, and by N the total number of nodes in the tree, then rebalancing will change the average path length by $\frac{\#A-\#E}{N}$. Thus, additional rebalancing is advantageous if $\#E > \#A$. Rebalancing can be done without causing any storage overhead if the operation is only performed on copied nodes. Thus in our example rebalancing would only be performed if nodes B and D were copied anyway.

[2]The balance factor of a node is equal to the difference in height between its left and right subtree.

A comparison of delta lists and AVL trees

There are two candidates for the implementation of versioned sets: delta lists and balanced trees. These data structures have different properties. The behavior of the delta list depends heavily on the value for the parameter f, the proportion of all versions that are stored in full as base version.

If storage space is at a premium, the delta list with a low value of f is better than a tree, due to the compact storage that is possible for delta-lists. But the access time will increase linearly with $(1 - f)$ so low values of f will give a bad access performance.

If, on the other hand, access time is more important than storage space, AVL trees will tend to be better, since their access time is always logarithmic in the size of the set. It seems likely that the delta list implementation with the same access speed as an AVL tree will consume more storage due to the large proportion of base versions.

Whether or not AVL trees are really superior in this case depends on implementation parameters and usage patterns.

7 Conclusions

Version management is an important ingredient of any software development process. It provides the basis for keeping track of changes to a system, and for managing and coordinating parallel development. Proper support for versioning is therefore a primary asset of any development environment.

In this paper we discuss an approach where version management is incorporated as a basic mechanism into an object system. Versioning is applied to complete, self-contained object worlds. This approach provides an intuitive and attractive way to improve consistency of version combinations, and avoids some of the problems that exist in other approaches, most notably that of version control of arbitrary references among objects.

Versioning does not come for free, and one of the main reasons that many systems do not provide version management is because it is too expensive. Of course, decreasing hardware costs, increased cpu power, cheaper storage media, such as WORM disks, make version control more attractive. Nevertheless, to make large scale application of versioning — and especially integral version management — viable, it is necessary to have suitable implementation techniques.

This paper presents several data structures and algorithms to implement integral version management which will be applied in the development of CAMERA. Although the techniques are applied to versioning of a self-contained object management system, we feel that it is possible to use them also in other situations, e.g. for the versioning of complex objects.

We expect that the data structures in this paper will give acceptable performance for frequent versioning during development. The results of our initial prototype implementations in this area are encouraging (some of this work is described in [Lip90]). More experience will be gained from the current prototype development of the complete CAMERA system.

This further research will also allow us to tune the algorithms to usage patterns. For example, the size of the index for a non linear history, that was described in section 6 depends on the chosen mapping function. Which mapping table performs best is determined by the actual usage patterns, e.g. the similarity between successive snapshots, the rate at which development lines branch and merge etc.

Acknowledgments

We would like to thank Jan Wielemaker and Doaitse Swierstra and the referees for their comments on versions of this paper.

References

[BCG*87] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou, and Hyoung-Joo Kim. Data model issues for object-oriented applications. *ACM Transactions on Office Automation Systems*, 5(1):3–26, January 1987.

[BGW89] David B. Miller, Robert G. Stockton, and Charles W. Krueger. An inverted approach to configuration management. In *Proceedings of the 2nd International Workshop in Software Configuration Management*, pages 1–4, November 1989. ACM SEN 17:7.

[BMO*89] Robert Bretl, David Maier, Allan Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams. The GemStone data management system. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, chapter 12, Addison-Wesley, 1989.

[F*88] Daniel H. Fishman et al. *Overview of the Iris DBMS*. Technical Report, Hewlett-Packard Laboratories, Palo Alto, 1988.

[GB86] Ira P. Goldstein and Daniel G. Bobrow. A layered approach to software design. In *Interactive Programming Environments*, chapter 19, Mc Graw-Hill, 1986.

[GMS89] W. Morwen Gentleman, Stephen A. MacKay, and Darlene A. Stewart. Commercial realtime software needs different configuration management. In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 152–161, November 1989. ACM SEN 17:7.

[Hen88] David Hendricks. The translucent file service. In *EEUG Autumn*, 1988.

[Hum89] Andrew G. Hume. The use of a time machine to control software. In *Software Management Workshop*, Usenix, April 1989.

[KBC*87] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. Composite object support in an object-oriented database system. In *Proceedings OOPSLA '87*, 1987.

[KL84] Randy H. Katz and Tobin J. Lehman. Database support for versions and alternatives of large design files. *IEEE Transactions on Software Engineering*, 10(2):191–200, March 1984.

[KL89] Won Kim and Frederick H. Lochovsky. *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, 1989.

[Knu73] Donald Ervin Knuth. *The Art of Computer Programming 3: Sorting and Searching*. Addison-Wesley, 1973.

[LCM*89] Anund Lie, Reidar Conradi, Tor M. Didriksen, Even-Andre Karlsson, Svein O. Hallsteinsen, and Per Holager. Change oriented versioning in a software engineering database. In *Proceedings of the 2nd International Workshop in Software Configuration Management*, pages 56–65, November 1989. ACM SEN 17:7.

[LF91] Ernst Lippe and Gert Florijn. *CAMERA: a Distributed Version Control System*. Technical Report 91/1, Software Engineering Research Centrum, 1991.

[LFB89] Ernst Lippe, Gert Florijn, and Eugène Bogaart. *CAMERA: Architecture of a Distributed Version Control System*. Technical Report RP/DVM-89/4, Software Engineering Research Centrum, April 1989.

[Lip90] Ernst Lippe. *Index Structures for Integral Version Management*. Technical Report 90/2, Software Engineering Research Centre, 1990.

[NSE88] *Network Software Environment: Reference Manual.* Sun Microsystems, March 1988.

[PAC89] *Configuration Management Guide.* PACT, December 1989.

[PCT] *PCTE: A Basis for a Portable Common Tool Environment.* European Economic Community, fourth edition.

[PPTT90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell labs. *EUUG Newsletter*, 10(3):2–11, 1990.

[Pre90] Vasilis Prevelakis. Versioning issues for hypertext systems. In Dennis Tsichritzis, editor, *Object Management*, chapter 6, pages 89–106, Centre Universitaire d'Informatique, Université de Genève, 1990.

[Rum88] James Rumbaugh. Controlling Propagation of Operations using Attributes on Relations. In *Proc. of the 1988 Object-Oriented Programming Systems and Languages Conference*, pages 285–296, September 1988.

[SR87] Michael Stonebraker and Lawrence A. Rowe (editors). *The POSTGRES Papers.* Memorandum UCB/ERL M86/85, Electronics Research Laboratory, U.C. Berkeley, June 1987.

[Tic85] Walter F. Tichy. RCS — a system for version control. *Software Practice and Experience*, 15(7):637–654, July 1985.

A Proof of NP-completeness

Lemma: Finding an optimal mapping function for a sparse index, as described in section 6.1 is NP-complete.

Proof:

First observe that the corresponding decision problem is in NP, since if we are given an ordering and its path length, it can be checked in polynomial time whether this ordering has indeed this path length. Since the decision problem is in NP, the problem of finding an optimal ordering is in NP, too.

Now we will reduce a standard NP-complete problem, a version of the Hamiltonian path problem, to our ordering problem. The Hamiltonian path problem can be stated as follows: Given a graph consisting of a set of nodes: $N = \{\nu_1 \cdots \nu_n\}$ and a set of edges: $E = \{\epsilon_1 \cdots \epsilon_e\}$ without double edges between nodes, find a path that passes exactly once through all nodes.

Now we are going to construct an instance of the optimal ordering problem for an instance of the Hamiltonian path problem. We construct a set of snapshots $\overline{\mathbf{X}} = \{\overline{\mathbf{x}}_1 \cdots \overline{\mathbf{x}}_n\}$, where

$$\overline{\mathbf{x}}_i = \begin{pmatrix} x_{i,1} \\ \vdots \\ x_{i,e} \end{pmatrix}$$

Every snapshot corresponds with a node in the Hamiltonian path problem. We represent a snapshot by a column vector that contains the values of the objects, thus $x_{i,j}$ contains the value of object j at time stamp i. For this proof we only need objects that can have integer values.

The contents of each $\overline{\mathbf{x}}_i$ is as follows:

$$\begin{aligned} x_{i,j} &= 0 \quad \text{if } \epsilon_j \text{ is connected to } \nu_i \\ &= i \quad \text{otherwise} \end{aligned}$$

This transformation can be performed in polynomial time.

Remember that the distance between two snapshots is equal to the number of objects that have different values in both snapshots. Observe that the distance between two points that represent two connected nodes is $e - 1$ while the distance for unconnected nodes is e. Now there exists a Hamiltonian path iff there exists an ordering of the $\overline{\mathbf{X}}$ such that the total path-length is equal to $(n - 1)(e - 1)$. □