LIMITED DISTRIBUTION NOTICE
(NOT TO BE REMOVED FROM DOCUMENT):

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communication and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

Version Coherence in Fine-Grained Object Storage and Its Interoperation with Versioning Approaches to Coarse-Grained Objects

William Harrison

IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, N.Y. 10598
harrisn@watson.ibm.com
(914) 784-7631

Abstract

Many uses of information occur in contexts which require the maintenence and manipulation of more than one version of the information in a store. In development environments, the need for support of multiple versions is motivated by alternate variants for modified function, sequential revisions for release of a design, etc.. Traditional approaches to versioning are generally effective only for large object structures, in which the developer can be dependended upon to avoid data incoherence. This paper explores an approach to automatically maintaining coherence and to relating this more sound approach to the incoherent versioning models currently in use today.

1. Introduction

Current development environments, like those found in Sun's NSE, or Draft ISO Standard for Portable Common Tools Environment (PCTE) are designed for the versioning of rather large (coarse-grained) objects, identifiable by the humans who manipulate them. These objects are units of information which have no internal state dependent on the representation of the internal state of other objects. In these approaches, the human-identified objects each have their own history of version creation and usage, separate from that of the others. Generally, some assistance is provided to form larger groups of these objects which are, at least locally, changed together in an attempt to preserve inter-object consistency. I will call such approaches *separate-object versioning*. Figure 1 illustrates such four such groupings of three such objects. The sequential numbering of versions is not intended to imply sequentiaI creation or use. These may follow any pattern like a two-level nesting of variants and revisions, (called branching and linear versions in [13]). I will use the term *versionable unit* for the units which can be separately versioned and the term *versioned group* for the groups manipulated together.

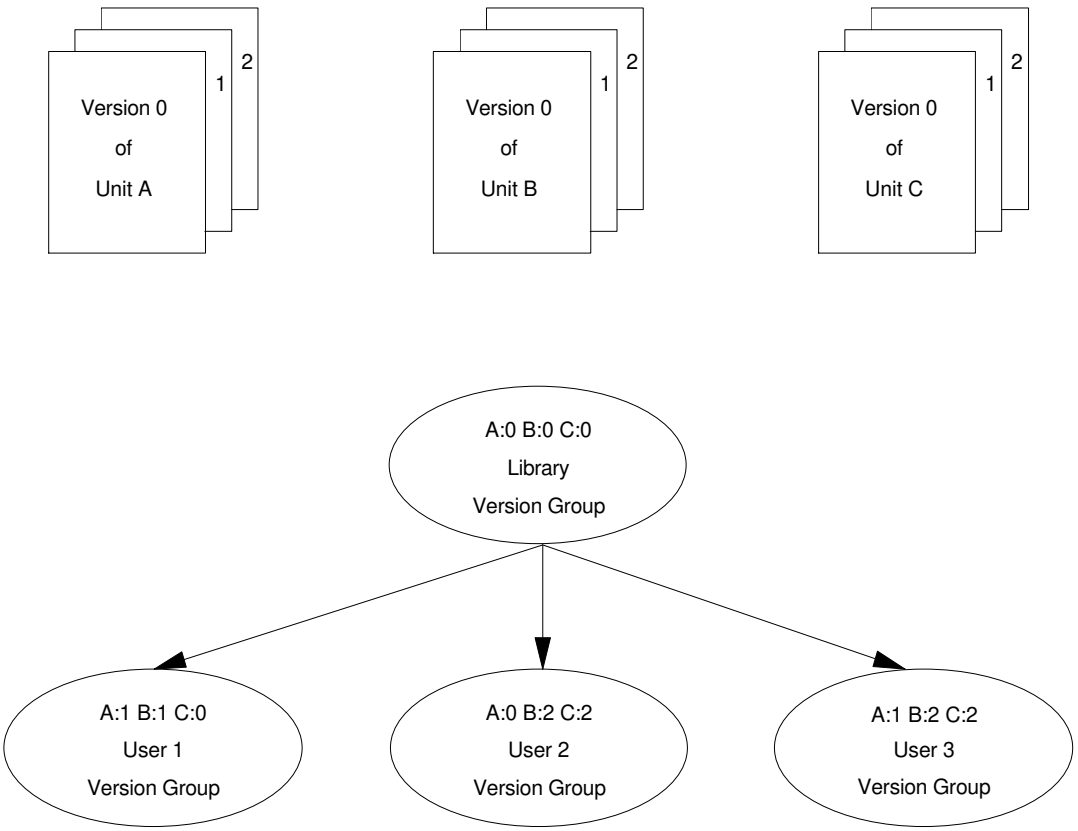

Figure 1 - Sample Versioning of Some Units Within Groups

More recently, the same approach to versioning has been suggested in fine-grained object contexts [2] or [9]. However, fine-grained object stores have two characteristics which make separate-object versioning unworkable in these contexts:

1. the versionable units are, in fact, collections of many smaller objects linked together by a rich web of pointers, or of relationships that are often implemented by pointers. For example, the collection representing a chip's logical design will often have objects that related in complex structure, and these objects will, in general, have relationships to physical placement objects, to engineer's names, to project-tracking events, etc., each stored in some other versionable unit.
2. neither the humans nor the tools manipulating these smaller objects possess the knowledge needed to predict or explicitly keep track of the versionable units changed by human interactions. For example, a human may change the name of an employee in some collection of employees in a versionable unit, but be unaware that an index stored in some other versionable unit is also updated, or that a consistency maintenence scheme also updates counters by surname in a third versionable unit. In a well designed Object-Oriented implementation, even the name-changing method may be unaware of the consequent activity.

Separate-object versioning of fine-grained information is unworkable because it can result in information in the store becoming *incoherent*.

- Section 2 of this paper discusses the problem of incoherence and how loss of coherence differs from, lack of atomicity and from serializability failures, and gives an example.
- Section 3 discusses strategies that can be used to prevent incoherence, exhibiting major drawbacks to each of them.
- Section 4 discusses Multiversioning, an approach to versioning that prevents incoherence but avoids these drawbacks.
- Section 5 presents a model for describing how multiversioning can interoperate with traditional separate-object versioning.
- Section 6 gives an example relating PCTE's versioning to multiversioning.

2. Incoherence

How incoherence arises is most easily shown in terms of a simple example. The example in Figure 2 extends Figure 1. Let us assume there are three versionable units: A, B, and C, each of which (ultimately) has 3 alternatives 0, 1, and 2. In this example, assume that two users, User 1 and User 2 each start from the versioned group in the library in which Units A, B, C, are version 0 (A:0, B:0, C:0). User 1 checks out A and B, getting new versions A:1 and B:1. User 2 checks out B and C getting versions B:2 and C:2[1]. The library version initially contains an object OB in versionble unit B.

In this example, objects are connected by a reversible 1-1 relationship $\mathfrak{R}$ stored as a pair of pointers, one in each object. User 1 changes the data so that OA $\mathfrak{R}$ OB. They now both point to each other in A:1 B:1 as shown for User 1 in Figure 2. Similarly, User 2 changes the data so that OB $\mathfrak{R}$ OC. They now both point to each other in B:2 C:2 as shown for User 2 in Figure 2. User 1 commits the changes for general availability and then User 2 commits the changes for general availability. Thus far all is fine. However, now assume that a User 3 obtains the most recent versions of the available versionable units: A:1, B:2, C:2. When the link is traversed from OA it goes to OB, but OB does not link back to OA (other scenarios in which OB is unallocated, OB is the wrong type, etc. are also all possible). The result is an unexplainable fault. The actual fault happens because a OB is referenced from A:1 without having selected B:1. We term phenomena of this sort *incoherence*. With incoherent information, assertions which are expected by and provable from programs on a local basis are not always true about the information in the store because of concurrent activity..

Traditional concurrency management techniques which guarantee serializability [6] do not prevent incoherence, because incoherence arises not from the operations which create and modify the data, but from selections of versions of versionable units to be used together. In our example, traditional concurrency management techniques guarantee that the changes made by User 1 show up in both A:1 and B:1 or in neither, and that the

---

[1]Actually we could have described this version of C as C:1 since no previous version of C exists except 0, but we skip to 2 to assist in keeping User 1 as versions 1 and user 2 as versions 2 for readability. There is no loss of generality, etc..

changes made by User 2 show up in both A:2 and B:2 or in neither. However, no constraint is made on the combination of the versions, like A:1 with B:1 or B:2. Such choices are made outside the serializability semantics altogether. In fact, it has been suggested [1,10] that the use of selected versioning control strategies can suffice for the management of concurrency in environmnents which need to support long (e.g. day long) trasactions. It is in this context that issues of coherence are raised to the same level of importance as atomicity and serializability.

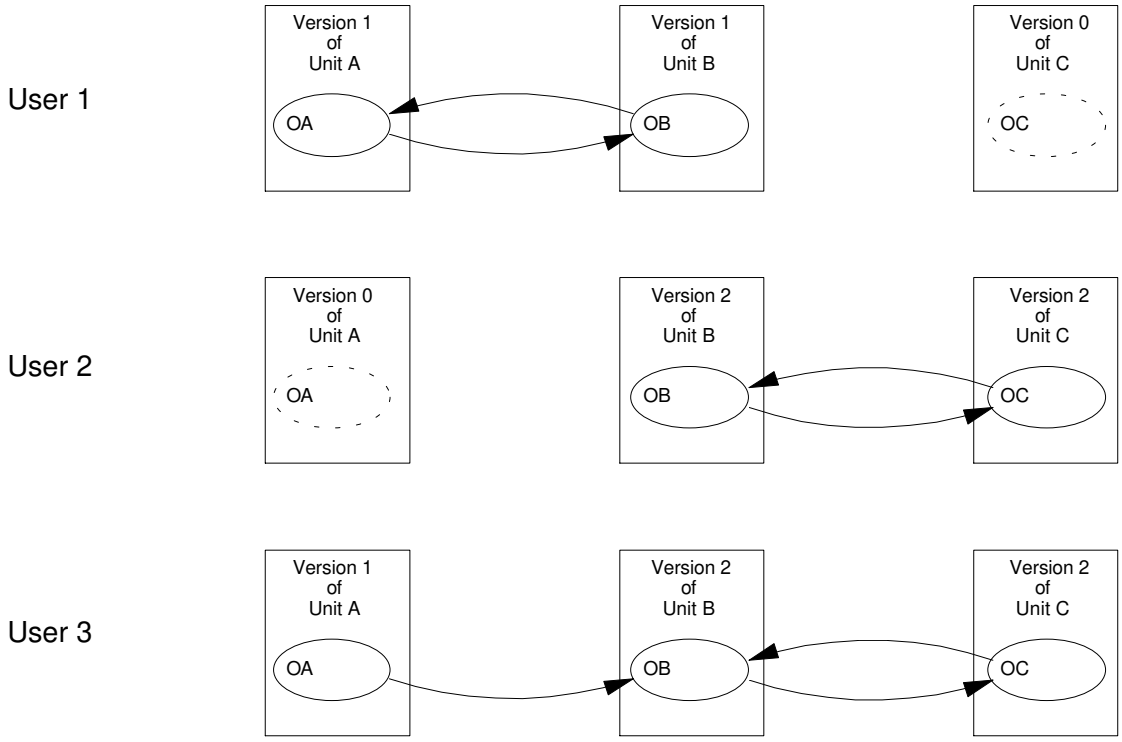

Figure 2 - Causing Incoherence in Object Stores

3. Preventing Incoherence

There are several strategies that are used to prevent or minimize incoherence:

prohibition	prohibit linking of objects in different versionable units,
propagation	automatically create versions of sufficiently many other objects to preserve the coherence
universioning	allow only a single versioned group
regulation	regulate the permissible sets of versions allowed within versioned groups.
checking	examine the permissible sets of versions allowed within versioned groups., checking for incoherence

Of these, all but the last (checking) provide computationally tractable solutions to the

version coherence problem. A brief summary of each, and of the disadvantages it imposes follows in the subsequent parts of this section.

Prohibition

Prohibition is the simplest way to guarantee coherence of a store with separate-object versioning. The store is partitioned into two levels, with each separately versioned object, sometimes called a *complex-objext*, containing many smaller objects. These may point to one another within a versioned object but, since no pointers or links exist between the separately versioned objects, no incoherence can arise. However, the use of prohibition to preserve coherence has strong impacts on the usefulness of the data model. For example, in a software data base, if each program is an object, no direct use-definition links can exist between a variable reference in one program and its declaration in another. Nor, in fact, can there be a relation linking a program to the programs that contain declarations it uses.

Prohibition, with its concommitant division of the objects into a two-tier store of complex-objects, is generally used in the hope that humans will be able to manage the smaller number of relationships among fewer versionable objects. But because of the limitations it imposes on evolution of data models, the prohibition is often bypassed in practice. In an attempt to remedy the consequent potential for incoherence, a permissive prohibition is often used used in conjunction with one of the other strategies, although it contributes little in such usages.

Prohibition is impractical because of the richness needed in the data models for complex systems like hardware and software development environments.

Propagation

Propagation is the approach used in PCTE[15]. In this approach, the links between objects are of three different sorts, which I will call *propagating*, *fanning*, and *dropping* links. When an object is versioned, a new object - the copy - is created and:

1. new versions are also created of all objects to which it is connected by propagating links, and corresponding links are set up between the copies corresponding to the originals,
2. additional links are set up between the copy and the objects linked to the original by fanning links,
3. objects related to the original by *dropping* links remain so and are not related to the copy.

In effect, the propagation strategy reduces the number of separately versionable objects. in the hope that humans will be able to manage their smaller number. Dropping and fanning links divide the store into separately versionable complexes. If there are any dropping links in the data model, propagation does not preserve the establlished relationships among the objects being versioned. In this sense, it causes incoherence because there may be semantic expectations of the existence on the relationships which

are not preserved. It can be argued that fanning links should then be used instead. However, the evolution of a data model may require versioning to occur at a boundary which was originally represented by a dropping link. Changing a model so that a link becomes a fanning link requires changes to the applications using the model.

Fanning links are, of necessity, many-to-many relationships which impose attendant requirements on tools traversing them to understand how to disambiguate the relationships. In particular, a tool must be written to understand:

1. that the link represents a transition between versionable regions of the graph, and
2. how to select appropriate version of the target of the relationship, even if some default is to be used.

The consequence is that defining a link as a fanning link just to cope with version creation requires tools to deal with significant additional complexity. This complexity is a source of fragility in the application structure. In particular, if the need for versioning forces a link to become a many-to-many instead of a one-to-one, any tool using the link will need to be modified.

In summary, dropping links lose information in the versioned copy and fanning links cause additional effort and complexity for tools that traverse them. However, if all links in the data model are propagating links, propagation becomes equivalent to universioning.

Universioning

Universioning guarantees coherence by permitting only a single versionable unit. It can be viewed as a special case of prohibition, in which links between objects in different versionable units are prohibited by virtue of the non-existence of such different versionable units., but it does not suffer from the drawback of restricting the data model.

Efficient implementation of universioning is possible, but requires great care. This care is needed because naive creation of new versions of a store requires creation of copies of objects. Delta-creation approaches like that described by [12] can be employed.

Universioning implies that versioning verbs apply to the entire store, rather than to individually identified objects. This makes it incompatible with several of the proposed industry-standards like ATIS[2] or like those proposed by Case Communiqué[14]. The whole coarse-grained community seems to be racing to standardize on models that cannot prevent incoherence without unacceptable burdens. On the other hand, universioning models are conceptually simple; a new version of the repository is created from an existing one by copying it.

Universioning does not interfere with the ability to separately version independent parts of the store. The problem of merging versions[3] can be equally well treated within a concurrency management framework for long-term transactions.[8].

The primary drawback of universioning is only its inability to give support to the apparent separate versioning hiererchies and histories for different parts of the store.

Regulation

Regulation guarantees coherence by making sure that dependent updates place constraints on the use of versionable units, guaranteeing that versionable units changed together are always used together. The most general form of regulation requires any set of versions changed together[2] to continue to be used together, with a closure placing the same requirement on derived versions. For instance, in the example of Figure 2, the A:1 B:2 C:2 combination is invalid because A:1 must be used along with B:1. (The A:1 B:1 C:2 combination would also be invalid for similar reasons.)

We can be more precise if we use a formal model. Consider the store to be comprised of versionable units U_i. Let us label the versions of U_i as U_{ij}. (Note that, as in our prior discussions, the ordering of the subscripts does not imply successive derivation.) Systems that employ such a store generally maintain a relation indicating whether versionable unit U_{ik} is derived from U_{ij}. Let us call it δ. To assure coherence, we need another relation to track whether two versions of two versionable units were modified together and therefore must be used together. Let us call it τ. Note that τ always relates a U_{ik} and a U_{jl} where $i \neq j$. It will be convenient to assume that δ is kept transitively closed, but τ is not. We can also represent each of the versioned groups G_i as a function from i's to j's specifying a version U_{ij} for each corresponding object U_i.

Using this notation, we can state the two important constraints relating τ, δ and the G_i's:

1. $U_{ik}\ \tau\ U_{jl}$ and $U_{ik}\ \tau\ U_{jm}$ only if $U_{jl}\ \delta\ U_{jm}$ or $U_{jm}\ \delta\ U_{jl}$. This means that τ cannot require two different versions of a U_i be used with the same version of U_i.
2. If $U_{ik}\ \tau\ U_{jm}$ and there exists a G_m containing a $U_{iG_m(i)}$ and a $U_{jG_m(j)}$, where $U_{ik}\ \delta\ U_{iG_m(i)}$, then $U_{jm}\ \delta\ U_{jG_m(j)}$. This means that if versions of two versionable units are constrained to be together and a versioned group contains a version derived (via δ) from one of them, then the version of the other in that versioned group must be derived from the other.

Formulating the constraints in this way helps us to understand the complexity costs of regulation. In general, creation of a new versioned group may cause a violation of the second constraint. In addition, the joint manipulation of two versionable units in a versioned group may create a new linkage in τ. Creation of this linkage may cause a violation of constraints 1 or 2. While detecting violations that occur at versioned group creation may be done in time linear on the number of versionable units involved,

---

[2]"Together" might have many different interpretations, from the smallest atomic operation to a "commit" following a long transaction. This discussion is equally applicable in any case.

algorithms for detecting violations at times when τ is extended are essentially the same as deadlock prevention algorithms in locking systems, are nontrivial, and require global access to the information in δ and τ.

There are a number of ways to avoid the cost of maintaining τ and its constraints by imposing a set of rules on the formation of versioned groups and/or updates. One such set of regulations exploits concepts like variant(branch) and revision(version) and prohibits transfer of a versionable unit from one variant to another, requiring instead the creation of a new revision in the target variant and the safe merging [7] of the information from the source variant. In terms of our original example, the attempt to introduce B:2 C:2 into a branch containing A:1 B:1 would be rejected because B:2 conflicts with B:1.

In general, the drawback of regulation is either the complexity of the maintenence of τ or of the complexity of proving that the induced series of behavioral rules will guarantee this coherence. Each time the versioning model is altered, it must be done so by developers sufficiently expert to re-prove that the resulting set of rules still avoids incoherence. Simple fixes introduced by unskilled developers may allow accidental incoherences to begin to arise because of "bugs" in the versioning code.

Checking

Regulation requiring safe merges can be relaxed to the use of unsafe merges if sufficient checking can be employed to guarantee coherence. Sufficient checking is, however, generally not possible. There are two requirements that must be met by a checking system:

1. The list of constraints be explicitly stated. This is generally impossible for two reasons:
   - Incompleteness — Each piece of software which effects a transformation of the store embodies a series of constraints, whether stated or unstated. For example, it may always update certain attributes pairwise, or make sure that a digital signature matches its owner's. These constraints are often implicit, arising originally as computational accidents which are then presumed as facts by software written later.
   - Inexpressibility — If all constraints must be explicitly stated, then the constraint language must be sufficiently powerful to express all of them. This means that the language must be recursive. It is often the case that constraint languages fail to allow the expression of complex constraints. But if even one constraint cannot be expressed, checking will fail to provide coherence. The contents of significant portions of the store can become unusable.
2. The check must be computable in reasonable time. There are 3 levels at which this fails to be possible:
   a. In the most general situation, any constraint language powerful enough to meet the expressibility requirements must involve boolean expressions, and general proof problems in this domain are known to be co-NP-complete[4]. It could, however, be argued that since the data base itself is finite, this fact may limits the amount of computation required. But it turns out that the problem of determining

the validity of predicate calculus assertions about integers in a database can be exponential in complexity.[5].

b. Or it can be argued even further that not only is the database finite, but the constraints to be solved are computationally simple in themselves. This argument falls on two grounds:
   i. Some useful constraints are not computationally simple. For example, suppose a store contains documents and indexes by certain keywords. Changes to a document imply changes to the index. However, selection of two versions for the document and the index can yield inconsistent results. The constraint checking to recognize this and indicate the inconsistency is tantamount to rebuilding the index from scratch.
   ii. Even when the constraints are computationally simple, sophisticated analysis of the constraint interdependencies will probably be needed to keep the computational complexity from becoming exponential. It has been shown that simple change propagation can require time exponential in the size of the database[11].

In summary, there are three reasons why checking is inadequate to provide coherence - incompleteness, inexpressibility, and computational complexity.

4. Multiversioning

The primary disadvantage to universioning is that it does not directly allow expression of the often-felt need for separate versioning of relatively disjoint "parts" of a software system - parts like the supervisor vs. a compiler. Each of these tool groups needs to form its own conventions for the branch and variant expression to reflect its own view of development and release cycles. As we have argued earlier, maintaining coherence in the face of separate-object versioning is difficult and although prohibition can be used in this case, numerous cases do arise in which some connections among the versionable units are too useful to ignore. Since only universioning presents an acceptable solution, a universioning solution which allows construction of parallel development hierarchies, limiting conflicts to "real" cases, would be quite advantageous. This provides the motivation for *multiversioning* which allows only a single versionable unit, but permits software to operate on many versions of a versionable unit at the same time.

The ability to manipulate many versions at once allows a simulation of more than one versioning hierarchy and/or more than more than one separately versionable object. When versioning is used for check-in/check-out it is actually providing a long-transacrion concurrency management service, rather than real support for sharing of data among versions of a component which are never intended to be reconciled or merged. Uses of such real versioning include examples like an AIX and a Berkeley-Unix version of a tool. Many of the componentsand the share relationships to other system elements are common to both versions, but some differences are also present. One of the other advantages of multiversioning is that it gives integrated support to the use of versioning for these situations, and not just for long-transaction concurrency management.

Using multiversioning, the possible confusion arising from changes to the same object in separate version hiererchies does not entirely go away. However, instead of creating incoherence in the database, scenarios like that illustrated in Figure 2 result in resolvable ambiguities. These ambiguous situations arise only with respect to those specific objects having a multiplicity of versions when seen from a single point in one hierarchy. They ambiguities can be resolved to give clearer insight into which version should actually be used.

For example, from the point of view of a developer in the AIX project mentioned above, a single version of some module might be being selected. However, because of changes made by the Berkeley-Unix project, this single module might actually have more than one interpretation because it is shared between the AIX and Berkeley-Unix team.

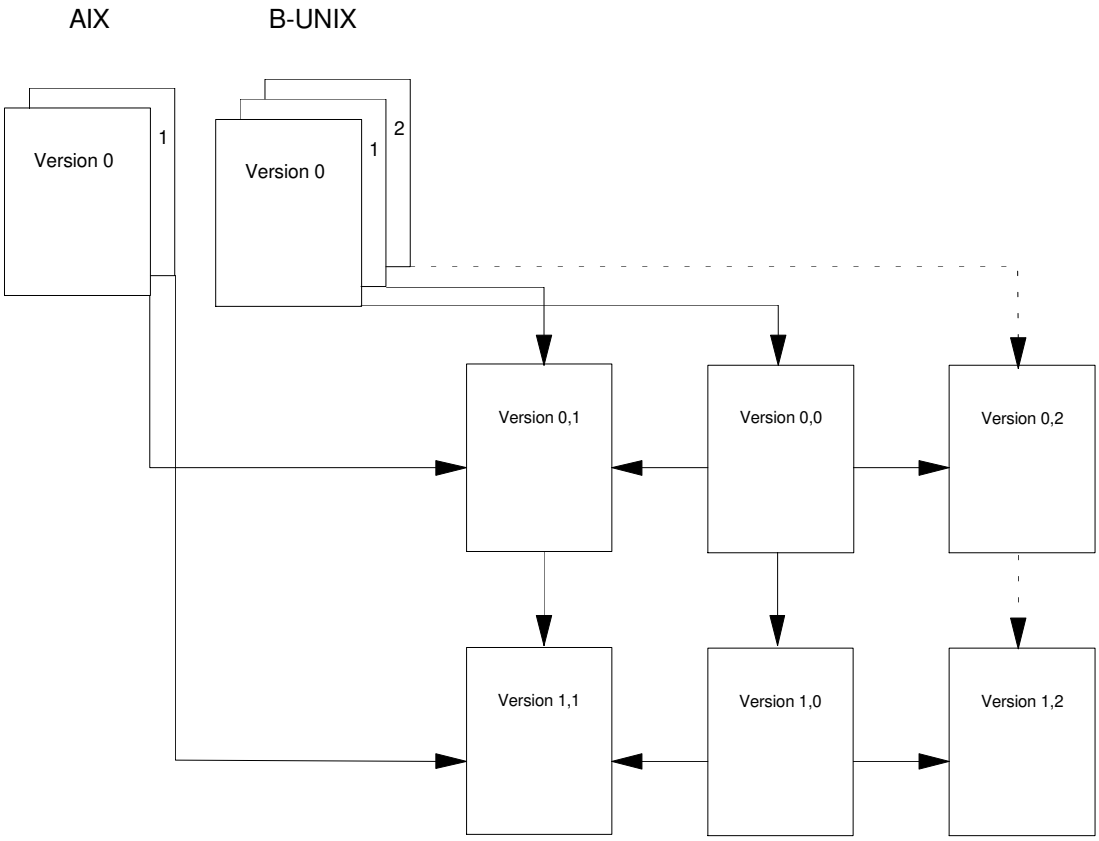

Figure 3 - Multiversioning with Two Versioning Dimensions

Figure 3 illustrates this example by considering a simplified model in which there are two version spaces, AIX and BU (Berkeley-Unix). Let us start with a repository containing only the version labelled 0,0 and assume the AIX developers and the BU developers both call this version 0. If the AIX developers create a version 1 from this version, and the BU developers create both a version 1 and a version 2 from this version, we get the 6 actual versions of the store shown in the figure. The versions labelled (0,0), (0,1), and (0,2) are called "version 0" by the AIX developers. Similarly, the versions

labelled (0,1) and (1,1) are called "version 1" by the BU developers.

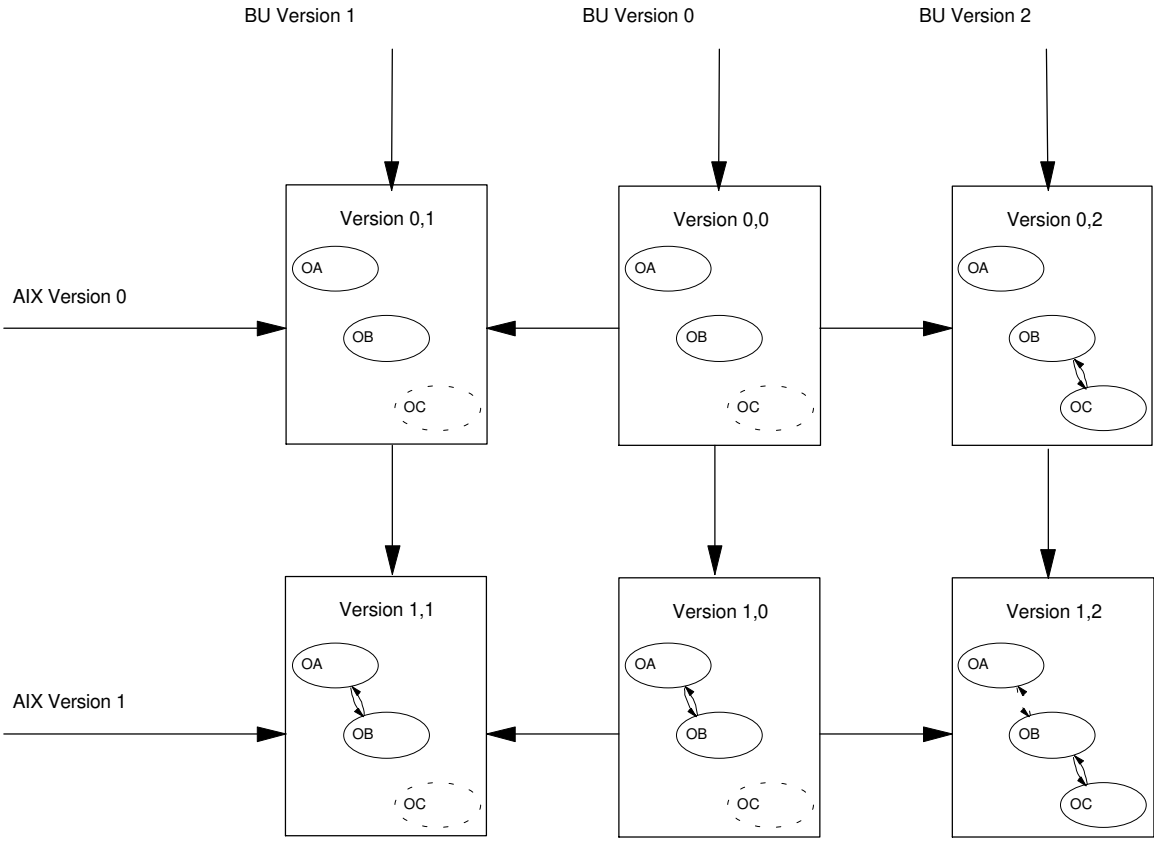

Figure 4 - Multiversioning Preventing Incoherence

Figure 4 illustrates the multiversioning consequencess of scenarios that lead to incoherence in separate-object versioning architectures. We replay the scenario that leads to the incoherence shown in Figure 2. Although this scenario contains an unresolvable conflict, leading to incoherence or some other exception, as long as changes made in AIX versions and BU versions are to disjoint sets of objects, the results are identical to the use of two versioned groups.

Assume the AIX developer first selects version 1 and then links objects OA and OB as before. When the AIX developer selects AIX version 1, operations are applied to the underlying versions (1,0), (1,1), and (1,2). Since the contents of objects OA and OB are the same in all of these versions, it is simple enough for the store to reflect the execution in all three at the same time. (Note that if BU versions 1 or 2 are not yet created, there are fewer versions to be modified in parallel, but the end result is the same since the subsequent creation of BU versions 1 or 2 copies the modifications.)

Assume the BU developer subsequently selects version 2, corresponding to the underlying versions (0,2) and (1,2). When the developer links OB to OC, it turns out that OB has different values in the (0,2) and (1,2), and the store indicates a multiple-value exception. It is important to note that in these circumstances, multiversioning has two advantages over separate-object versioning:

1. the store does not become incoherent and either version can be elected by reducing the set of versions being manipulated, and
2. the ambiguity indication is detected and localized to its point of impact, so that it can be explained and/or resolved.

To proceed, the application must narrow the range of the versions upon which it operates, either operating on (0,2) or on (1,2). The results of operating on (0,2) are simply shown in Figure 4. A similar result occurs if the change is applied to (1,2), except that the link from OA to OB previously established is removed in the process (shown as dotted arrows). If this latter course is pursued, attempts to manipulate OB when viewed as AIX version 1 will result in a multiple-value exception.

5. Interoperation of Multiversioning with Separate-Object Versioning

Given the number of repositories being constructed today using separate-object versioning for coarse-grained objects, it is important to develop a model which allows obects in a universioned or multiversioned store to maintain semantically meaningful relationships with objects in a store with separate-object versioning. One way of doing so exploits some of the same optimization mechanisms that can be used to efficiently implement the uni/multi-versioned store in the first place. We will describe this mechanism in terms of a concept called *value-set*.

Value Sets

A value-set is simply a set of versions of the store in which the object has been determined to have the same values for all of its attributes and links. One way in which a version set grows is by creation of new versions of the store from earlier versions - unless an object is otherwise changed both the original and the copy of the object can be in the same version-set. Note that this definition allows for the possible existence of two value-sets for an object which could be joined into a single value-set, although some quirk of the history or implementation may have resulted in this possible join being undetected and unrealized. Figure 5 illustrates one way in which the scenario used for Figure 4 partitions the objects' values into value-sets. It will be instructive to examine how the sequence of modifications results in the value sets for various objects.

Object OC has the simplest history, and the simplest collection of value-sets. If an object is never modified, it has only a single value-set, containing the one collection of values it owns. In the case of OC, the object has the same untouched contents (value-set OC1) in the versions that BU calls 0 and 1 and AIX calls 0 and 1, versions (0,0), (0,1), (1,0) and (1,1). However, since OC was changed separately in versions (0,2) and (1,2), it has two other value sets, one for each. In fact, the object has identical contents in both value set OC2 and OC3, so a smarter implementation might have used just a single other value-set, OC2.

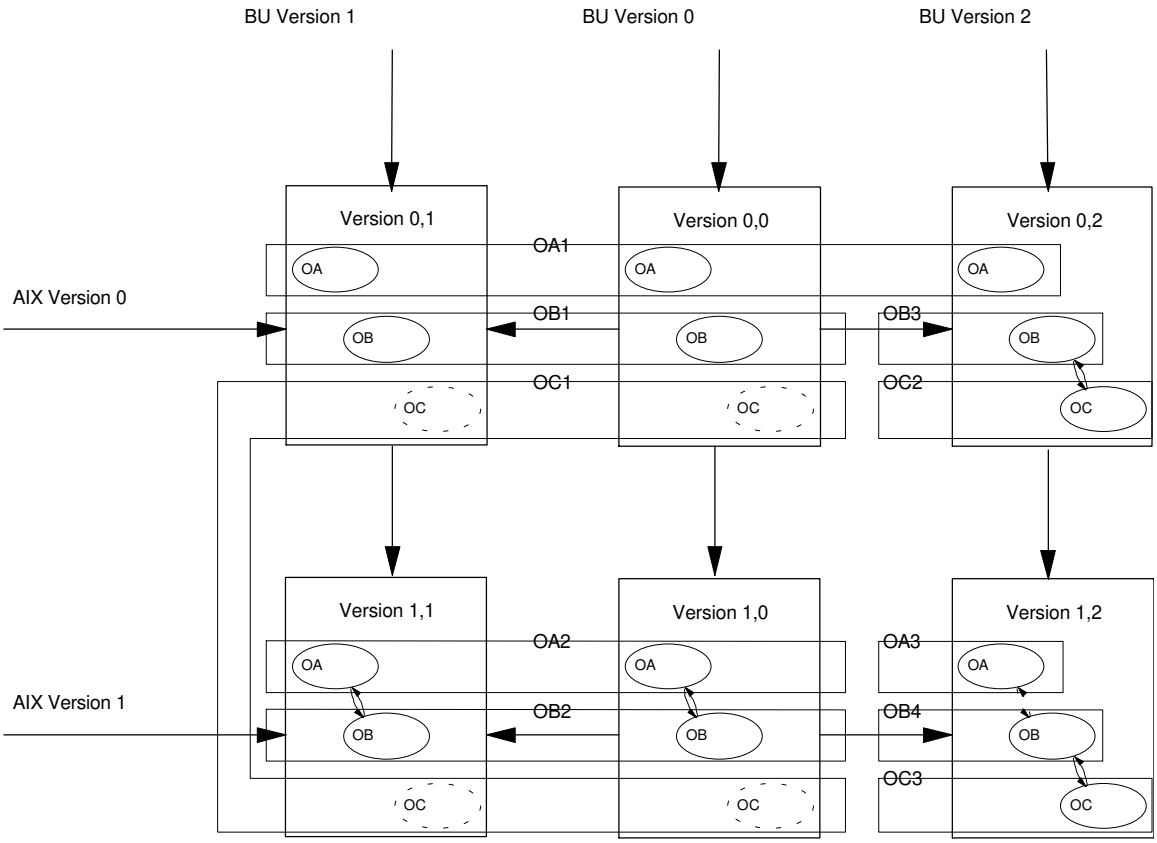

Figure 5 - Versions of Objects Grouped into Value Sets

Object OA has a somewhat more complex history and corresponding value-sets. It is unchanged in the versions AIX calls 0, that is in (0,0), (0,1), and (0,2), so that this content can be called value-set OB1. In the versions AIX calls 1, OA is linked to OB, and the resulting value-set OB2 initially covers (1,0), (1,1) and (1,2). However, a later change is made in (1,2). This final change, de-linking OA results in the third value-set (OA3) for version (1,3). Once again, OA3 could have been merged with OA1 since OA has the same contents in both.

Object OB has the most complex history. The unchanged values in OB1 reside only in the space changed neither as AIX version 1 or BU version 2. This version-set (OB1) spans the remaining versions (0,0) and (0,1). When changes are made in AIX version 1, a new version-set (OB2) is created to reflect those changes. The subsequent change in BU version 2 (for AIX version 0) requires the creation of the version set OB3. Similarly, the even later change in BU version 2 (for AIX version 1) requires the creation of version set OB4. As with other cases, OB3 and OB4 could actually be coalesced into a single OB3.

Value-Sets and Separate-Objects

The concept of value-sets, which can be used to reduce the storage costs for a uni/multi-versioning store can be employed to relate objects in a multiversioning system to those in separate-object versioning system. In general, since both the multiversioning system and the separate-object versioning system have versioning models, complete with native

operations for creating and updating versions, we must describe the interoperation in terms of these native operations.

Separate-object versioning systems employ a version creation operation to produce a new object which is a different version of an original object. In addition, the system maintains an accurate derivation relationship among the versions. We will use the notation $C_S(U_{ij})$ to indicate the creation of a new version of object U_{ij}, and the relationship δ_S to indicate if object U_{ik} is derived from U_{ij}. This is like the notation introduced in the earlier section on Regulation for preventing incoherence. In addition, we will use the notations $R_S(U_{ij})$ and $W_S(U_{ij})$ for reading and updating object U_{ij}, respectively
.
Uni/multi-versioning systems employ a version creation operation to preoduce a new version of the entire store. Although these creations may be in response to complex user models of versioning, such as the multidimensional one introduced in the earlier section on multiversioning, it suffices for us to deal with a basic model which uses simple integers for the versions of the store. Like the notation in the above paragraph, we will use the notation $C_M(i)$ to indicate the creation of a new version of store version i, and δ_M to indicate if store version k is derived from j We also use the notations $R_M(U_i^j)$ and $W_M(U_i^j)$ for reading and updating object U_i in store version j, respectively. In addition, we denote the value-sets for an object U_i in store version j by $V_M(U_i^j)$.

We will adopt a semantic mapping that has the following characteristics:

1. For each collection of objects U_{ij} related by δ_S we identify a fictional *representative* object identity that allows us to talk about U_i notationally. Since U_i does not really exist, we must always qualify it with a store version j by writing U_i^j. We will now employ a notational sleight-of-hand which has rather subtle semantic implications. So, let us expose the implications in advance. We wish to extend the previously defined notations for R_M, W_M, so that they refer to U_i^j when U_i is a representative object.

   Although notationally convenient, we should observe that the objects in the separate-versioning store are not actually in versions of the multiversioning store. Thus it would be meaningless to write, for example, U_{ik}^j. This conseqence is quite important. It means that the suffixes S and M which are used to qualify the R and W operations are redundant since the appropriate behavior can be determined from the identity of the object, U_{ij} or U_i^j. In other words, we may construct implementations in which the actual read and update function interfaces are insensitive to which store the object is in.

2. We must also extend the definition of V_M to cover the representative objects U_i. To do so, the versions of an object in the separate-object versioning store must partition the versions in the multiversion store so that they can be viewed as a value-set of the representative object in the multiversion store. We must have for any i, $((V_M(U_i^j) \cap V_M(U_i^k)) \neq \phi)$ iff $(V_M(U_i^j) = V_M(U_i^k))$.

3. This property allows us to define a function κ which relates each U_i^j to a corresponding U_{im}. From the previous we then have $(\forall j,k)(V_M(U_i^j) = V_M(U_i^k)) \rightarrow (\kappa(U_i^j) = \kappa(U_i^k))$.

6. Interpretation of Version Sensitive Operations in PCTE

We will illustrate the interpretation of a program on these stores by discussing the coordination of the snapshot semantics of a multiversion store with PCTE, a separate-version store, via the operation of R_S, W_S, C_S, R_M, W_M, and C_M. The mapping is well-behaved in terms of its snapshot semantics, the semantics of each operation expressed on a given state of the store. We will not claim that the semantics preserves the temporal semantics as well. For example, a listing of all of versions of an object in the separate-version store may be rendered out-of-date by an update made in the multiversion store. The new state will, however, be semantically meaningful and obey all the expected constraints.

We assume that a multiversioning context is set for the execution of any program, even one which manipulates only the objects in the separate-versioning store. This context is specified as a set of versions μ. It suffices to supply a default value of μ to be used when none is specified.

Figure 6 shows a sample system, with a single version in the multiversioning store. The PCTE store contains three objects: U_{31}, U_{41}, and U_{51}. The multiversioning store has two "native" objects U_1, and U_2, and three "representative" objects U_3, U_4, and U_5. The PCTE objects have two relationships *p* (intended to be a propagating kind of relationship like a composition link) and *d* (intended to be a dropping kind of relationship like a designation link). The mapping κ is illustrated by the lines from the representative objects to the PCTE objects.

Reading

The operation of $R_S(U_{ij})$ is unchanged, as is the operation of $R_M(U_i^j)$ when U_i is not a representative object. When U_i is a representative object, two cases arise:

1. $(\forall\ m \in \mu)(V_M(U_i^m) \supseteq \mu)$ This means that the versions specified by the context all map to the same separately-versioned object. In this case, we have $R_M(U_i^j) \equiv R_S(\kappa(U_{ij}))$.
2. otherwise there is a multiple-value exception of the kind described in the section on Multiversioning.

These definitions preserve both the snapshot and temporal semantics of each versioning scheme (easy enough when you're just reading the information).

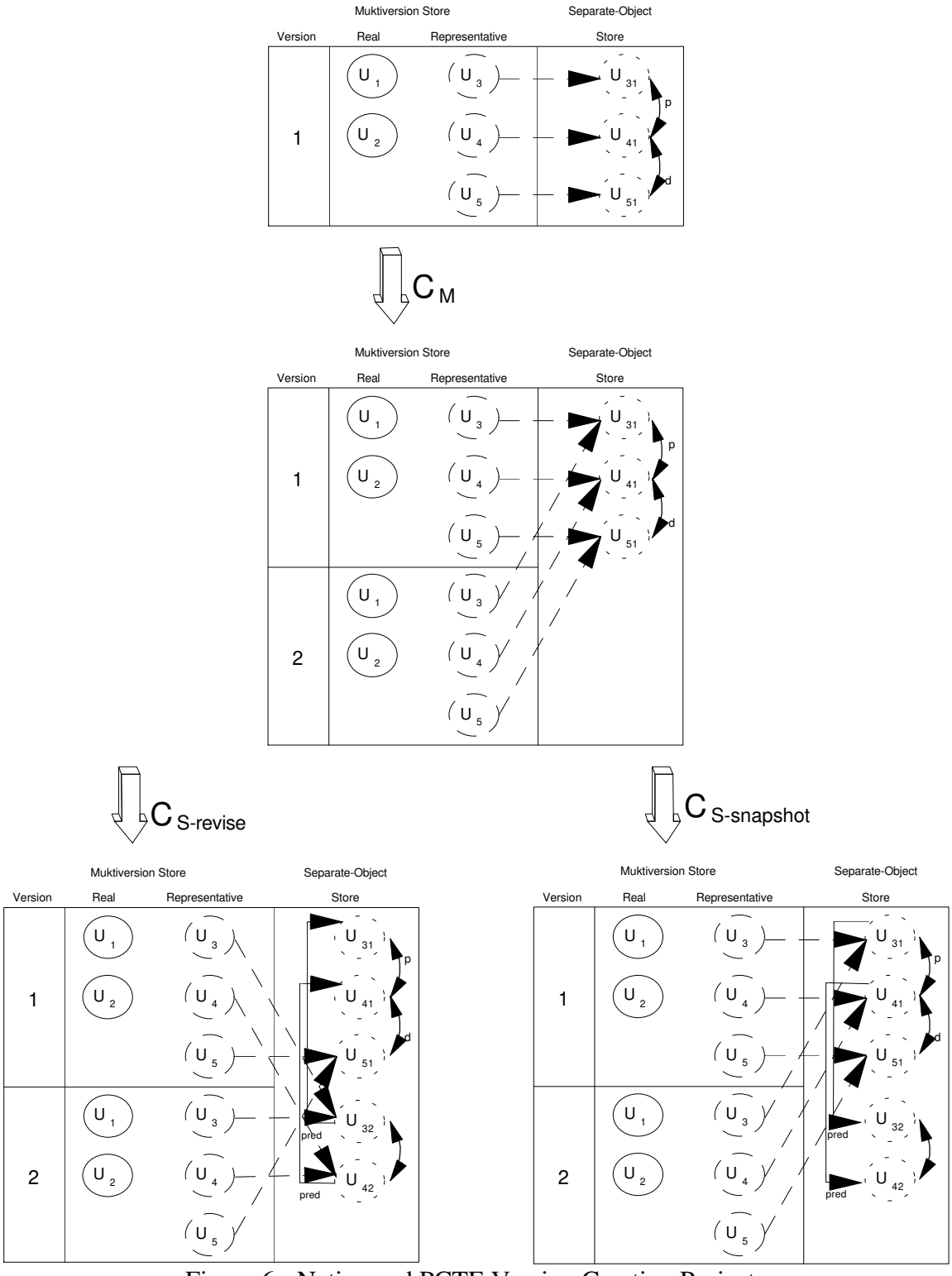

Figure 6 - Native and PCTE Version Creation Projects

Creating Versions

The native version-creating function C_M must be augmented somewhat to preserve the κ function, although its snapshot semantics remain unchanged.

If $C_M(i)=k$, then C_M must adjust the version-set partition V so that $(\forall i)(V_M(U_i^k) = V_M(U_i^j)$. This would imply that $(\kappa(U_i^k) = \kappa(U_i^j))$. Figure 6 illustrates this operation of C_M, showing the resulting version 2 of the multiversion store and the fact that κ maps the representative objects U_3, U_4, and U_5 to U_{31}, U_{41}, and U_{51} in both versions of the store. A similar extension is made to the C_S; $C_S(U_{ij})$ operates exactly as it does natively, yielding an object we will denote as U_{ik} and updating δ accordingly. It then must also modify κ so that the version-set partition covers the newly created U_{ik}. The exact choice made for this covering should be made in a manner appropriate to the C_S. For example if the representative should be mapped to the new U_{ik} the new κ should be $(\forall\, m \in \mu)(\kappa(U_i^m) = U_{ik})$. PCTE has two version-creating operations, labelled C_S-revision and C_S-snapshot in Figure 6. As can be seen, both of these copy through the propagating link and omit past the dropping link. In PCTE's terms, the predecessor relation is also established, although differently for C_S-revision and C_S-snapshot which differ in which copy is considered the "active" one and which is therefore its stable predecessor. To make the correct "active" copy be the one designated by the multiversioning store, κ is adjusted in correspondingly different ways for each operation.

The observant reader will notice that use of U_5 in version 2 of the S-revised store leads to incoherence. This is, in general, an example of the fact that interoperating a multiversioned store with a sepatare-object versioning store does not remove the potential for incoherence present in the separate-object store.

Writing

As with reading, the writing operations, $W_S(U_{ij})$ and $W_M(U_i^j)$ are unchanged when U_i is not a representative object. When U_i is a representative object, three cases can arise:

1. $(\forall m \in \mu)(V_M(U_i^m) = \mu)$. This means that the set of versions specified by the context is exactly the set of versions which map to the same separately-versioned object. In this case, we have $W_M(U_i^j) \equiv W_S(\kappa(U_{ij}))$.
2. condition (1) is false but $(\forall m \in \mu)(V_M(U_i^m) \supset \mu)$. This means that the corresponding object in the separate-object versioned store represents a value-set that contains more versions than those in the current version context. We say that a *version split* is required. Version splitting is the subject of the remainder of this section.
3. otherwise there is a multiple-value exception of the kind described in the section on Multiversioning.

Version Splitting

For the write operation to apply only to the versions in the context, the corresponding separately-versioned object must be restricted to be the value-set for only versions not in μ. In addition, a new object must be created within the separately-versioned store to represent the new version. The new object is the value-set for all the versions in μ

It is this creation of a new version of the object in the separately-versioned store which may cause violations of the temporal semantics of the separately-versioned store. This

effect occurs because an update which causes a version-set to be split into two separate version-sets causes the creation of a new version in the separate-versioning store. This is infrequent, generally a W_M operation does not cause the creation of new versions in the separate-versioning store. This split takes place as follows:

1. Assume that the versioning context is μ and that a write operation $W_S(U_{i\mu})$ being performed which meets the conditions described above for a version split. (These conditions justify the use of μ inside the W form.)
2. Assume that prior to the "write" operation, the versioning state was described by δ_{0S}, V_{0M}, and κ_0.
3. For convenience let $\kappa(U_i^{\mu})$ be $U_{i\omega}$, let the newly created version in the separately versioned store be $U_{i\upsilon}$ and let $U_{i\pi}$ denote the object from which $U_{i\omega}$ is derived according to δ_{0S}.
4. Then the new value of V_M is given by:

$$V_M(U_i^j) = \begin{cases} j \notin \mu & \Rightarrow V_{0M}(U_i^j) \\ j \in \mu & \Rightarrow \mu \end{cases}$$

5. and the new value of κ is given by:

$$\kappa_M(U_i^j) = \begin{cases} j \notin \mu & \Rightarrow U_{i\omega} \\ j \in \mu & \Rightarrow U_{i\upsilon} \end{cases}$$

6. and if permitted by the constraints on valid forms for δ_S, the new value of δ_S is given by: $\delta_S = \delta_S + (U_{i\pi}\ ,\ U_{i\upsilon})$.

   This is a branch (variant) version of the same version which gave rise to the object from which $U_{i\omega}$ derives. If the constraints on the allowable forms for δ_S do not permit the creation of this branch variant, we could employ a different construction for the new δ_S: $\delta_S = \delta_S + (U_{i\omega}\ ,\ U_{i\upsilon})$.

   This is a sequential (revision) version descended from $U_{i\omega}$ itself. Of course, this may also not be permitted under the constraints on δ_S:, or may require $U_{i\omega}$ to become unacceptably read-only. In general, there are hypothetical constraints on δ_S which may be imposed by the separate-object versioned store which may make version-splitting semantically invalid. In these cases, we signal a *version-policy-mismatch* exception instead. Fortunately, stores like PCTE and NSE generally permit us to elect the first alternative.

Figure 7 shows two examples of version splitting, using the example from Figure 6. In the simpler case, an update is made to U_5. Since the underlying PCTE object U_{51} is initially shared, a new version (revision) of U_{51} is needed before this can take place. This

revision becomes the object represented by U_5 in version 2, while the original U_{51} remains the object represented by U_5 in version 1..

In the more complex case, a similar change is made to U_3. The difference here is that U_3 bears relationships to other objects that must be treated according to PCTE's version-creation semantics. The splitting of the corresponding U_{31} also splits U_{41}, with the new PCTE objects being represened appropriately in version 2 of the store.

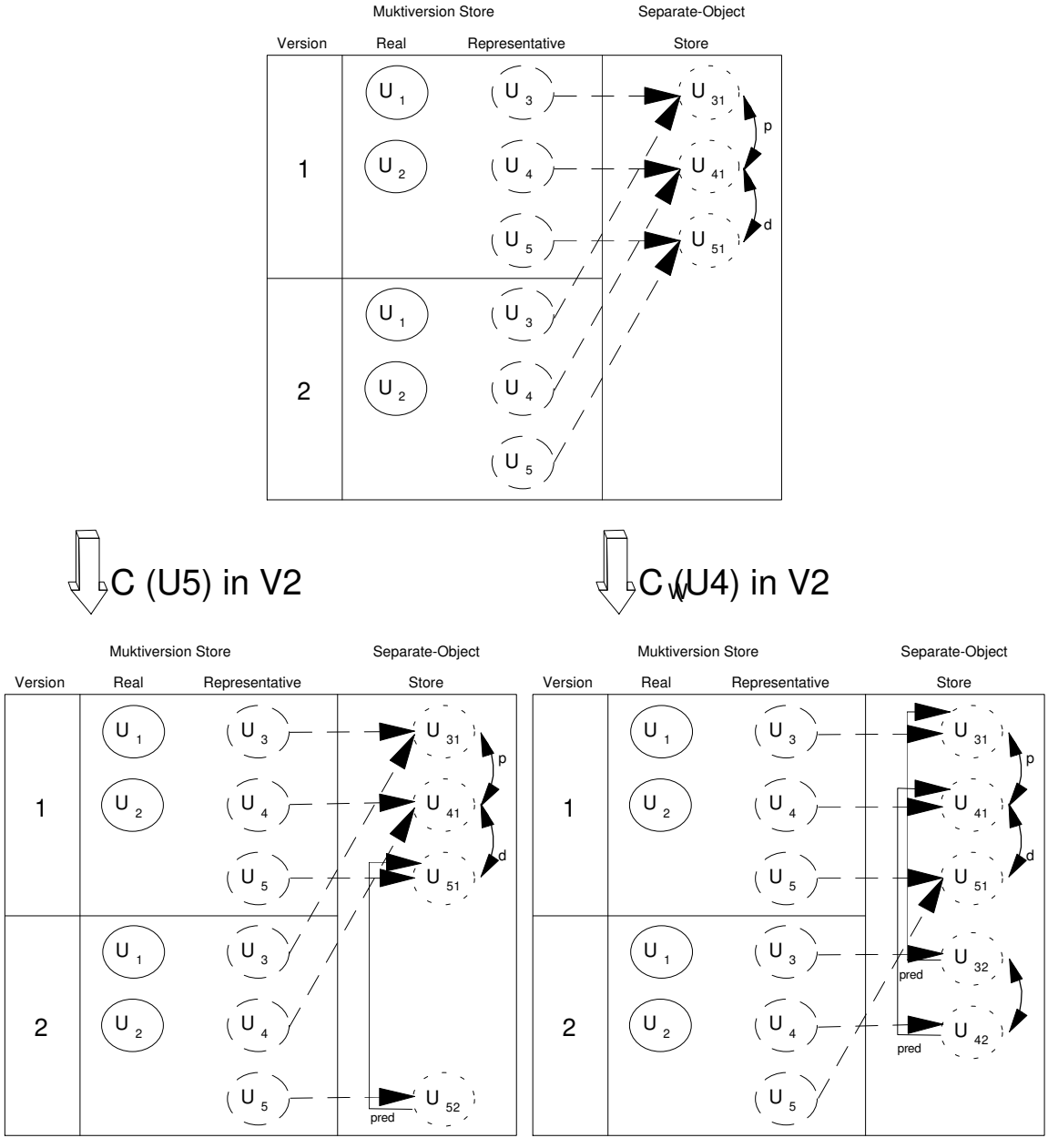

Figure 7 - Native and PCTE Version Splitting on Update

Summary and Future Work

Conventional models of versioning, called separate-object versioning, are subject to corruption of information of information integrity charcterized as *incoherence*. Several ways of minimizing or eliminating incoherence can be employed, but most of them suffer from important disadvantages. A solution approach, multiversioning, exists which avoids these difficulties. We make use of a formalism to describe the interoperation of multiversioning with separate-object versioning. The feasibility of such interoperation is shown and examples of the interoperation with PCTE are provided.

References

[1]Evan Adams, Masahiro Honda, Terrence Miller, "Object Management in a CASE Environment", Proceedings of the 11th International Conference on Software Engineering, May 1989, Pittsburg, PA.

[2]Augusto Argento, Chiara Bonferini, Fabrizio Dematté, Serena Manca, "ECMA PCTE, CORBA, and ATIS 'the easy way to O-O ECMA PCTE services'", PCTE Newsletter Number 10, June 1992

[3]A. Björnerstedt and C. Hulten, "Version Control in an Object-Oriented Architecture", in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley, Reading Mass., 1989, pp. 451-485

[4]S. Cook, "The Complexity of Theorem Proving Procedures", Proceedings of Third Annual ACM Symposium on Theory of Computing, May 1971, New York, NY

[5]Michael Garey, David Johnson, *Computers and Tractability*, W. H. Freeman, San Francisco, 1979, p. 47 ff.

[6]James Gray, "Notes on Data Base Operating Systems", in *Operating Systems, An Advanced Course*, (Bayer, Graham, Seegmüller, Eds.), Springer Verlag, Vol. 60, Berlin 1978

[7]William Harrison, Harold Ossher, Peter Sweeney, "Coordinating Concurrent Development", Proceedings of the Conference on Computer Supported Cooperative Work, October 1990, Los Angeles CA

[8]William Harrison, Harold Ossher, and Peter Sweeney, "Coordinating Concurrent Development", Proceedings of the Conference on Computer-Supported Cooperative Work, (New Orleans), pp. 157-168, ACM, October 1990.

[9]W. Kim, et. al., "Architecture of the Orion Next-Generation Database System", IEEE Trans. Knowledge and Data Engineering, Vol 2, No. 1, 1990, pp 109-124

[10]David Leblang, Robert Chase, Howard Spilke, "Increasing Productivity with a Parallel Configuration Manager", Proceedings of the Workshop on Software Version and Configuration Control, (Berichte des German Chapter of the ACM Vol. 30), January 1988, Grassau, Germany

[11]Thomas Reps, "Generating Language-Based Environments", ACM Doctoral Dissertation Award 1983, MIT Press, 1984, pp. 29-48

[12]Neil Sarnak, R. Bernstein, Vincent Kruskal, "Creation and Maintenence of Multiple Versions", Proceedings of the Workshop on Software Version and Configuration Control, (Berichte des German Chapter of the ACM Vol. 30), January 1988, Grassau, Germany

[13]Stanley Zdonik, "Version Management in Object-Oriented Database", Proceedings

of an International Workshop on Advanced Programming Environments, (Springer-Verlag Vol. 244), June 1986, Trondheim, Norway
[14]-, “Operation Specifications for the CM-General Operation Set, Case Communiqué, September 1992
[15]-, Portable Common Tool Environment (PCTE), Standard ECMA-149, European Computer Manufacturers Association, December 1990