Chapter 8

Versioning of Technical Documents - design and implementation

Hans Weigand, Robert Meersman

8.1. Introduction

Technical documentation differs from normal documentation as created by ordinary text processing systems in the following areas:

Technical documents are often very large in size. Documents exceeding 1000 pages are no exception.
Technical documents are created by a group of authors often working concurrently.
Technical documents that describe a product have a long lifetime, which follows the progressive development of the product described. This development has to be supported by multiple versions of one document.
Technical documents have to incorporate information from other sources (other documents, but also paper and remote computer systems).
Since technical documents are generally complex, there is a need for management support for such documents.
Since technical products are often designed as a series, e.g. configurations which differ in details only, the various documents describing the products also exist as close variants of each other.

The Sprite Document Management System is an integrated system for the production and maintenance of technical documents. An important feature is the version and configuration mechanism. In this chapter, we describe the basic mechanism and how it is implemented in the Sprite MMD database.

The chapter is organized as follows. In section 2, we consider the basic requirements of version control, and in section 3 and 4 we describe the solution of Sprite and compare it with some other systems and proposals. Finally, in section 5, we suggest some possible extensions, particularly in the direction of Cooperative Working Support.

8.2. Version model of Sprite

As stated in the introduction, a technical document management system (TDMS) should support the use of versions. Two types of versions must be distinguished: *historical versions* and *configurations*.

Historical versions correspond with either the derivational or logical history of a document. Technical documents are developed on a project basis, in consecutive steps, over a long period of time. The TDMS should support the derivation of new versions of documents.

Configurations, or alternatives, represent slightly different versions of a certain document These configurations may correspond with different versions of the product described. For example, a keyboard manual may exist in an 220V and a 110V configuration. It is also possible that the versions differ in style, or in language.

The MINOS system (Christodoulakis et al, 1986) and the EXODUS system (Carey et al, 1986) also support the derivation of new versions of a document, but no distinction is made between historical versions and configurations. However, the combination of these two dimensions is not trivial, as we will see below, and can easily lead to chaos if no special organizational measures are taken. Distinguishing the two has the advantage of a clear conceptual picture. At several occasions, it also increases the level of data sharing, that is, higher efficiency in storage use. A distinction between historical versions and alternatives is made in several CAD/CAM systems and in the object-oriented database systems like ONTOS (Andrews, 1989).

Version control in a CAD environment is addressed in (Chou & Kim, 1986). The engineering environment requires a distinction between three database spaces: private, public and project. The private database is manipulated by one designer. When he is ready he checks it into the public database so that other people can read it. When a new version must be created, the user checks data out of the public database again. It often occurs that a group of users (project) works together on one complex object. In that case, the designer first checks his data into the project database which is accessible to project members only. When all the pieces are collected the project leader checks the data into the public database. Hence three different kinds of versions are distinguished: *released* (in the public database), *working* (in the project and private database) and *transient* (in the private database). Each type has different authorizations. For example, a working version can be checked into the public database only by a user with project management privileges.

Version control in a CAD environment must also take the aggregation hierarchy into account. Design objects are typically made by the configuration of more primitive objects that are relatively independent and can have versions of their own. They can be shared between several higher-level objects. According to Chou and Kim, a distributed CAD environment must also support notification techniques. Two types are distinguished: *message-based* and *flag-based*. In the message-based approach, the system sends messages to notify human users of potentially affected versions, either immediately or at some later time. In the flag-based approach, the system simply updates data structures so that affected users will become aware of changes in a version only when they explicitly access the version. A rather simple notification technique is called *version percolation* (Atwood, 1985). This means that when a new version is derived from an old version of an object, the system automatically generates new versions of objects that directly or indirectly reference the old version. This technique is not favoured by Chou and Kim, because it easily leads to an explosion of versions, but has the advantage of reducing some of the complexity.

For completeness, we also mention a few different purposes of versions, such as for concurrency control (shadow files), recovery, enhancing performance in distributed systems (replication), and implementing update-free databases. Such versions are usually not visible to end-users. In chapter 9, the multi-authoring mechanism of Sprite is discussed, which includes the use of shadow checkpoints. In the present chapter, we restrict ourselves to versions that support the documentation design process and therefore are an important part of the user model.

Technical documents exist over a long period of time and are developed in several steps, much like CAD design objects. Moreover, just as the products they describe, the manuals can exist in several alternatives. Hence the requirements of Sprite are very similar to the ones mentioned by Chou and Kim.

8.3. Historical versions

Sprite allows the user to keep historical versions of a certain document, alternatively called checkpoints. This mechanism has several functions:

recovery from mistakes. In the course of development, the author may want to start again from some previous version. In that case, he can use the browser to locate that old version and start editing again from there.

data sharing. When the author wants to rewrite some existing document, he need not copy its entire content. Deriving a new version from it is sufficient and guarantees efficient content sharing.

project management. Technical documents typically have more than one edition. The version mechanism maintains the logical relationship between the consecutive editions.

The historical version mechanism in SPRITE is implemented by the following methods (where oid stands for object identifier):

NEWJZHECKPOINT(oid): oid
FREEZE(oid)
CHECKIN(oid)
CHECKOUT(oid)
DELETE(oid)
ARCHIVE(oid)

NEW_CHECKPOINT takes an object identifier as argument and returns the object identifier of a new object. This new object (document) is initially the same as the old object; attribute values are copied and the content is shared. When the user starts editing the document, the affected components in the logical structure are automatically replaced by new versions. Replacing a component by a new version triggers the replacement of the parent component by a new version, possibly up to the root component. The updates are performed on the new versions. In this way, the data sharing is maximal; this is essentially the method used in the EXODUS system (Carey et al, 1986). Figure 8.1 shows two related versions after an update on the latest one.

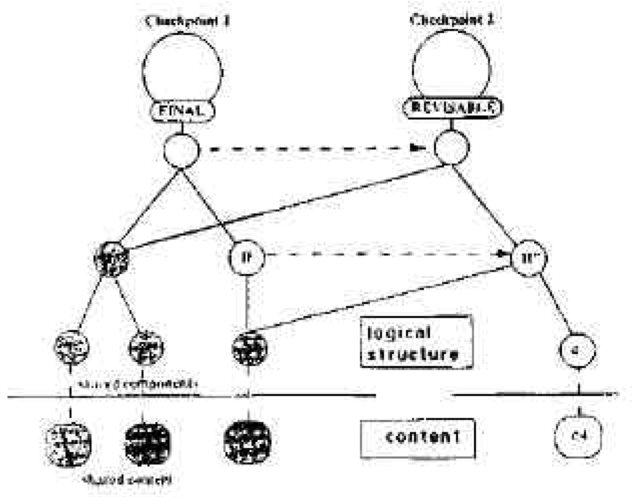


Fig 8.1 Sharing logical structure and content between versions

The new checkpoint is connected to the old checkpoint by means of a previous/next relationship. In this way, it is easy to go back in the derivation history of a document.

Note that more than one new checkpoint can be derived from an existing checkpoint. The checkpoints therefore form a *tree*.

The effect of FREEZE is that the object is no longer revisable: any attempts to update its attributes or content are blocked. However, frozen documents can be displayed, printed and used to derive new (revisable) checkpoints. In Sprite, a FREEZE (of the old document) is triggered by NEW.CHECKPOINT, so that all internal nodes of the checkpoint tree are always frozen, and hence immutable. FREEZE can also be done directly by the user.

Not all checkpoints are equally important in the project history. Usually, authors will work on a document for some time, and then decide to turn the last checkpoint into an *edition*. An edition is defined as a special checkpoint with a certain public relevance; it may be the checkpoint that is actually printed and shipped to the clients. The operation CHECKIN is used to promote a checkpoint to the status of an edition. Editions have edition numbers, so that it is easy to go through all editions of a document. Editions are always frozen, and, even more strictly, cannot be used to derive new checkpoints. An explicit CHECKOUT command is needed beforehand. This mechanism works in the same way *as* the distribution in different databases (private and public) in CAD systems.

DELETE just deletes a checkpoint. Any checkpoint can be deleted, unless it has been archived (by means of the ARCHIVE command).

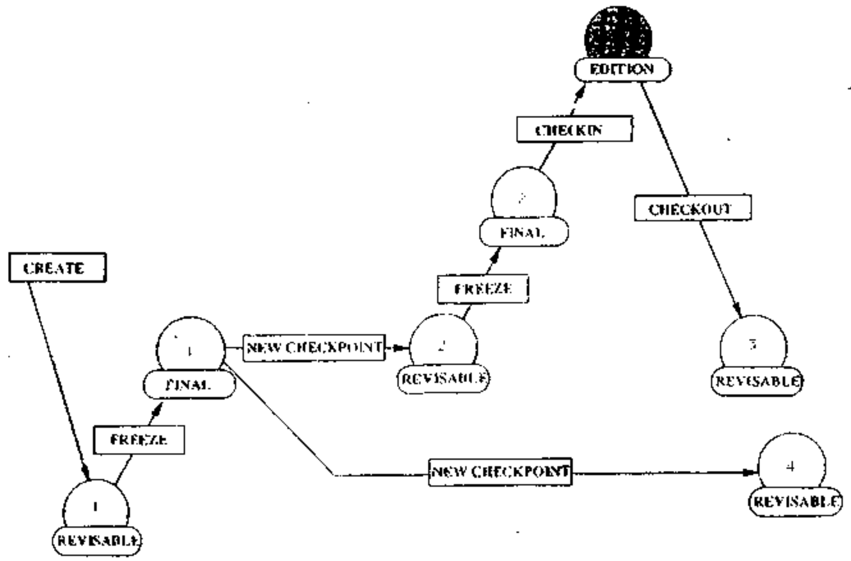


Fig 8.2 A sample document history

Figure 8.2 gives an example of a document history. Note that edition numbers are independent from checkpoint numbers. It is possible to filter out all non-editions and view the edition tree only.

8.4. Configurations

Sprite supports the creation and retrieval of documents describing different configurations of a product. Two perspectives can be distinguished, the perspective **of** the authors, called the *aggregation* view, and the perspective of the readers or reviewers, called the *specialization* view.

Suppose we have a manual of some electronic product which was available for 110V, 220V and 360V, and suppose that for each of the alternatives a separate manual must be created. Then the three manuals will share the bulk of their content and differ on parts. The common part can be considered as a generic manual, and the 110V manual for example is a special case of this generic manual. It inherits all its contents and properties but adds something more specific. In turn, if we consider differences in plugs, even more specific cases can be considered, for example the 220V-2-Pins-Plug manual. A manual "user" is interested in some or all of these special cases. However, for the authors, or creators, of the documentation, such a full manual consists of several parts: the common stuff, the 220V stuff, the 2 pins plug stuff. In Sprite, these components are called *building block documents*. Each building block is an identifiable object and can be manipulated separately. It is only when a manual must be released, that the different building blocks must be put together into one configuration. Fig. 8.3 shows two possible configuration trees.

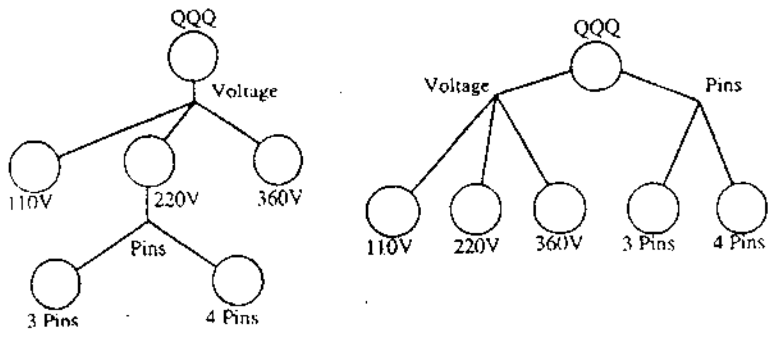


Fig. 8.3, Two possible configuration trees

In Fig. 8.3a we see that it is possible to group together related alternatives together even if they occur on the same level. Such a group is called a *configuration dimension.* In general, a Sprite document can have several configuration dimensions. For example, it is possible that the manual has a configuration dimension "voltage" and a configuration dimension "plug". The number of full manuals that can be produced then equals the number of voltage values times the number of plug values. However, when plug type is only applicable for 220V manuals, it is specified as a subconfiguration of 220V. In that case, the number of full manuals is equal to the sum of the two value set cardinalities minus 1. In general, the configurations make up a tree. (For tractability reasons, a lattice structure is avoided.) This tree must be defined before an author can create and edit a certain building block, although it can be modified in the course of time. The Document Organizer application allows browsing through the configuration tree. In this way, there is always a well-defined organization. The system maintains consistency when a configuration of building blocks is produced, since not all combinations are allowed.

Document material occurring in one building block document must be integrated with the material from other building blocks. It must be specified of course how this is to be achieved. This is done by means *of placeholder* components. Suppose we have a building block containing the common stuff and one containing the 220V stuff. We call the former the super building block and the latter the sub building block. The super building block is itself a complex object consisting of chapters, sections etc. At those places where material of the sub building block has to be inserted, the user must set a placeholder. Each placeholder is marked with a configuration dimension, for example, voltage. In each of the voltage sub building blocks (110V, 220V, 360V) there is one corresponding "filling-in component". Placeholders and filling-in components are connected internally by pointers (Fig. 8.4).

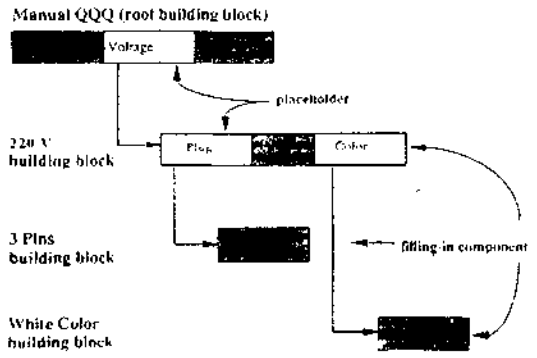


Fig 8.4 Building Blocks and placeholders

The configuration mechanism in Sprite is implemented essentially by the following methods:

CREATE_VERSION_CLUSTER
(UN)MARK_CONHGURATION_DOCUMENT
INSERT_CONFIGURATION_DOCUMENT (REMOVE..)
(UN)MARK_CONFIGURATION_COMPONENT
COMPOSE_DOCUMENT INSERT_BB_COMPOSED

Not all documents in the Sprite system need the complexity of configuration management Therefore Sprite requires the explicit declaration of a configurable document, or want the promotion of a normal document into such a configurable document. The CREATE_VERSION_CLUSTER command is used for this purpose. It creates a configuration tree, initially empty (except for the "root").

The MARK_CONFIGURATION_DOCUMENT is used to mark configuration dimensions and can be applied to any building block in the configuration tree. To state the different values of a certain dimension, INSERT_CONFIGURATION_DOCUMENT is used. When UNMARKTNG a certain dimension, it is possible to select one value to be promoted to the higher more generic level. This means that the placeholders of the dimension in question are replaced by the filling-in components of the specific sub building block, the configuration dimension is removed from the configuration tree and the sub building blocks for the various values are deleted.

To indicate where configuration-dependent material has be to inserted in the super building block, the user must execute MARK_CONF_COMPONENT on an existing empty or filled component. The configuration dimension must be specified and must be one of the configuration dimensions of the building block. If the component was not empty, the operation implies that filling-in components are created with the same content in all applicable sub building blocks.

A full "composed" document is created by the COMPOSE_DOCUMENT method. It is also possible to add the building blocks one by one in the composed document by means of INSERT_BB_COMPOSED.

Building Block documents are identifiable document objects in the Sprite system with their own checkpoint history. If we consider all checkpoints of building blocks, the configuration tree becomes more complicated. We use the term "version cluster" for the whole set of related building block checkpoints. For example, the 220V building block may have three checkpoints and its super building block five. It is quite well possible that only the first checkpoint of 220V is compatible with the first two checkpoints of the super building block, while the second and third are compatible with the others. "Compatible" means here that for each filling-in component there exists one placeholder component. The compatibility relationship is explicitly recorded by the system, and can be browsed.

Notification.
Sprite uses a minimalistic notification technique for effects of updates on building blocks. There are two basic rules. (1) consistency between compatible building blocks is maintained automatically as long as both are revisable. For example, when a placeholder component in the super building block is removed, by UNMARK_CONF_COMPONENT, then the corresponding filling-in components in the revisable sub building blocks are removed as well. (2) a sub building block must always be compatible with at least one super building block (so that it can be printed properly). This means that when X is a final, non-revisable sub building block of Y and Y is modified in such a way that X is no longer compatible, there must exist some previous version of Y that is compatible with X. If not, it is necessary to make such a version first by means of NEW_CHECKPOINT(Y). The effect of the modification is then that X is no longer compatible with the new Y[1], but of course X is still compatible with the old Y (now frozen).

For the rest, Sprite uses a minimalistic policy. The user can make as many new versions of building blocks as he wants, and edit them, but Sprite will not generate new versions of reference objects as long as this is not necessitated by the two integrity rules (1) and (2). Besides configuration links, dynamic references to components or documents (where the actual reference is determined by dynamic binding) are possible in Sprite. No notification of changes is done for these references. What Sprite does support is that when an object with some dynamic reference is *frozen,* the referred object is frozen as well.

The notification technique described above is not used when the super and sub building blocks are of different editions, that is, when there has been a CHECKIN/CHECKOUT event in between. For example, when sub building block X is compatible with super building block Y, and Y is checked out again, crearingia new version Y' and a new version X', then modifications to Y' will never affect X although formally they are still compatible and could be combined in one composed document. This avoids long-distance dependencies among objects. The versions between a CHECKOUT and CHECKIN can be closely connected but do not have active references to older or later versions.

Working set.
The set of all versions of all building blocks of a certain configuration tree is called the version cluster. The version cluster is an identifiable object in the database and can be manipulated in the document space (for example, moved to some folder). The interface allows the user to open the version cluster and browse through the configuration tree and checkpoint trees, which can become rather complex. Usually the user is only interested in some particular building blocks, the ones that he is currently working on. For this purpose, Sprite allows the definition of *working sets,* which is a set of compatible building blocks. The working set acts as a default mechanism for references to some generic building block (checkpoint tree). Working

sets can be shared between members of a project group but each user can have at most one active working set per version cluster.

The Working Set is similar in meaning to the Environment in (Dittrich & Lorie, 1988) and the Context in (Chou&Kim, 1986), although Sprite does not allow generic references in the Working Set nor indirect references (via other Working Sets). However, our concept of version cluster is different from the logical version cluster concept used by Dittrich and Lorie, who do not make an a priori distinction between historical versions and configurations but allow the user to assign such semantics to the versions themselves by means of dusters. They propose a fairly general clustering mechanism. A similar approach is taken by (Klahold et al, 1986) using version graphs and partitions. Version graphs correspond to the version clusters of Dittrich and Lorie (although the latter, by allowing different levels of clustering, are even more general); a user can define as many version graphs as he wants but the price is that he must specify himself explicitly in which graph a certain newly created version has to be inserted and at which position. Similarly for partitions, which correspond to mathematical partitions. For example, a partition could be {revisable, final, archived) - the semantics must be defined by the user. Both mechanisms are admittedly more general, but also more low-level than the Sprite mechanism; they need an application environment to be defined on top of them in which more specific semantics can be instantiated.

8.5 Conclusions and directions for future research

Distinctive features of the Sprite version model as implemented are:
versioning is defined at the conceptual level;
a clean distinction is made between historical versions and configurations;
the version model is supported by an efficient storage mechanism;

For the rest, the versioning mechanism of Sprite is somewhat less general then many proposals in the CAD area. This was done on purpose. Sprite is not a programming environment but a system in which the objects are directly manipulated by an end-user. Hence the user model had to be easy to understand and teach.

The problem of versions has many aspects. For future research it appears promising to distinguish three different subproblems: (1) the versioned object problem, (2) the context problem and (3) the coordination problem.

The versioned object problem. ,
Many object-oriented database systems already provide the concept of "version hierarchy" (for the ORION system see e.g. Kim, Bertino and Garza, 1989). Each object in a version hierarchy is derived from another object in the hierarchy and has its own object-identifier. Information about the version hierarchy is often stored as part of the root object, called generic object. This mechanism, when supported by an

intelligent implementation technique, is sufficient for handling historical versions. However, no standard exists yet for configurations. Generalizing Sprite, we may suggest the following features to be added to the object-oriented framework:

a class can be said to be versionable and configurable. This means that instances can exist in different versions and different configurations respectively. For instances of versionable classes, the method NEW_VERSION is applicable that derives a new version of the object.

- for instances of configurable classes, the methods (UN)MARK_VARIANT (oid, dimension) and the methods INSERT_VARIANT/REMOVE_VARIANT(oid dimension_value) are applicable. They add (remove) configuration dimensions and configuration values to a certain object. The implementation of these methods can be based on Sprite, but we suggest that building blocks need not be visible to the user; only the "specialization" view is presented. This approach is only feasible when the query language is expressive enough to refer to any version and variant. This is a topic for future research.

The context problem.
The working set feature provided in Sprite allows users to select only relevant items. In general, information systems, including object-oriented ones, need some way to specify a context so that operations and retrieval can be simplified, for example a default version of a versioned object. A proposal for a context mechanism can be found, among others, in (Beech & Mahbod, 1988). More research is needed in order to understand the proper place of context in the object-oriented framework and explore the possibilities it may offer.

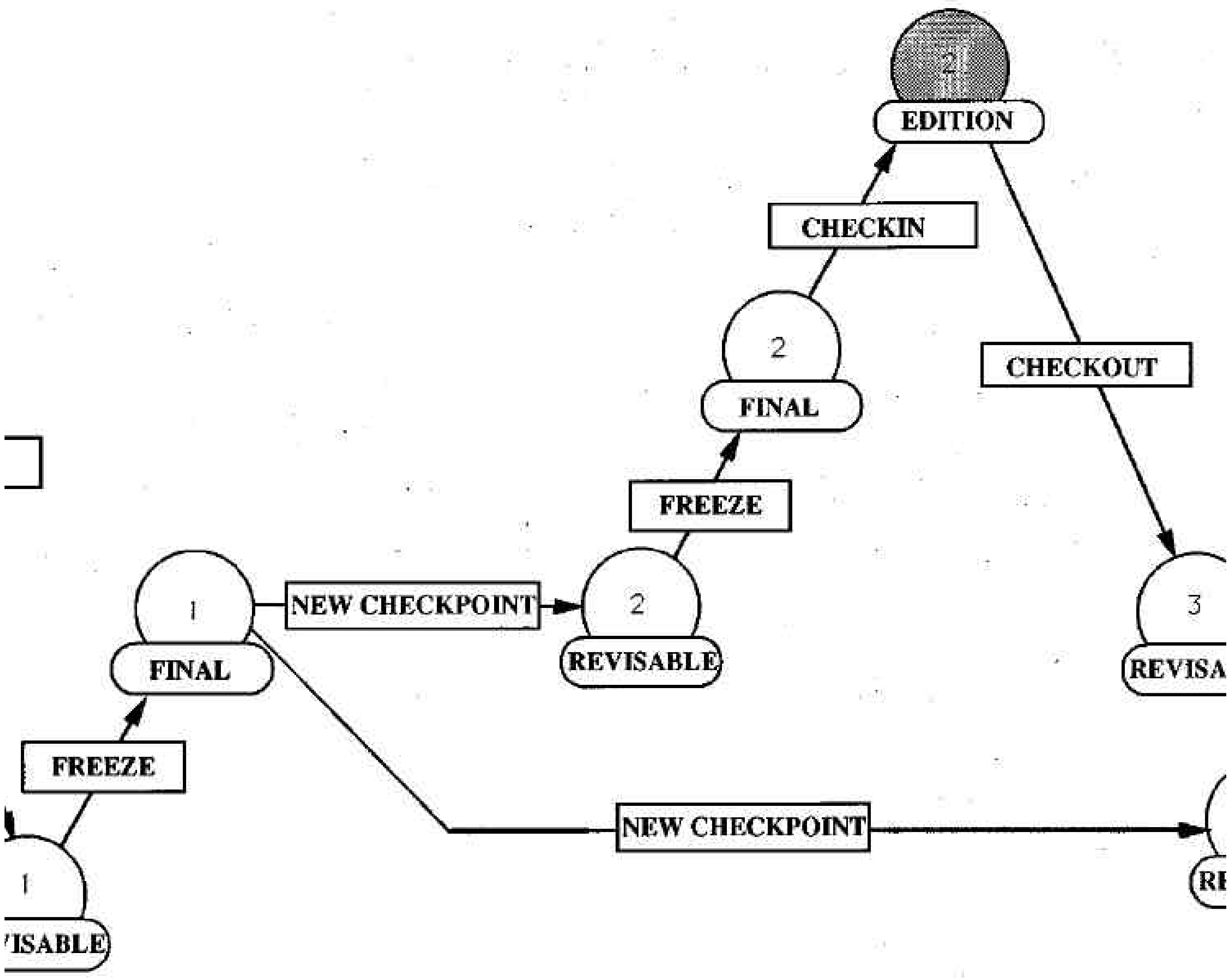
The coordination problem.
Although a basic versioning and variant mechanism for technical documents is necessary, we also think that the concept of version has not yet been fully exploited. As in CAD environments, in the technical documentation environment a version is not just a possible state of an object but has an important organizational meaning, as was already indicated by the discussion on the CHECKIN/CHECKOUT mechanism. Hence it might be better to say that a version does not represent so much a state of an object, but rather a state of the communication flow, or the *collaborative working* procedure. For example, this collaboration may include authors, product specialists, managers, project leaders and quality assurance personnel. Recognizing this, the CHECKIN/CHECKOUT operations may in fact be only the more significant steps, but not by far the only ones. In the above example, a request for reviewing, or for approval, and the granting of these, might be significant as well. We feel that besides the basic versioning mechanism the system should also provide means to specify such collaborative working procedures; some basic operations, such as requesting and granting approval can be foreseen immediately, but in a more general - and practical -

context it should be possible for the users to set up and modify such procedures for their own particular organizational environment.

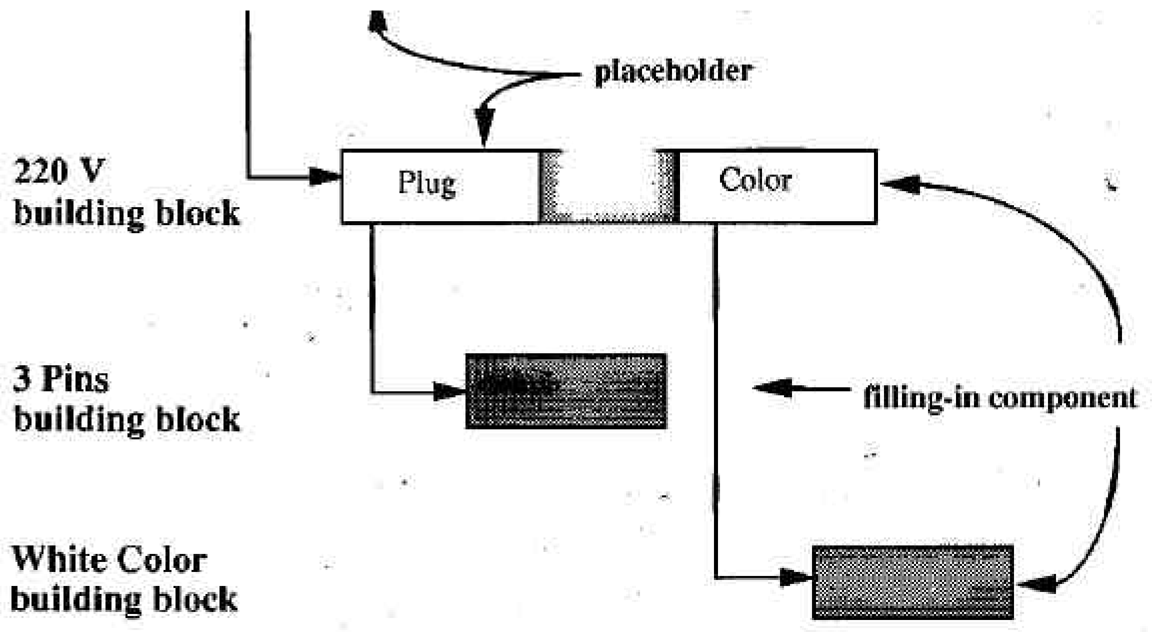
Splitting up the version problem in the three aspects mentioned above may lead to a more modular organization which enhances-reusability and comprehensiveness.

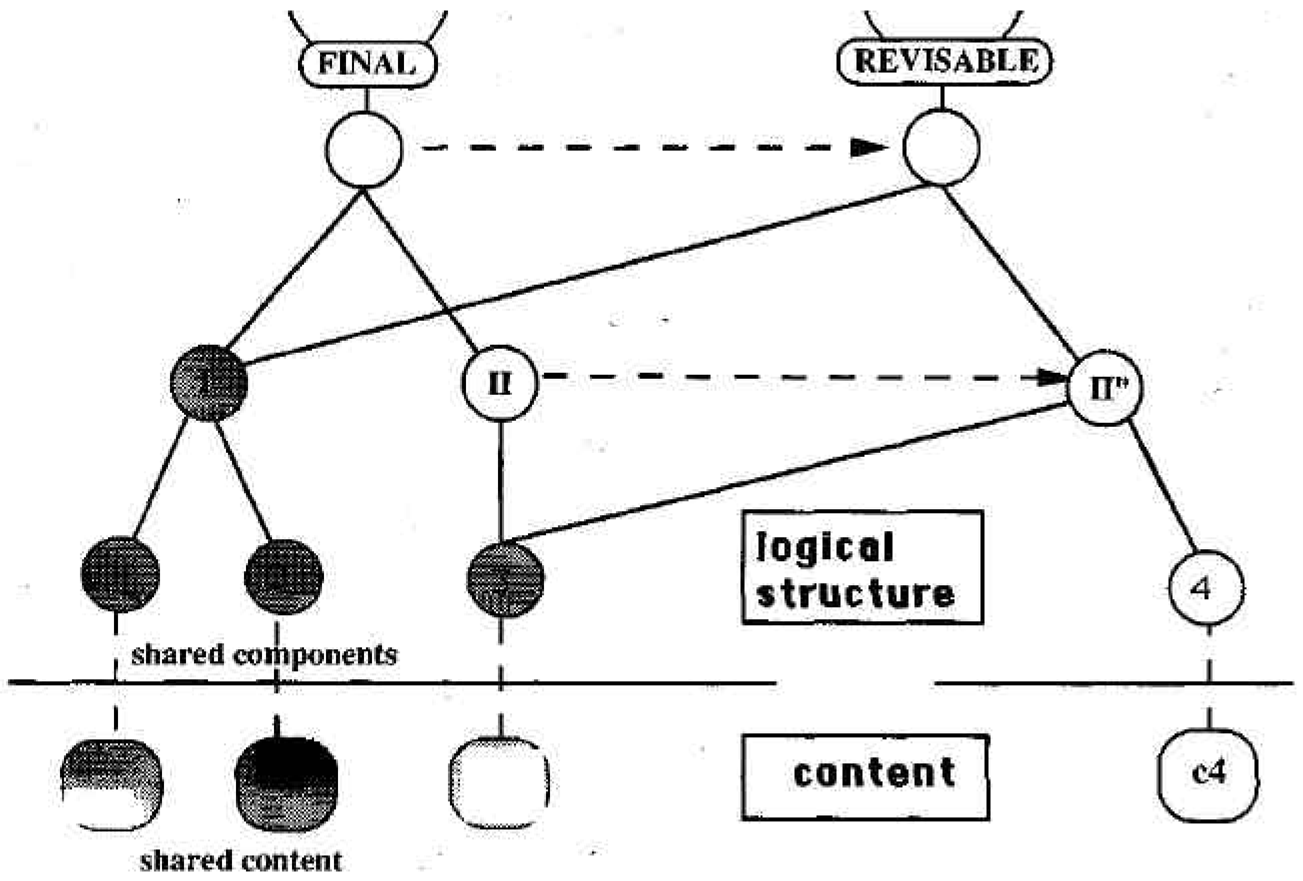
References

Andrews, T., C. Harris, K. Sinkel, 1989.
The ONTOS Object Database. Ontologic.
At wood, T., 1985.
An Object-Oriented DBMS for Design Support Applications.
Proc. IEEE COMPINT 85, Montreal, pp.299-307.
Beech, D., B. Mahbod, 1988.
Generalized Version Control in an Object-oriented database.
Proc. of the IEEE Data Engineering Conference, Los Angeles, pp. 14-22.
Carey, M. et al, 1986.
Object and File Management in the EXODUS Extensible Database system.
Proc. Int. Conf on VLDB, Kyoto.
Caruso, M., E. Sciore, 1988.
Contexts and MetaMessages in Object-Oriented Programming Language Design.
Proc. ACM SIGMOD.
Chou, H., W. Kim, 1986.
A unifying framework for version control in a CAD environment.
Proc. 12th VLDB Conference, Kyoto, pp.336-346.
Christodoulakis, S. et al, 1986.
Multimedia Document Presentation, Information Extraction, and Document Formation in MINOS: A Model and a System.
ACM Trans. on Office Information Systems, 4(4).
Dittrich, K, R. Lone, 1988.
Version support for engineering database systems.
IEEE Trans. Software Engineering, 14, 4, pp.429-437.
Katz, R.H., 1990.
Toward a Unified Framework for Version Modelling in Engineering Databases.
ACM Computer Surveys 22,4, pp.375-408.
Kim, W., E.Bertiono, J.F. Garza, 1989.
Composite Objects revisited.
Proc. ACM SIGMOD, pp. 337-347.
Klahold, P., G. Schlageter, W.Wilkes, 1986.
A general model for version management in databases.
Proc. Int. Conf on VLDB, Kyoto.



VISABLE
1
FREEZE
FINAL
1
NEW CHECKPOINT
2
REVISABLE
FREEZE
FINAL
2
CHECKIN
EDITION
2
CHECKOUT
3
REVISA
NEW CHECKPOINT
RE


placeholder
220 V
building block
Plug
Color
3 Pins
building block
filling-in component
White Color
building block


FINAL
REVISABLE
II
II"
logical structure
4
shared components
content
c4
shared content