Linköping Studies in Science and Technology
Thesis No 328

Aspects of Version Management of Composite Objects

by

Patrick Lambrix

Submitted to the School of Engineering at Linköping University in partial fulfillment of the requirements for the degree of Licentiate of Engineering.

Department of Computer and Information Science
S-581 83 Linköping, Sweden

Linköping 1992

Contents

1 Introduction

1.1 Thesis Overview

Object oriented databases have become increasingly popular during recent years and are seen as providing a more appropriate representational model for applications such as engineering, CAD and office systems, than the traditional relational database model. An important aspect of the object oriented model for these systems is the ability to build up composite objects from object parts (e.g. [An*90], [NiT88]). This then allows modularity in the representation of objects and reuse of parts where appropriate. It is also generally accepted that object-oriented databases for such applications should be able to handle temporal data ([SnA86]). However little if any theoretical work has been done on temporal behavior of composite objects, and only relatively few systems actually attempt to incorporate both historical information and composite objects in a multi-user environment. In the other sections of this chapter, we address composite objects and historical information in general, while in the next chapter we describe these issues and the data model in the LINCKS[1] ([Pad86], [Pad88]) system. In section 2.3 we propose two different kinds of historical information of objects which we call *temporal* and *edit* history. The edit history gives information about what versions[2] are used to create other versions. This edit history is usually available in other systems in some form. The temporal history gives an order on the creation time of objects. This kind of history is usually not available in other systems.

Even the most basic kind of historical information, snapshot information regarding the database (or in our case objects) at a past point in time ([SnA86]), is problematic when we deal with composite objects. Two important problems arise immediately. First, it is unclear when a change in a part of a composite object should be propagated into a change in the composite object itself[3]. Secondly, maintaining information regarding changes in atomic objects (or parts

[1] LINCKS (Linköping INtelligent Communication of Knowledge System) is an experimental object oriented database system developed at the Intelligent Information Systems Lab of Linköping University. The main purpose of LINCKS is to be used as a testbed for ongoing research.

[2] Every object consists of *versions* where each change to the object gives rise to the creation of a new version. A version is thus the (static) appearance of an object over a certain time. The historical information can then be seen as a partial order over the set of versions. [Katz90] gives a survey of the different version data models proposed up till 1989.

[3] Katz [Kat90] considers this to be a difficult but challenging problem.

of composite objects) is not in itself sufficient to enable recreation of snapshots of the composite object at past times.

One of the questions around which this work is centered is "when do changes in a component object propagate to the composite object in which it is contained?" We have identified two different groups of composite objects which behave differently with respect to this question : one which changes when one of the composing objects is changed, or components are added or deleted (*strong connection*), and one which changes only when composing objects are added or deleted, but not when they are changed (*weak connection*). These two different kinds of composite objects can be exemplified by a document on the one hand and a folder or cabinet on the other hand. The document contains sections and subsections while the cabinet or folder contains, say, a number of personnel files. When a section of a document changes it is natural to consider the document itself to have changed, whereas a change within a personnel file does not naturally induce a similar change in the cabinet. In chapter 4 we define formally the notions of *strong* and *weak connection*, which are more general then the composition notions. We also sidestep for a moment to have a look at (non-)transitivity properties of those connections. Then we derive synchronization constraints which can be applied to composite objects in a database setting, to ensure that they exhibit the desired behavior. These constraints form a basis for providing support to ensure that composite objects are maintained in a consistent form as the database changes over time. As a formal framework for our theoretical results we use LITE ([Rön90], [Rön92]). We present LITE in chapter 3.

Further, in chapter 5 we discuss the problem of maintaining historical information for compositions using the historical information of its components. This problem arises from sharability and dynamic linking. Indeed, there is some recognition in the literature that there are problems associated with dynamic linking (a mechanism commonly used for linking to a conceptual object rather than to a particular version of that object, allowing the version to be determined dynamically) and propagation of change from components to composition. Systems such as IRIS [BeM88] and ORION [KBG89], for instance, support some sort of propagation of change to the composite object, but at the cost of making dynamic linking useless. This is undesirable as dynamic linking is an important mechanism for managing computational complexity, allowing links to be instantiated on demand, rather than requiring updates whenever a change is made.

We discuss a mechanism, the *composite object representation*, which allows us to maintain historical information of compositions (see chapter 6). The mechanism is based on the theoretical results and therefore ensures the maintaining of the two different types of compositions. Further, it also gives a solution to the problem of sharing and dynamic linking. We also describe how the LINCKS system copes with composite objects and describe an implementation of the composite object representations in LINCKS.

Two major issues concerning a versioning mechanism for composite objects are *scope of propagation* and *path ambiguity* ([KaC90]). We discuss briefly how our mechanism copes with those problems in chapter 7.

In chapter 8 we discuss several systems and proposals for handling temporal information of objects. We include both version management of compositions and of complete databases. We also address related concepts to our strong and weak connection.
We conclude the thesis with a conclusion and topics for further work in chapter 9.

In an appendix we give the structure charts of the implemented functionality of the composite object representation mechanism in LINCKS.

The original contributions of this thesis are the connection notions (chapter 4) and the mechanism for maintaining historical information of composite objects (chapters 5, 6, and 7 except 6.2.1 and 6.2.3).

1.2 Object Oriented Databases

Although object oriented databases have become increasingly popular during recent years, there is no, as there is for relational databases [Cod70], clear specification of what an object oriented database is. It is not the case that no complete object-oriented data model exists, but rather that there is no agreement yet. Another characteristic of the field is the lack of a strong theoretical framework. Finally, a lot of experimental work is underway. Some of the findings of a group of object oriented database people are given in the 'object-oriented database system manifesto' ([At*89]). They propose a working definition where the characteristics of an object-oriented database system are divided into three groups : mandatory, optional and open characteristics. The mandatory characteristics are complex objects (and as a subclass composite objects), object identity, sharing of objects, encapsulation of objects, types or classes, inheritance, computational completeness, extensibility, persistent data, secondary storage management, concurrent users, recovery from software and hardware failures, and a query facility. The optional characteristics are multiple inheritance, type checking, distribution, long and nested transactions and versions. The authors did not reach agreement regarding whether views, integrity constraints and schema evolution should be optional or mandatory.

The authors of [St*90] describe the proposal in [At*89] however as too narrowly focused on object management issues. They want to address the larger issue of providing solutions that support data, rule and object management with a complete toolkit including integration of the database management system and its query language in a multi-lingual environment.

Bertino and Lorenzo, in their overview article [BeM91], see as the basic concepts of an object-oriented data model, the notions of objects, having attributes and methods[4], complex objects, classes and inheritance. An important relationship that could be imposed on the complex objects is the part-of relation, forming composite objects. Further a query language should be available to be able to retrieve data by specification of the data contents. Effective support of an object-oriented data model requires specific mechanisms. These are a versioning mechanism, a concurrency control and transaction mechanism, an authorization mechanism and a schema evolution mechanism.

In this thesis we concentrate on two of these notions : the versioning mechanism and the composite objects. We investigate the behavior of composite objects in a system with temporal information.

1.3 Composite Objects

Complex objects are objects for which the properties defining the object can have values of a complex type (ie. different from the basic types boolean, integer and string). For instance a flag has as one of the defining properties the attribute colors. The value of 'colors' is then usually a set of colors. *Composite* objects are complex objects that are conceptualized as a hierarchy of objects such that the hierarchical links represent the part-of relation[5]. We refer to

[4]In our system, LINCKS [Pad88], we do not consider methods to be a necessary part of the data model.

[5]Similar terminology is also used in for instance [BeM91].

both the root object and to the whole hierarchical grouping as a composite object.

However 'part-of' is not such a simple concept. The part-of relation is a denotation for a set of different relations all of which have the composition-component intuition in common but which differ from each other in other ways. In work on version management of composite objects in the ORION system ([KBG89]) the authors recognize two properties of the part-of relation. First a component can be sharable between different compositions or not. (A section can be part of several documents. However the motor of a car belongs to only one car at any particular time.) Secondly one might want to say that an object can only exist when there exists a composition for which it is a component. (One might for instance want to speak about the volume of an ice cube as a component, but only if the cube itself exists.)
Also in the cognitive science area work has been done in exploring different classes of 'part-of' relations. In [ILE88] the authors studied the way people usually speak about part-of. They divide the spectrum of the part-of relation into four different types. The functional component type describes the relation between my bike and its wheels. The part contributes to the composition, not only as a structural unit, but also as an essential unit to the purposeful activity of the composition. The segmented whole type is illustrated by the pieces of a pie. This part-of conception implies some kind of removability of the components from the composition and also the composition should exist before the components do. The two other types are the relations which are exemplified by the membership relation between a collection and its members and the relationship between a set and its subsets.
Another taxonomy of part-whole relations is provided in [WCH87]. The authors divide the part-of relations into six classes[6] : the component-integral object relation (*pedal-bike*), the member-collection relation (*ship-fleet*), the portion-mass relation (*slice-pie*), the stuff-object relation (*steel-car*), the feature-activity relation (*paying-shopping*) and the place-area relation (*Brussels-Belgium*). The different classes of part-of relation differ from each other with respect to three properties : functionality of the component in the composition, separability of the components from the composition and whether the composition and component are of the same 'kind'. The authors do not include the class inclusion relations in the part-of relation. Interesting to note is that they then use this taxonomy to investigate the transitivity of part-of. They conclude that

[6]The taxonomy is extended in [Lan91] to include the relations 'pertains-to' and 'is-alienable-to'.

transitivity in general is lost when we mix different classes of part-of relations.

In this work we are mainly interested in questions about change over time. A natural question to ask is then whether a composition changes when a component (or part) changes. In an office information environment for instance the answer to this question varies. For instance when a section in a document changes, goes in or comes out of the document then also the document has changed. We say then that the section is in a *strong connection* with the document. On the other hand when a folder in a cabinet changes, the cabinet is not required to change. It is only when folders are put in or taken out of a cabinet that the cabinet changes. We say that the folder is in a *weak connection* with the cabinet. Formal definitions are given in chapter 4.

1.4 Historical Information in Databases

Conventional databases can be viewed as snapshot databases in that they represent the state of an enterprise at one particular time, generally 'now'. As a database changes, out-of-date information, representing past states, is discarded. Many different authors have advocated the importance of historical information in a database or information system environment. It is not hard to see that as we seek to add greater functionality to information systems it is often necessary to have access to historical information. For example a query such as "what was Sally's salary 12 months ago" can only be answered by a reference to historical information.

There are three orthogonal kinds of time that a data model may support : *transaction time*, *valid time*, and *user defined time* ([SnA86]). Transaction time concerns the time information was stored in the database. Valid time concerns modeling a time-varying reality. User defined time is necessary when additional temporal information, not handled by transaction or valid time, is stored in the database[7].

There has been considerable work done on temporal information in relational databases[8]. In [McS91] the authors define 26 criteria to evaluate temporal relational algebras. Temporal relational algebras are procedural query languages for relational databases (relational algebras) that can handle time-varying

[7] A database handling transaction time is called a *rollback database*. A database handling valid time is called a *historical database*. When both transaction and valid time are handled the database is a *temporal database* ([SnA86]).

[8] see for instance overview in [Sno90]

data. The authors show however that the criteria are mutually unsatisfiable. They also show that in a sense the possibilities to produce temporal relational algebras according to the choices with respect to conflicting criteria are covered by the different approaches. The evaluated systems in [McS91] all handle valid time, but only a few of them also cope with transaction time.

This work on temporal information in relational databases however, is not directly transferable to object-oriented databases, which have an entirely different structure and representation. A primary concern of object-oriented databases is version management. Version management, together with schema evolution are ways to model transaction time. A survey of work on object-oriented systems which support some handling of temporal data is reported in [Kat90]. Snodgrass ([Sno90]) sees as some further research directions for the management of transaction time the issue of integrating schema evolution and versioning, and the issue of versioning of complex objects. In this work we concentrate on the latter issue and address very briefly the former.

2 The LINCKS System

LINCKS ([Pad86], [Pad88]) is an object oriented information system where different users work on a shared database. Each user has his own workspace where he can check objects out from the database. All modification of the objects occurs in the workspace. Later on the user can then check the objects back into the database.

In this chapter we describe the LINCKS data model. We also address the way compositions are treated in LINCKS. Further we describe the history mechanism for atomic objects. The LINCKS system supports two kinds of historical information for objects, temporal history and edit history. It also supports a history for transactions, the *command history* ([Hal92]), both with respect to a particular workspace and with respect to the shared database, but a description of the command history is beyond the scope of this work.

2.1 Objects in the LINCKS System

An *atomic object* in LINCKS (see figure 1) is a complex object represented by two kinds of nodes ([Åbe89]) : *versions* and a *history structure* node.

A node has three distinct parts ([Pad88], [Åbe89]). These are :

- the *image*, which is uninterpreted information (usually textual or graphic),

- a number of *attributes* with values, which describe the properties of the object which are not related to other objects,

- and a number of *links* which describe the relationships of the object to other objects.

Nodes are identified by unique *node identification numbers*. An object is then identified by the node identification number of its history structure node. A particular version of an object is identified by a pair of node identification numbers. The first number is the node identification number of the history structure node of the object where it belongs to and the second node identification number is the node identification number of the actual version node.

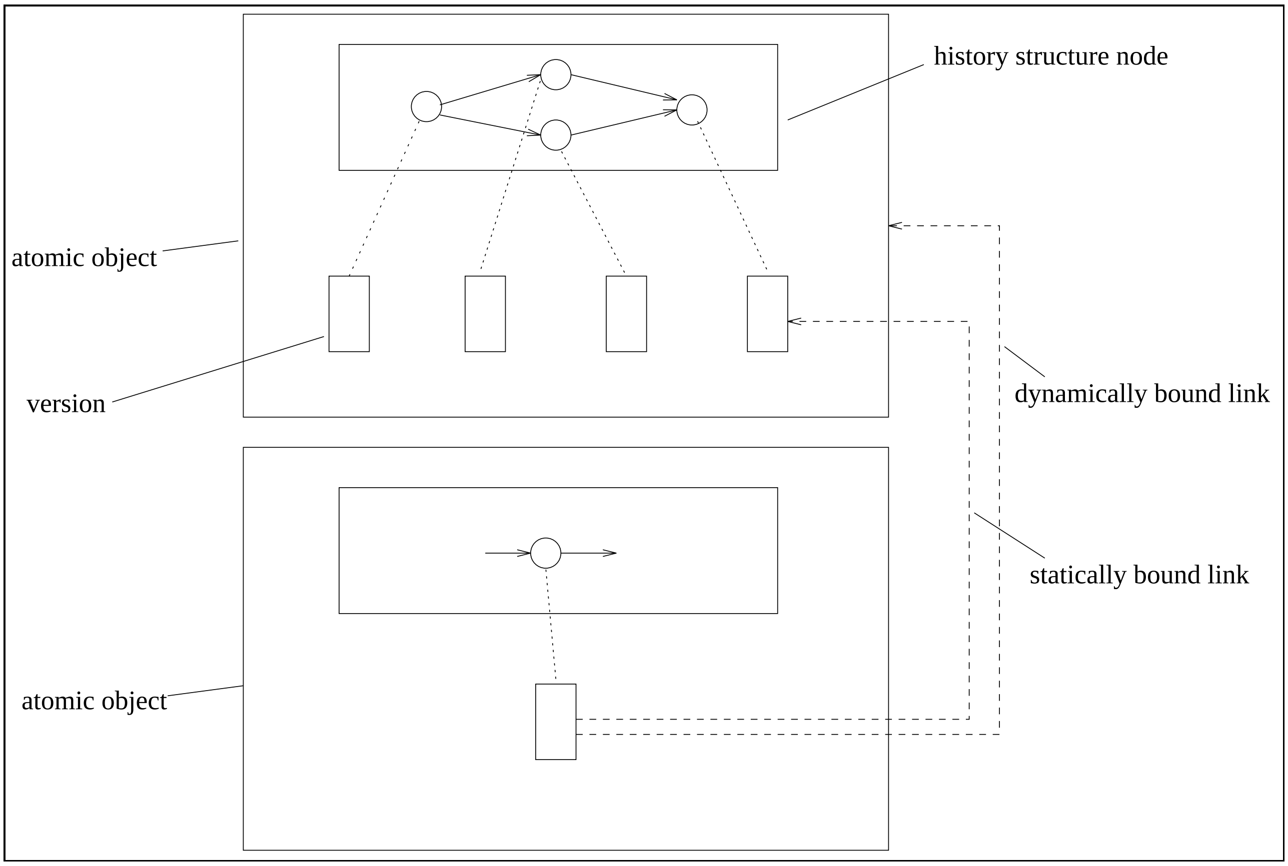


Figure 1: Atomic object in LINCKS

An object can have different links with the same name[9] to different objects. For instance, a piece of text can have different footnotes. The different footnotes are all in the footnote-relation with respect to the piece of text. On the other hand an object can have only one value for a particular attribute at a particular time. For instance at any particular time an object has just one creator.

Versions have a status. *Transient* versions are versions which have been created by a user in a workspace, but which have not yet been stored into the database. When an explicit store-command occurs, the version is promoted

[9]Attribute names and link names are actually two dimensional. They consist of a *group tag* and a *field tag*.

to a *working* version[10]. A working version cannot be changed anymore. This means that whenever one wants to change something to the working version, a new transient version is created.

2.2 Composite Objects in the LINCKS System

In LINCKS the part-of relation can be represented using attributes and links. For instance the part-of relation between a document and its title is represented using the attributes while the part-of relation between a document and its sections is typically represented using links.

Important in our model is that we allow for sharing of objects among different compositions. For instance a section can belong to different documents. Sharing is important to be able to maintain a consistent database. When sharing is not allowed, then for the section in different documents, we would have to maintain a copy of the section object for each document for which the section is a part. An update to the section has then only effect to the particular document that links to the particular copy of the section where the update occurred. To have the change also occur in the other documents, we would have to find all the copies of the section, and update them as well. When sharing of objects is allowed, several documents can contain the same section. In systems supporting version management we would like to go even a step further. When several compositions contain the same component, then we want to have a certain flexibility as to which version of a component is contained in which version of a composition. A user might want a particular version of a component to be part of a particular version of a composition. The user might also want to say that when a new version of a component is created, that this new version should be part of the composition. For this reason several systems (e.g. [Ba*87], [BeM88]), among which LINCKS ([Pad88]), introduce the notion of *dynamically bound links*. The dynamically bound link is a link between a version of an object and another object. The binding of the target object to a particular version of that object occurs on demand. In this way it is not necessary to introduce new links every time a component changes. In contrast to dynamically bound links one can also have *statically bound links* which link directly to a particular version of an object and do not require any further resolution.

In figure 2 we see that the first version of the composition root has a statically

[10] We use the ORION terminology ([Ba*87]).

bound link to the first version of the second component. This means that the first version of the composition root *always* contains the first version of the second component. The other links are dynamically bound links and the resolution is therefore time dependent. In the situation at hand the dynamically bound links refer to the second version of the first component and the last version of the second component.

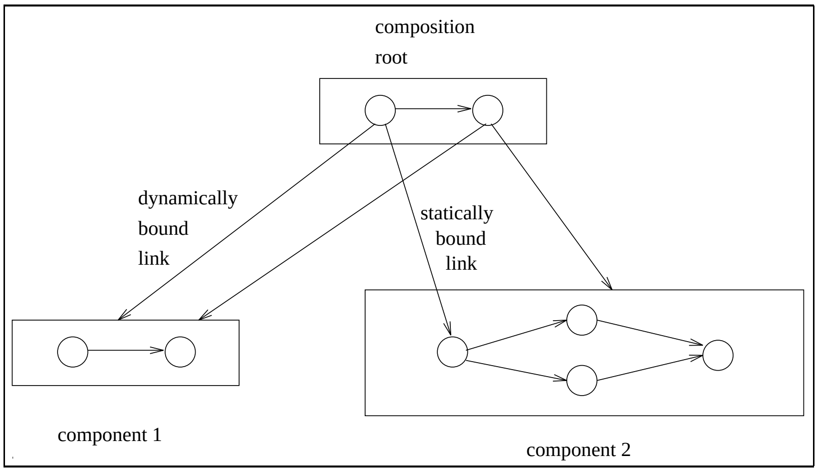


Figure 2: Dynamically and statically bound links

LINCKS supports both dynamically bound links and statically bound links (see figure 1). The version referenced by a dynamically bound link is time dependent and is in the LINCKS system by default the most recent version of that object. It is however possible for the user to interactively specify the desired version[11].

In our model we also allow the sharing of roots of compositions by several compositions. Objects can at the same time be components in compositions and root of a composition. A section for instance can be seen as a composition containing paragraphs, at the same time as it is contained in a document.

[11]In the current implementation the default version is arbitrarily chosen among the latest versions. However there is the possibility to query the user or add further reasoning. Supplementary to the current default mechanism we plan also to add the following possibilities for the default value : (i) a particular version, supplied by the user, (ii) the version which was edited latest by a particular user, (iii) the latest version in the development branch containing the version which was edited latest by a particular user.

Therefore we want the same sharing flexibility for composition roots[12].

2.3 Historical Information of Objects in LINCKS

It is important to have at least the following two different kinds of historical information about objects in an object oriented database system: information about what versions of an object were known at a particular time and information about which things are used to create something new. In this section we define the notions of *temporal history* and *edit history* which contain exactly this information.

2.3.1 Temporal History

The *temporal history* of an object is an order between the different versions of that object and reflects the order of creation time of the versions. In our model we allow that several users take out an object at the same logical[13] time and create a new version of that object. In that case we say that logically all the new versions come after the latest common version, but there is no order between the new versions. Consequently the temporal history is a partial order between the versions of an object. Versions that cannot be ordered following creation time are said to be *parallel*.

When parallelism is allowed for the temporal history, several users can work at the same time using and even editing the same piece of information in the shared database without disturbing each other. Although this results in the potential problem of parallel versions which at some time must be merged this may be preferable to locking, given the fact that transactions are relatively long. If desired, constraints can be placed at a higher level to enforce locking and a sequential history. This is then just a simplification of the temporal model which we take as a baseline.

[12] An objection to this flexibility might be that when we have two different compositions, we should also have two different composition roots. The reason however for sharing the root is the same as for sharing other objects. There is information in the root that is shared by the compositions. However we do not impose it on the user to use this flexibility. We only allow it.

[13] Two users take out an object at the same logical time if the object they access has the same state at both accesses, ie. there has been no change following the database involving that object between both accesses.

The model also allows that several users connect with the same database, take a copy, and work further disconnected from that database. (We say then that a user works in his own disconnected *workspace* and has a *session* until he reconnects with the database.) When the user reconnects with the database all new information in the workspace is integrated in the database. It is interesting to note that all versions of an object which are created in one session can be totally ordered with respect to each other in the temporal history.

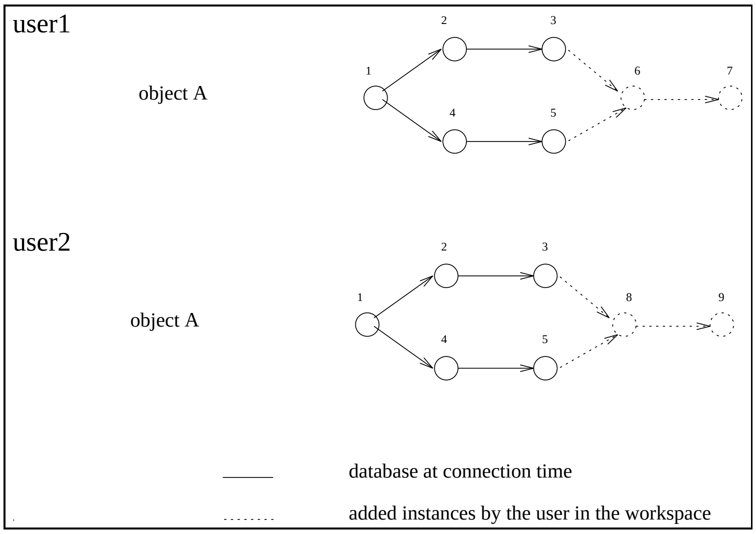


Figure 3: Temporal history in each workspace

In figure 3 we see how two different users have started a session when the same temporal history was known for object A. Each user has created two new versions of this object in his own disconnected workspace. As mentioned, the new versions can be totally ordered with respect to each other in the workspace (dotted lines). When the users both integrate their work in the database, the temporal history of the object A is as shown in figure 4.

Observe that parallelism in the temporal history is only obtained when two different users take out the same state of an object, ie. access the object at the same logical time, and make (logically) simultaneous changes to it.

This kind of history is generally not supported in other systems, although they usually do have time stamping (e.g. [BeM88], [Zdo86]) which then allows building of a linear temporal history. However there are two problems with time stamping. Time stamping requires synchronization of clocks used by the several users. This can be difficult considering that users may have disconnected workspaces and that machines can be all over the world. A worse drawback is that when two users are working in parallel, the information given by time stamping is actually misleading. Time stamping forces a linear order even where this is not desired. Indeed whether one parallel version of an object was time stamped before the other or vice versa is not relevant. The important fact is that the users do not have information about each other's versions, before they are stored in the database.

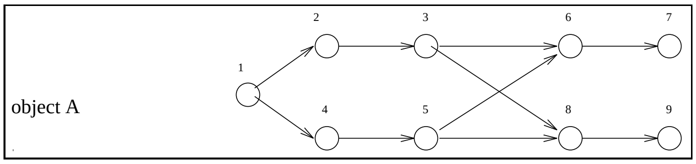


Figure 4: Temporal history after integration in the database

2.3.2 Edit History

The *edit history* is an ordering which reflects information about what version of an object is used to create a new version of a (possibly different) object. The new version is then called the child; the original version is the parent of the new one. As one can use something several times to create something new, this kind of history is also a partial order. (This situation is shown in figure 5. Version 2 of object A is used to create both versions 3 and 9 of object A.) As one can use versions of another object to create a new version of an object, the edit history is a partial order across the borders of objects (see figure 5 : version 1 and 3 of object B are created by using versions 4 and 7 of object A respectively.).

There is a natural constraint that the parent must be created temporally before the child. In figure 5 for instance neither version 3 nor version 4 of object A could have been the parent of version 2. Indeed version 2 was created before version 3 (i.e. version 3 did not exist yet) and in parallel with version 4 (i.e. version 2 could not have known of version 4's existence at its own creation time). These constraints can be checked using the temporal history as long as the parent and the child belong to the same object. (In the case where another object provides the parent, the constraint can be checked using the *command history* ([Hal92]).)

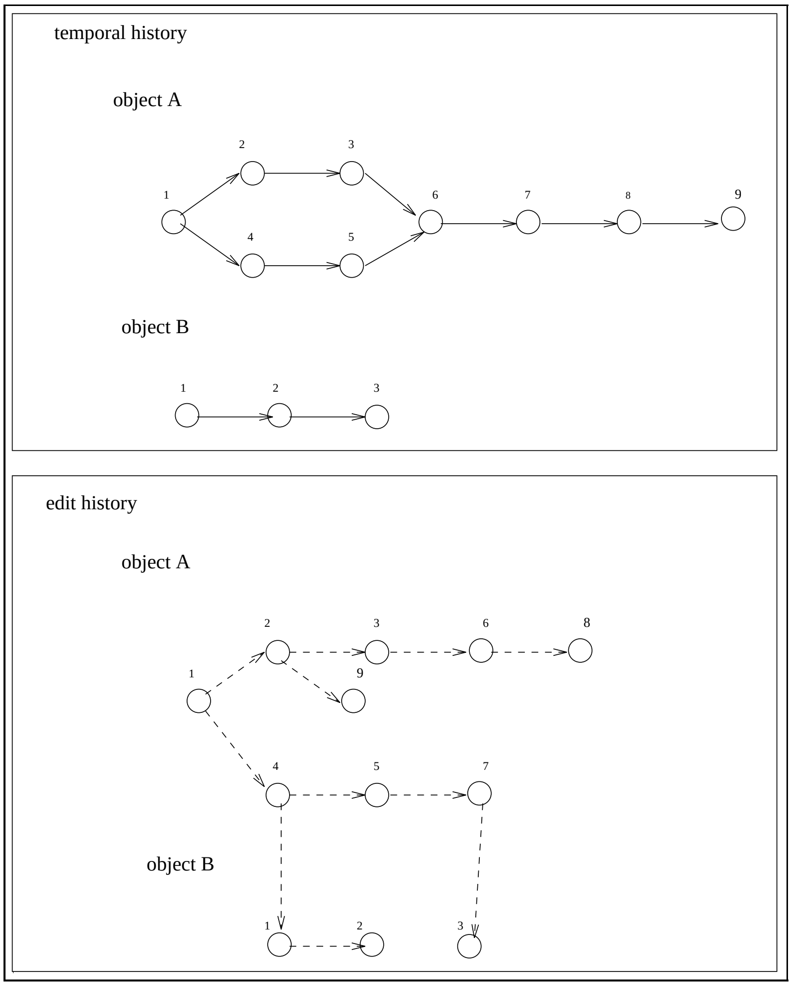


Figure 5: Temporal history and edit history

Intuitively the edit history contains information about the development of an object with respect to the view of the users. A development in the edit history graph where we have no parallelism denotes that an object has been developed in a way such that every new version has been created using the former one. When parallelism occurs we have an indication that two alternatives are tried out to develop the object. This can be done at the same time or it can be done by going back to an older version after not being satisfied with the first development. The information about which of these two has happened can be obtained by using the temporal history together with the edit history of the object. A user can also choose to continue a parallel editing which was initially a result of parallelism in the temporal structure.

In figure 5 we see this illustrated. Versions 2 and 4 both have version 1 as parent and also are parallel successors of version 1 in the temporal history. Thus we conclude that the edit parallelism was a result of editing by different users at the same logical time. Versions 6 and 7 however are ordered in the temporal history and are after versions 3 and 5 respectively in the edit history. What we see here is that two users chose both to continue their own parallel editing branch, although they had the possibility to do otherwise. Version 9 comes after all the others in the temporal history and has as parent version 2. This means that the user has decided to start a new development using an older version of the object.

Observe that in this edit history notion one and the same user can introduce parallelism. So we can make a distinction between different kinds of parallelism. The first one arises from the fact that different users unconsciously update the same object at the same time using the same version as parent. A user can also consciously introduce parallelism by deciding to start a new branch of development starting in an older version. Finally a user can also consciously continue parallel branches, which arose initially by virtue of parallel updates.

2.3.3 Use of History

The usefulness of the edit history is presumably never in question. Most systems supporting version management actually support something similar to our edit history (e.g. [Ba*87], [BeM88], [KCB86], [La*91], [Zdo86]). It seems however that in the other systems one is not allowed to use a version of one object to create a version of another object. Edit history involves historical information about development of an object and about what alternatives were tried out during the development.

The temporal history is essentially historical information representing *transaction time* ([SnA86]). It gives us a means to go back in time and look at what really existed at that time in the database. We discussed before why time stamping alone was not enough to obtain this information. There are many reasoning situations where we need the information given by the temporal history. For example, the temporal history can be used as a means of checking temporal consistency in the edit history graph. We can check whether a specific version could be a parent of another version, as a parent must always exist before a child in the temporal history. It is possible to build an edit history graph which does not fulfill this requirement, but by examining the temporal history, inconsistencies can be discovered.

In reasoning about development decisions it may well be important to know which versions of an object were known to a user when a new version was created. This information is not present in the edit history. From the edit history we can deduce that the parent and its ancestors were known, but there is no means of telling whether versions on alternative branches were known. Looking at the temporal history gives us this information as only the versions which are temporally before the new version could have been known to the creator of the new version. Also the edit history contains no information about abandoned alternatives. Alternative branches which have been abandoned will be in the edit history and look the same as branches which are still open for development. The temporal history can provide us with the information that particular branches have not been pursued for a long time, allowing the system to reason about which branches are no longer current.

3 LITE as Temporal Framework

The LITE[14] ([Rön90], [Rön92]) logic is a first-order predicate logic with an extension allowing objects to be seen as sets of versions. The syntax of the temporal framework is the normal syntax of first-order predicate logic extended by the use of the temporal operator @. The normal first order logic semantics is also used, but over versions, rather than over objects.

In traditional views, time is usually related to propositions and functions. The LITE temporal framework is based on the conception that objects appear as time structures over versions. In other words, a functor or object referent in a sentence refers to a set of formal individuals that constitute the various appearances over time of an object. Interpretation is then done relative to a temporal context through which the object referent is resolved into one of the object versions.

The formal language consists of constants, variables, time constants, time variables, relation symbols, function symbols, the usual connectives, the usual quantification symbols, the tense shift operator @ and auxiliary symbols like parentheses and colon.

Formally, a structure is a tuple (Vs,Obj,Col,Rel,Func). Vs is the set of distinct individuals that correspond to the appearances or versions of the objects. We have a partial time order between versions. Versions belonging to the same object which are not ordered by the partial time order are said to be parallel versions. Obj is the set of objects. An object is a set of versions. All objects are pairwise disjoint. Col is the set of collections. A collection is a set composed of exactly one version of each object such that all those versions are potentially contemporary (see below). The version of object x in the collection t is denoted by x@t. A collection can be seen as a possible time-slice over the objects. Func is the set of functions. A function has its arguments in Vs or Col and its image in Obj. Rel is the set of relations. A relation has its arguments in Vs or Col.

A formula is interpreted with respect to a structure S, a collection t in the structure and a binding h which maps variables to objects and time variables to collections. The principle is to interpret formulas in the classical way with the exception that object references are disambiguated to versions by intersecting the object with the collection t.

[14]LITE stands for Logic Involving Time and Evolution.

By revising the notion of object from being an indivisible entity into being a temporal structure of versions, mixed time relations are easily expressed in LITE. A sentence like "Now, I like young Plato but I don't like old Plato" can be expressed in LITE as follows.

$$\forall \mathrm{t} : ((\mathrm{Young}(\mathrm{Plato@t}) \rightarrow \mathrm{Like}(\mathrm{I@now},\mathrm{Plato@t}))$$
$$\wedge\ (\mathrm{Old}(\mathrm{Plato@t}) \rightarrow \neg \mathrm{Like}(\mathrm{I@now},\mathrm{Plato@t})))$$

We see that in this sentence three different tenses are at work, indicated by 'now', 'Young' and 'Old'. With temporal indexing on objects, 'now' qualifies 'I' while 'Young' and 'Old' are different qualifications for Plato.

Time is thus represented by collections. Due to the importance of this concept, we give the formal definition [Rön92].

Definition

> A set u of one version of each object is a *collection* if and only if there is no object x whose version x@u in u is succeeded by a version α of x that precedes the version y@u of another object y in u.

The words 'precede' and succeed' refer to the partial time ordering for versions. We denote the partial time ordering by $\preceq$ and strict ordering by $\prec$.
An object whose versions are totally ordered in $\preceq$ is said to have a *linear development.*

The time ordering of versions induces an ordering on the collections such that t1$\preceq$t2 for collections t1 and t2 if for all objects x : x@t1$\preceq$x@t2. We note that not all collections are related to each other in this way. If t1$\not\preceq$t2 and t2$\not\preceq$t1 then t1 and t2 are regarded as parallel collections. This could for instance represent different possible ways in which the world can have evolved.

Let us assume for the example in figure 6 that u ={x3,y1}, t1={x1,y1}, t2={x2,y2} and t3={x3,y2}. Then we have that u is no collection. We also have that t1, t2 and t3 are collections and t1$\preceq$t2, t1$\preceq$t3, and t2 and t3 are parallel collections.

As collections and time are so strongly related, we often write in this text 'x at time t' where we actually mean 'the version of object x in the collection t'.

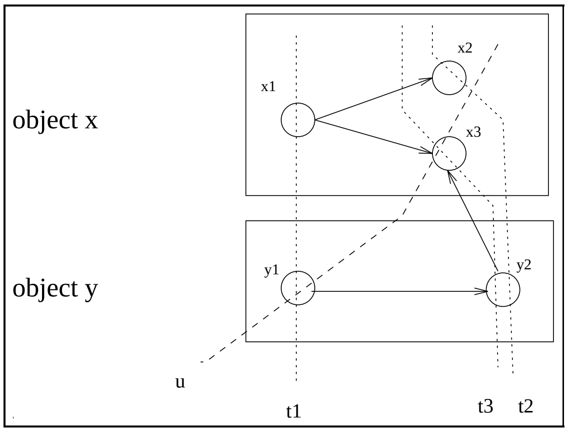


Figure 6: Ordering between collections

4 Connections

We have identified two important but different types of composite objects that typically occur in information systems. A crucial difference between those types becomes clear when we want to propagate changes of connected objects (components) to connecting objects (composition).

In this chapter we define the notions of strong and weak connection formally. It will be intuitively clear that the notions of composition described in chapter 1.3, are examples of the different connection notions. We rather define 'connection' because the same behavior exists in other examples where no composition is involved. For instance the length of a list of persons wanting a copy of a paper is in a strong connection with the copy count which says how many copies should be made.

4.1 Strong and Weak Connection

That which we call strong connection is exemplified by the relation which exists between a document and its sections. When a section of a document changes then we consider the document also to have changed.

We can formalize this in the following way.

Definition

> For objects A and B, relation R and collection t :
>
> A is in a *strong connection* with B at t with respect to R
> iff
> $R(A,B)@t \wedge \forall\, t' \in Col : B@t=B@t' \rightarrow A@t=A@t'$.

Weak connection is exemplified by the kind of relation which exists between a cabinet and its folders. We only consider a cabinet to have changed if a folder is added to or deleted from the container. When a folder of a cabinet changes internally then we do not consider the cabinet to have changed.

Definition

For objects A and B, relation R and collection t :
A is in a *weak connection* with B at t with respect to R
iff
R(A,B)@t $\wedge$ $\forall$ t' $\in$ Col : B@t=B@t' $\rightarrow$ R(A,B)@t'.

Observe that both definitions refer to objects at certain times, not to the objects in general.

The differences between the two kinds of connections become clear in the propagation of changes from connected object to connecting object. We have the following.

Definition

For objects A,B,C , relations R1,R2 and collection t1, :
such that R1(A,B)@t1 and R2(B,C)@t1,
we say that *changes are propagated from A to C via R1 and R2 at time t1*
iff $\forall$ t2 $\in$ Col : A@t1$\neq$A@t2 $\rightarrow$ C@t1$\neq$C@t2.
(Change Propagation Property)

We can then prove that changes are propagated via strong connections, but not via weak connections.

Theorem 1

1. For objects A,B,C , relations R1,R2 and collection t1,
   the following holds :
   if A is in a strong connection with B at t1 with respect to R1
   and B is in a strong connection with C at t1 with respect to R2,
   then changes are propagated from A to C via R1 and R2 at time t1.
2. The Change Propagation Property does not hold in general for weak connection.

Proof

(1) Assume A is in a strong connection with B at t1 with respect to R1 and B is in a strong connection with C at t1 with respect to R2 and A@t1≠A@t2. Then by the fact that A is in a strong connection with B at t1 with respect to R1, and A@t1≠A@t2, also B@t1≠B@t2. Then by the fact that B is in a strong connection with C at t1 with respect to R2, and B@t1≠B@t2, also C@t1≠C@t2.

(2) We provide a counterexample (see figure 7). Let A,B,C be objects. Let a1,a2 be different versions for A, b a version for B and c a version for C. Let these be the only elements for Vs. Assume R1(a1,b), R1(a2,b) and R2(b,c). Let t1={a1,b,c} and t2={a2,b,c}. Then A is in a weak connection with B at t1 with respect to R1 as R1(A,B)@t1 and for all t such that B@t=B@t1 holds that R1(A,B)@t. Also B is in a weak connection with C at t1 with respect to R2 as R2(B,C)@t1 and for all t such that C@t=C@t1 holds that R(B,C)@t. Also A@t1≠A@t2. Nevertheless C@t1=C@t2.

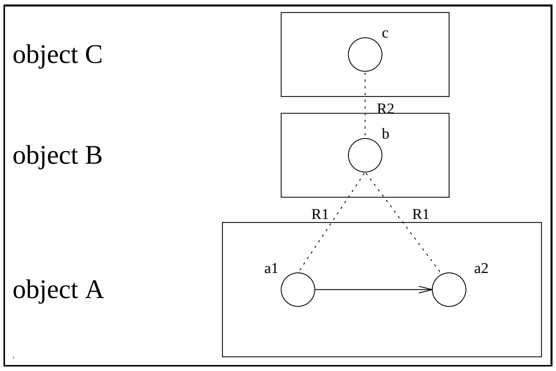


Figure 7: Counterexample

□

The theorem tells us that changes propagate via strong connection. Eg., if a paragraph in a section changes, then also the document containing the section changes. The Change Propagation Property does not hold in general for weakly connected objects. For instance if a folder in a cabinet changes then the cabinet itself does not have to change.

Another observation is that if an object A is in a strong connection with an object B at time t with respect to relation R, then it is also in a weak connection to B at t with respect to R.

4.2 Combining Strong and Weak Connections

One way to combine strong and weak connections is within one layer of the hierarchy. The definitions in the former section do not rule this out. It is possible that an object A is in a strong connection with B at time t with respect to R1 and also C is in a weak connection with B at time t with respect to R2. (see figure 8(a)). This is a useful observation. A section can have annotations. We may want to consider these annotations to be in weak connection to the section. At the same time a section consists of paragraphs which are in a strong connection with the section. We also allow an object C to be in a strong connection with object A with respect to relation R1 at the same time as it is in a weak connection with object B with respect to relation R2 (see figure 8(b)). A document can at the same time be a weakly connected part of a folder and a strongly connected chapter of a book.

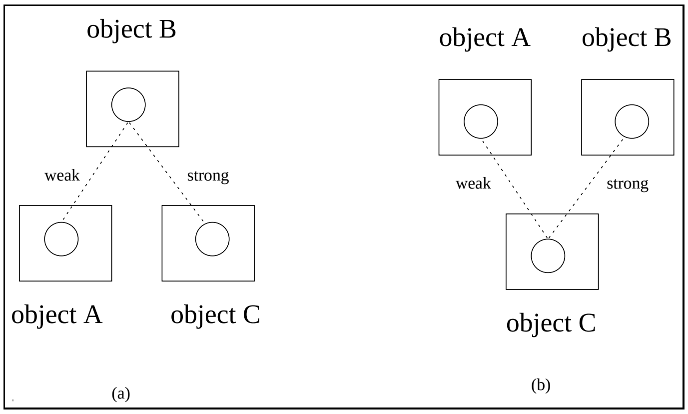
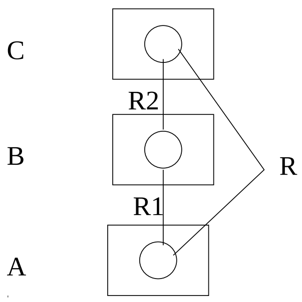


Figure 8: Combinations of weak and strong connections

When we combine the connections over several layers in the composition hierarchy it is interesting to investigate of what type the combined connection will be. These results work in two ways. As we will see, the combination of two strong connections results in a strong connection. This means that (i) the user has received a lower bound for the resulting connection (at least strong) and (ii) we have the strong connection for the resulting connection for free already. The first part gives us a way to check inconsistencies in the modeling by the user. The second part gives us a way to decide when some composed connections are already maintained by maintaining other connections.

Theorem 2 states that when in a 3-layer hierarchy, the connection between the two upper layers is strong, then the strength of the connection between the lowest and the highest layer is the same as the connection strength between the two lower layers (see figure 9).



connection strength

R1 \ R2	strong	weak
strong	strong	?
weak	weak	?

Figure 9: Connection strength of combined connection

Theorem 2

Let R1 and R2 be relations. Define R as a relation such that R(A,C)@t if and only if $\exists$B : R1(A,B)@t $\wedge$ R2(B,C)@t.

(i) if A is in a strong connection with B with respect to R1 at t, and B is in a strong connection with C with respect to R2 at t, then A is in a strong connection with C with respect to R at t.

(ii) if A is in a weak connection with B with respect to R1 at t, and B is in a strong connection with C with respect to R2 at t, then A is in a weak connection with C with respect to R at t.

(iii) if A is in a strong connection with B with respect to R1 at t, and B is in a weak connection with C with respect to R2 at t, then A is in general not in a weak connection with C with respect to R at t.

(iv) if A is in a weak connection with B with respect to R1 at t, and B is in a weak connection with C with respect to R2 at t, then A is in general not in a weak connection with C with respect to R at t.

Proof

(i) By definition of strong connection, we have that R1(A,B)@t and R2(B,C)@t. But then also R(A,C)@t. Using the Change Propagation Property we also know that for every collection t' holds that if A@t≠A@t' then also C@t≠C@t'.

(ii) By the definitions of strong and weak connection we have that R1(A,B)@t and R2(B,C)@t. But then also R(A,C)@t. To prove the second part of the weak connection, we have to prove that for every collection t' holds that if ¬R(A,C)@t' then also C@t≠C@t'. So, assume that we have a collection t' such that ¬R(A,C)@t'. Then we know that ¬R1(A,B)@t' or ¬R2(B,C)@t'. (Indeed, if R1(A,B)@t' and R2(B,C)@t', then we would have by definition of R that R(A,C)@t'.)
If ¬R2(B,C)@t', then by the fact that B is in a strong (and thus also weak) connection with C with respect to R2 at t, we know that C@t≠C@t'.
If ¬R1(A,B)@t', then by the fact that A is in a weak connection with B with respect to R1 at t, we have that B@t≠B@t'. Further, by the fact that B is in a strong connection with C with respect to R2 at t, we have that C@t≠C@t'. In both cases the desired result is obtained.

(iii) A counterexample is provided in figure 10 where t={a1,b1,c} and t'={a2,b2,c}. The conditions for strong connection between A and B are satisfied as well as the conditions for weak connection between B and C. However the fact that ¬R(A,C)@t' does not imply that C@t≠C@t'.

(iv) The counterexample here is similar to the one in theorem 1 (see figure 11).

□

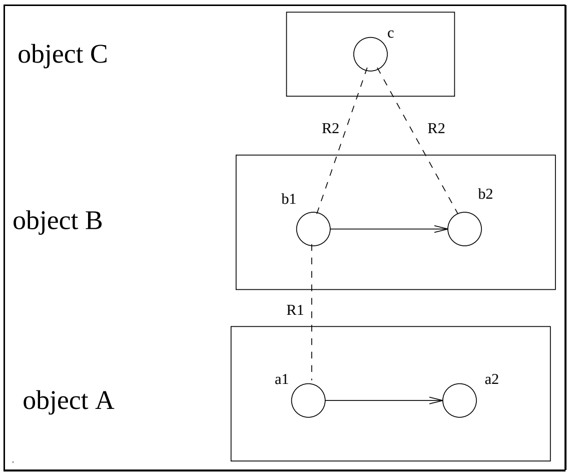


Figure 10: strong and weak connection

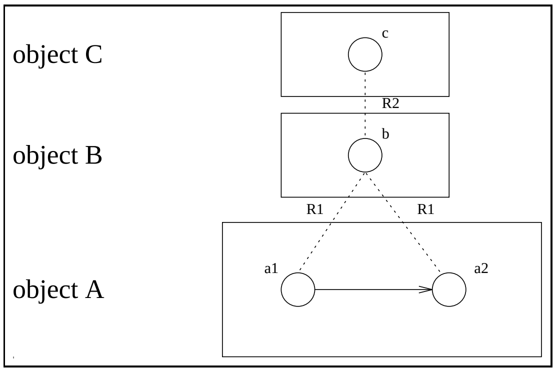


Figure 11: weak and weak connection

4.3 Synchronization Rules

To enforce the behavior of strong and weak connection for objects in a database in a consistent way, we have developed temporal constraints for the versions of the objects. We give here simple synchronization rules for the case where the objects have a linear development[15]. We note that assuming that the objects have a linear development does not mean that things cannot happen in parallel. It is possible that two different objects change at the same time, but it is not possible that one object has two parallel versions at the same time.

Theorem 3 gives us synchronization rules for a structure where the objects have a linear development such that if R(A,B) holds at time t, then A is in a strong connection with B at t with respect to R. In other words, the synchronization rule enforces strong connection.

The synchronization rule is a conditional statement. The conditional part depends on the relation and on the definition of strong connection. The synchronization itself is more universal. Informally, the synchronization rule describes a process such that whenever a connected object is changed, the new version of the connected object is set to be before the new connecting object version (see figure 12). In theorem 4 we have a similar synchronization rule for weak connection. The part in the rule depending on the definition of strong connection is there replaced by a part depending on the definition of weak connection.

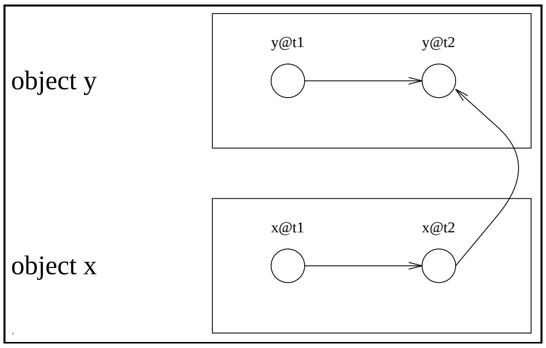


Figure 12: Synchronization rule

[15]We will discuss briefly how to extend the rules for developments where parallel versions are allowed further in the text.

Theorem 3

Let(Vs,Obj,Col,Rel,Func) be a structure.

Let R be a relation.

If

$R(x,y)@t2 \rightarrow (x@t1 \neq x@t2 \rightarrow x@t2 \preceq y@t2)$

for objects x,y and collections t1,t2 such that $x@t1 \preceq x@t2$.

then

$R(A,B)@t \rightarrow (B@t=B@t' \rightarrow A@t=A@t')$ for all t' such that $t' \preceq t$.

whereby we assume that the temporal relation between versions does not allow any cycles.

Proof

Assume R(A,B)@t and $t' \preceq t$. Then we have to prove that if B@t=B@t' holds, that then also A@t=A@t' holds. So let us assume that B@t=B@t'.

As $t' \preceq t$, we know that $A@t' \preceq A@t$. So we have two cases.

(i) A@t=A@t'.

Then the theorem is satisfied.

(ii) $A@t' \prec A@t$.

We can apply the synchronization rule. We have then $A@t \preceq B@t$. Then we would have the following : $A@t' \prec A@t$, $A@t \preceq B@t$, and B@t=B@t'. It means that we have an object A whose version A@t' is succeeded by a version A@t of A that precedes the version B@t' of an object B. By the definition of collection, t' cannot be a collection then. This is in conflict with the assumption that t' was a collection. So this case cannot occur.

□

For the document-section example, this theorem says that if a section is a part of a document at time t, then the synchronization assures us that if the section changed from t' to t, then the document also changed, i.e. we have a strong connection behavior.

Let us have a look at how the synchronization rule works for the strong connection behavior. Assume that A and B are objects and t and t' collections such that A@t$\preceq$A@t' and R(A,B)@t'. We have then 4 possible collections : t, {A@t,B@t'}, {A@t',B@t} and t' (see figure 13(a)). Assume also that A@t$\neq$A@t'. Applying the synchronization rule gives us then that A@t'$\preceq$B@t' (see figure 13(b)). But then {A@t,B@t'} cannot be a collection. This implies then also that object B must have changed from t to t'. There is nothing that disallows {A@t',B@t} to be a collection and in fact this collection could denote the intermediate time that the update for a connected object has been made, but the update is not yet propagated to the connecting object.

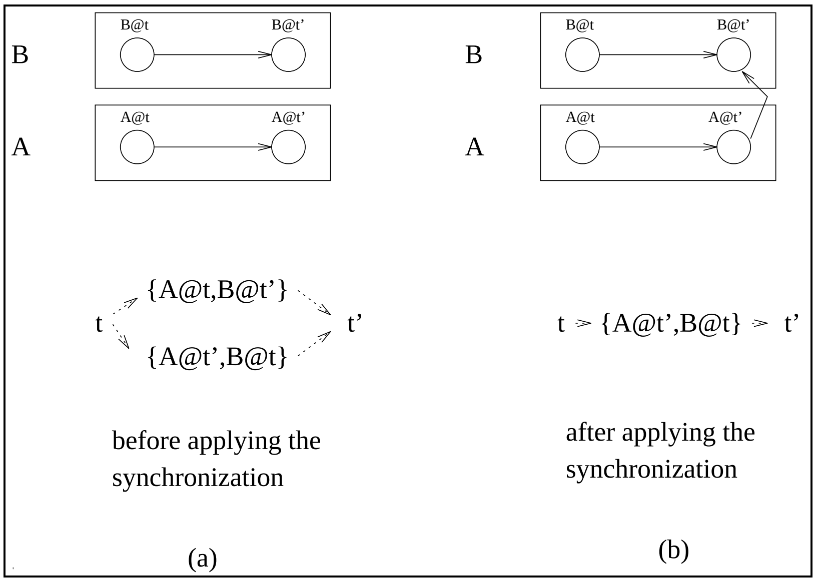


Figure 13: How the synchronization rule works

We can also enforce the desired behavior for weak connection by using synchronization rules. Observe the difference in the conditional part of the synchronization rule with respect to strong connection. We replaced x1$\neq$x2 by $\neg$(R(x,y)@t2). This difference is similar to the difference between the definitions of weak and strong connection. The synchronization itself is the same. Informally, the synchronization rule describes a process such that whenever a connected object becomes 'unconnected', the new version of the connected object is set to be before the new connecting object version.

Theorem 4

Let(Vs,Obj,Col,Rel,Func) be a structure.

Let R be a relation.

If

R(x,y)@t2 $\rightarrow$ ($\neg$R(x,y)@t1 $\rightarrow$ x@t2$\preceq$y@t2)

for objects x,y and collections t1,t2 such that x@t1$\prec$x@t2.

then

R(A,B)@t $\rightarrow$ (B@t=B@t' $\rightarrow$ R(A,B)@t') for all t' such that t'$\preceq$t.

whereby we assume that the temporal relation between versions does not allow any cycles.

Proof

Assume R(A,B)@t and t'$\preceq$t. Then we have to prove that if B@t=B@t' holds, that then also R(A,B)@t' holds. So let us assume that B@t=B@t'.

As t'$\preceq$t, we know that A@t'$\preceq$A@t. So we have two cases.

(i) A@t=A@t'.

Then by the fact that B@t=B@t' and R(A,B)@t, we also have that R(A,B)@t'.

(ii) A@t'$\prec$A@t.

If R(A,B)@t' holds, then the theorem holds for this case.

If on the other hand it does not hold, then we can apply the synchronization rule. We then have that A@t$\preceq$B@t. Then we would obtain the following : A@t'$\prec$A@t, A@t$\preceq$B@t, and B@t=B@t'. It means that we have an object A whose version A@t' is succeeded by a version A@t of A that precedes the version B@t' of an object B. By the definition of collection, t' cannot be a collection then. This is in conflict with the assumption that t' was a collection. So this case cannot occur.

□

The synchronization rules are thus proved to be adequate for maintaining the type of connection desired in the two different kinds of composite objects when we are dealing with objects with a linear development. In the case where

parallel versions are allowed, the theorem still holds, but the result does not include the full strong or weak connection behavior. Consider the example in figure 14. All the conditions are satisfied to apply theorem 3. So we know we have a strong connection behavior for all t such that t$\preceq$t2 and even for all t such that t$\preceq$t3. However the theorem does not say anything regarding collections involving parallel versions. Therefore the theorem allows both a2 and a3 to be in the R-relation with b2. For full strong connection behavior, we should have had that B@t2$\neq$B@t3.
For a document and a section, this means that the theorem allows two parallel versions of a section to belong to the same version of the document. A way to resolve this problem is to add the constraint that parallel versions for connected objects give rise to parallel versions of connecting objects. The implementation in one of the following sections applies in a way this solution.

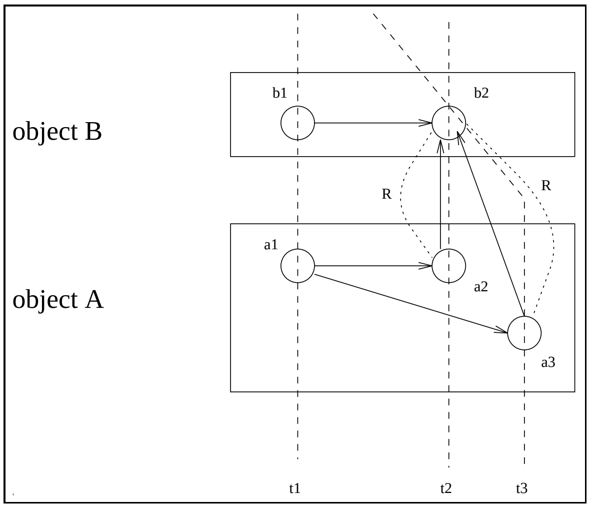


Figure 14: Theorem 3 and parallel versions

5 Historical Information of Compositions and Dynamic Binding

It is of course natural that we want the same kind of historical information[16] about composite objects as we do about atomic objects. However as we show below it is not necessarily straightforward to obtain such information using only the historical information of the components. In particular dynamically bound links and shared object parts can pose a problem for history management. Indeed, as the example below shows, with the historical information about the components we can construct several plausible developments for the composition. As only one of these can have occurred in reality, we are unable to look at a composition how it was previously. By taking all possible combinations of components we may show versions of the composition which never in fact existed. We describe and discuss this problem for strongly connected compositions. We note where interesting differences occur when we deal with weakly connected compositions.

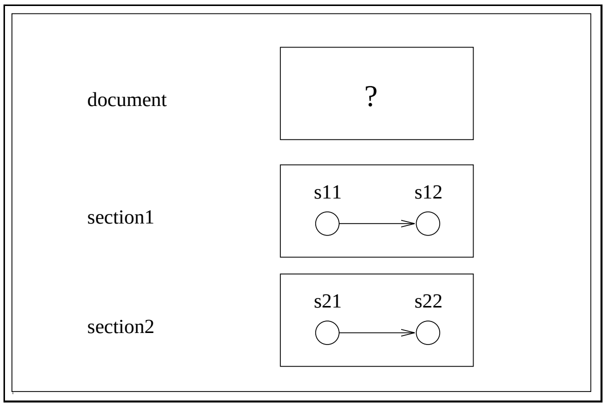


Figure 15: Historical information of components is inadequate for composition

Consider as example figure 15. We have a document that is composed of two sections, both of which have two versions. With the information we have, we

[16]In this chapter and the following chapters addressing the problem of maintaining historical information of compositions using the historical information of components, the examples will always concern temporal history. The problem and the mechanism described however are essentially also valid for the edit history. In section 9.2 we discuss briefly a complication concerning composite objects and edit history.

can compose the versions of the sections as follows to obtain four different versions of the composition :
{ s11, s21 }, { s11, s22 }, { s12, s21 }, { s12, s22 }. The following historical evolutions are then possible for the document.

(i) The first version of the document contains the first version of section 1 and the first version of section 2. Then section 1 changes and we have a document containing the second version of section 1 and the first version of section 2. Finally also section 2 changes and the last version of the document contains the second version of section 1 and the second version of section 2.

(ii) The second possible evolution for the document is the symmetrical case of (i) where the second section changes first.

(iii) There is also the case that both sections change simultaneously. Then the history of the document only contains two versions : one version containing both first versions of the two sections and one version containing both last versions of the two sections.

(iv) Finally if there is no synchronization at all between the two sections, we have parallelism in the document history. It means that two users take out the document simultaneously and each of them modifies a different section. After both changes are saved to the database the document contains both updated sections.

In the general case we will obtain many different possible combinations of component versions as versions for the composition. The lack of synchronization information between the different components gives us very large branching in the history where many branches contain combinations of components which in fact never existed.

This problem occurs as a result of using dynamically bound links. At each request the dynamic binding mechanism takes care of the bindings which have to be made to get the 'latest' version of the composition including the default versions of the components. However the component bindings that were used for a particular version of a composition are lost at later times. In figure 16 by using dynamically bound links the root of the composition does not need to

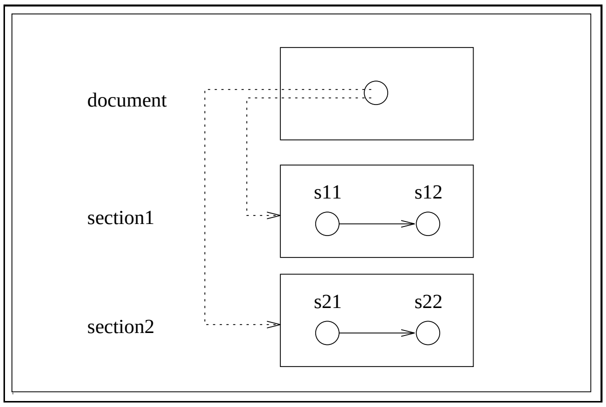


Figure 16: Dynamically bound links between document and sections

change although the components do. Therefore the logical history of the document is not reflected by the versions of the root. Trying to reconstruct versions of the whole may well lead to inconsistencies with respect to how the document has actually looked at previous times. A possible solution to the problem in this case is of course not using dynamically bound links. One could imagine that we would update the composition every time a component changes using statically bound links. However if components are shared by many different compositions this propagation becomes computationally complex and we do not derive the natural advantages of dynamically bound links. The solution is also not sufficient when an object is the root of different compositions. For instance a patient journal in a hospital environment may be accessed by doctors and nurses. However the composition for each may be different. The nurses may not be able to access all information a doctor can and vice versa, while they do share some information. The root of the patient journal serves then as root for two different compositions : the nurse's patient journal and the doctor's patient journal (see figure 17). In this case when a part of the doctor's journal changes which is not accessible by the nurses, a propagation of the change to the root would reflect a change of the patient journal for the doctor but this would not be desired from the nurse's point of view, as for the nurse the patient journal has not changed.

An objection to this example might be that as we have two different compositions, we should also have two different composition roots. However, a component and a root of a composition can be seen as an object in itself but we may want to pick up different parts of the underlying structure depending on what it is being seen as a component of, at the same time as it is important that components be shared between the different compositions.

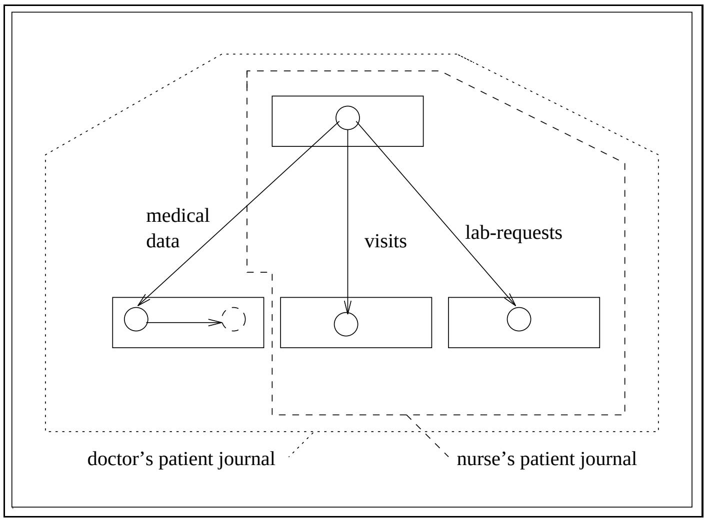


Figure 17: Objects can serve as root for different compositions.

6 Historical Information of Composite Objects

In this chapter we consider what are the requirements to be able to maintain historical information of composite objects. We define *composite object representations* as a means to satisfy these requirements. Then we describe an implementation in the LINCKS system.

6.1 Composite Object Representations

To be able to maintain a consistent history of a composition that provides us with the information about which versions of a composition did exist, we need to know the following information :

for each time point :

- which version of the composition root exists at that time,
- what the hierarchical structure of the composition is at that time, together with the information about which objects participate as components and at what place in the structure,
- whether a component is a statically or dynamically bound component,
- for each dynamically bound component, which version of the component exists at that time.
- (For each statically bound component no resolution for the component needs to take place.)

Observe that these requirements imply information about synchronization between the composition and its components as well as about synchronization between the components. (The information about synchronization between the composition and its components alone is not enough, as we saw in the former chapter.)

To maintain this information we introduce the notion of *composite object representation.* A composite object representation is an object for which each version represents a version of the composite object as it existed. Every version contains the structure of the composition, including which objects participate. Every version has also a set of statically bound links to versions of the objects in the structure. An update to the composite object representation occurs then when a weakly (or strongly) connected component is added or deleted or a strongly connected component is updated[17].

Observe that a composite object representation is an explicit representation of a particular composition whereas the root of a composition may represent several compositions. The history of the composite object representation reflects the history of its particular composition. This allows for the root to maintain dynamically bound links, while the composite object representation has the set of statically bound links which existed at various points in time. Note that a root can have more than one composite object representation if it is used for more than one composition. For the patient journal example for instance the root of the patient journal serves as root for two different compositions: the nurse's patient journal and the doctor's patient journal.

We show now how the composite object representations solve the problem in figure 16. To keep the figures simple, we have only represented the statically bound links from the composite object representation versions to the root and the component versions. Keep in mind that also the information about the structure of the composition is contained in the composite object representation versions. When section 1 changes first and then section 2, the composite object representation has a history as shown in figure 18. When section 2 changes first and then section 1, the composite object representation has a history as shown in figure 19. The sections can also change simultaneously and for this case the composite object representation history is as shown in figure 20. Finally as last possibility we have the case where the different sections are changed simultaneously by different users and the changes are merged afterwards. The composite object representation history shows the parallelism

[17]For the weak composition there is an interesting complication. As long as the structure of the composition is unchanged (i.e. no adding or deleting of components), the composition is considered to be unchanged. Therefore no new version of the composition should be created. However it is possible that the composition is unchanged, but still some of the components have changed. By looking at the composition we see then one composition version in two appearances. We have then two possibilities. We can ignore the fact that the components have changed, as by definition of weak connection, this has no real influence on the composition. The other possibility is to keep track of these component changes in *versions of composition versions*. (In the latter case it seems that accessing the weak connection induces a kind of strong connection on a lower level.) More work needs to be done on this topic however.

(see figure 21). In each case the composite object representation provides us the history we would want the composition to have.

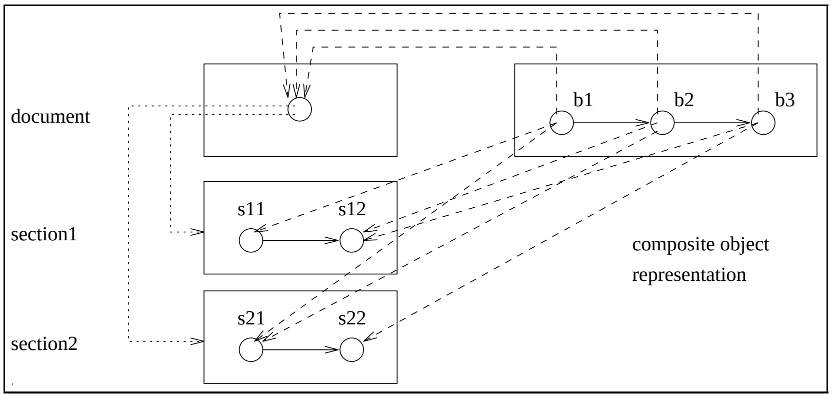


Figure 18: Section 1 changes first, then section 2

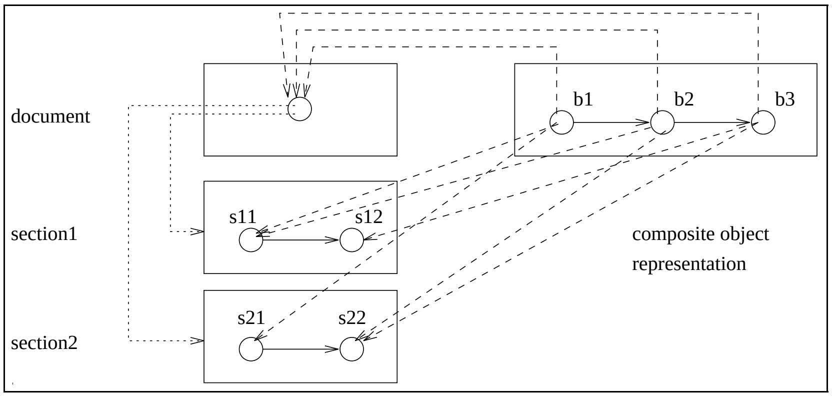


Figure 19: Section 2 changes first, then section 1

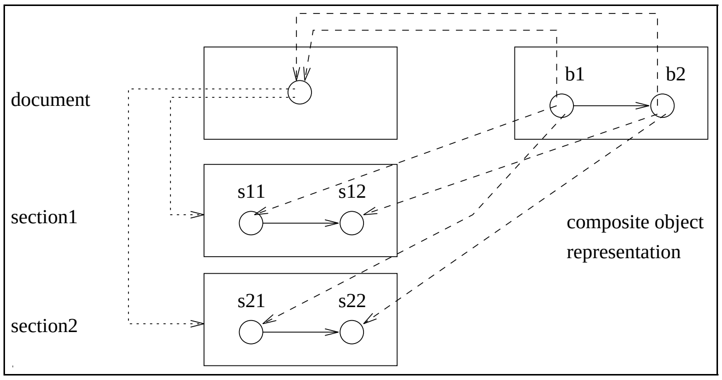


Figure 20: Section 1 and section 2 change simultaneously

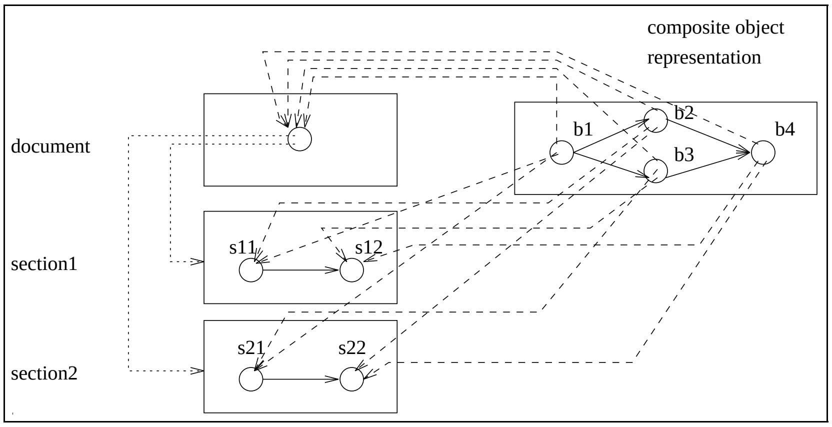


Figure 21: Simultaneous change by different users to section 1 and section 2, followed by a merge.

6.2 Composite Object Representations in LINCKS

In this section we describe the way LINCKS handles composite objects. We introduce the notion of *presentation description objects* ([Åbe89]) which provide templates for classes of composite objects. Then we define the *binding tables* and show that together with the presentation description objects they constitute composite object representations. We show how a form of delayed updating saves us from propagating all changes to a component immediately to all compositions where it is a part. Further we describe the *composite object structures* in LINCKS. These composite object structures are used to build displays of composite objects. We discuss the difference between a composite object and a *view* of a composite object. Finally we propose the implemented functionality.

6.2.1 Presentation Description Objects

Composite objects in LINCKS are built using *presentation description objects*, and *contents objects* (the 'real data' objects) ([Åbe89]). The presentation description objects provide a template for a class of composite objects. A particular composite object is then created by instantiating the template with a set of contents objects. The contents objects are the root and the components of the composition. This way of handling compositions gives us a strong flexibility. It allows us to share contents objects between different compositions. It also allows us to use different presentation description objects with the same root object, giving us different compositions sharing the same root.

The information contained in a presentation description object is divided into four parts [Tei90].

- The layout of the composition is given by the *structure* part. Both composite and terminal atomic elements in the composition are defined. Composite elements are themselves composed of sub-elements, which may be either composite or terminal atomic elements. The composite and terminal elements of the composition are given by the tokens in this definition.
- The information for locating objects and the parts of the objects to be placed within the composition can be found in the *access* part. This access information is also stored within the composite object structure

(see further) and is used to determine which database objects should be updated when performing editing operations towards the composite object structure.

- The *format* part specifies the formatting information to be associated with the tokens in the structure definition.

- The *expand* part specifies which presentation description objects should be applied towards the components when they are expanded. This can happen in the case that the components themselves are compositions for which the components are not necessarily considered as components for the original composition.

In LINCKS, the presentation description objects can also be composite objects. For instance, it is natural to have as part of a presentation description object for documents a part which is a presentation description object for sections. Presentation description objects are also versionable. If we look upon presentation description objects as classes for a particular kind of objects (e.g. the 'document' class) then versioning of presentation description objects is actually doing *schema evolution*. Schema evolution mechanisms address the problems that may arise when an object is created under one version of the class and is accessed through another version of that class. Consistency problems may occur when attributes or links are added or deleted. Work on schema evolution is reported in for instance [Ah*84], [KiC88], [NgR89], [PeS87] and [SkZ87]. Although the issue of schema evolution is outside the scope of this work, it seems that the mechanism of composite object representations and presentation description objects gives us attractive schema evolution features. It seems to give us the ability of updating objects and classes as well as using the old versions of objects and classes if desired[18].

6.2.2 Binding Tables

We define a new kind of object, the *binding table*, which will be part of the composite object representations in LINCKS. The binding table versions link

[18]An important element in our schema evolution mechanism would be the fact that the system can apply a presentation description object to any contents object as root. When the system does not find a component, it notifies the user and builds up the composition with the available components. The missing components are denoted by so called *place holders* (see further). The case of 'too many' components does not occur as the system uses the presentation description object to know which objects are components.

together a root version and the component versions of a composition which existed at the same time. We have chosen however not to represent the statically bound components in the binding table versions[19].

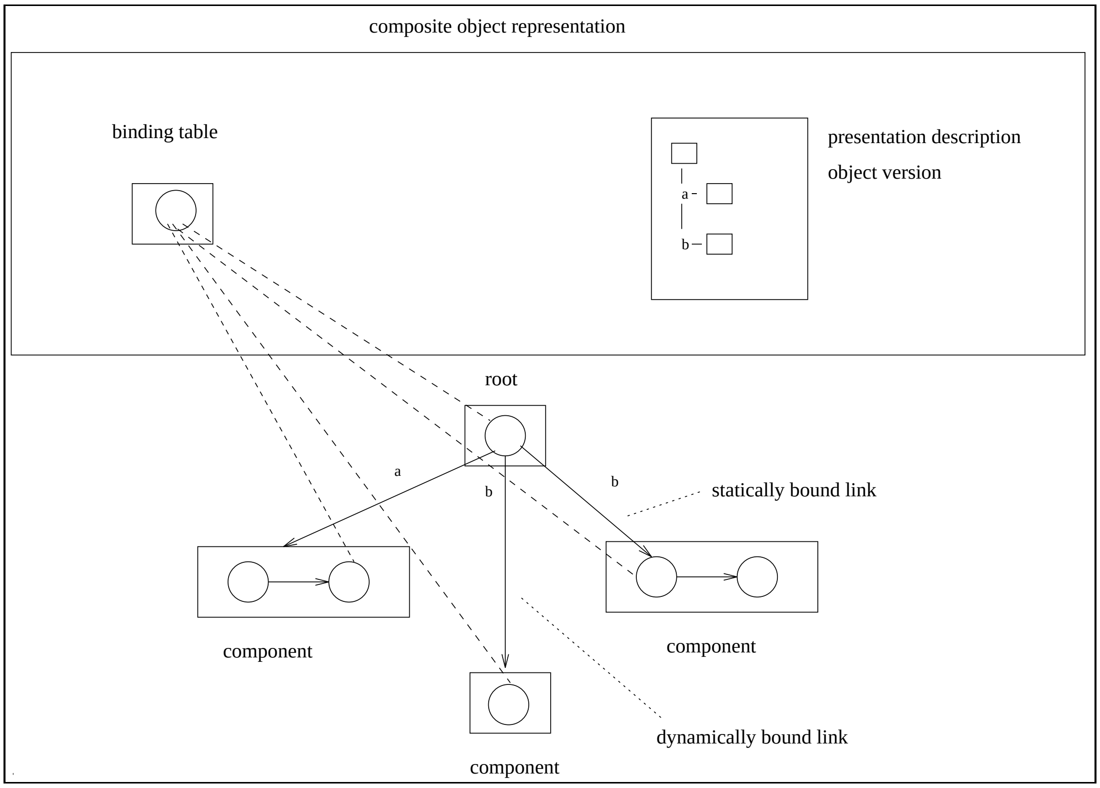


Figure 22: Binding tables and presentation description objects : composite object representations in LINCKS

It is clear that a presentation description object version together with a binding table constitutes a composite object representation (see figure 22). The hierarchical structure of the composition is found in the presentation description object version. The binding table version is connected to the root version of the composition. The information about which objects participate in the composition are found by using the presentation description object. Whether

[19]The intuition here is that if a user has created a statically bound link to a component version, then it is the case that he wants this version of the component to be in the composition (until an explicit command not to have this anymore). Therefore no update of the composition should be done, even if the component object has changed.

a component is statically bound or dynamically bound, can be found in the component object of a higher level. For the statically bound components, the component version is defined in a component of a higher level. For the dynamically bound components, the resolved component versions are found in the binding table object. So all the information needed for a composite object representation is available.

To maintain the historical information of a composition we should do the following. Whenever a component changes (strong connection) or a component is added or deleted (strong and weak connection), a new version of the binding table should be created. The root has to be resolved and linked statically to the new binding table version. For every dynamically bound component, which is found using the presentation description object, the component has to be resolved to a particular version and linked statically to the new binding table version.

In practice however we create for each composition a new object: the binding table for that composition. The history of the binding table reflects then the history of accessing and modifying that particular composition. This means that we do not always update the binding tables *immediately* whenever a component changes. Indeed we only update the binding table without delay if something in the composition changes[20] while accessed through the specific composition represented by the binding table[21]. However also before accessing a composition we make sure that the history of the composition is updated if changes have been made to a component since the last access to the composition. If several possibilities exist to do this updating (in the case of parallelism for instance), then the user is notified. The user can then decide whether he wants the parallelism in the history of the binding table or not. This policy allows us to maintain the benefits of dynamic link resolution while maintaining all relevant historical information. In figure 23 we see that a component is part of two different compositions. However as the component is accessed through the composition represented by binding table 2, it is only binding table 2, which is updated after a change of the component. When we afterwards access the composition represented by binding table 1 the system will check

[20]The 'change' in the composition means for strong connection, a change in a component, or deletion or addition of a component. For weak connection it means the deletion or addition of a component.

[21]This way of working, i.e. delaying the updating until access time is called *screening* in [NgR89] in contrast to *conversion* where all updates occur immediately. The advantages of screening are performance and a limitation of the propagation of the changes. A disadvantage is the possibility of inconsistency between two accesses (see the end of this section).

whether changes have been made since the former access and in this case automatically generate a new version for binding table 1 (see figure 24). The user is then allowed to edit the composition.

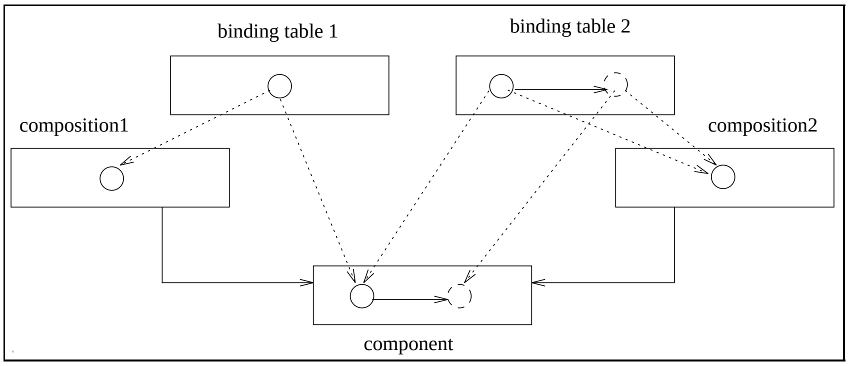


Figure 23: Binding tables and sharing : component accessed through composition of binding table 2

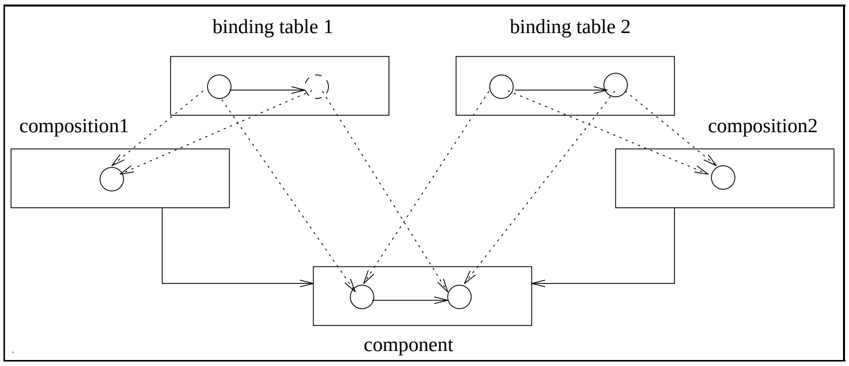


Figure 24: Binding tables and sharing : the composition of binding table 1 is accessed

An advantage of this use of the binding table is of course that we do not have to propagate changes to all components and all binding tables immediately. The intuitions behind this are the following. When a section is edited through a particular document 1 it is not of primary concern for users of another document 2 which shares the section with document 1 to know the changes that are made to the section until some user of document 2 accesses the section.

It is sufficient for a user of document 2 to know that there have been changes and to have a version where all part-of links are resolved at (document 2) access time.

In between two uses of a composite object C, it may happen that a particular component has been updated a number of times through other compositions. The composition C however will not contain the intermediate versions of the updated component. It is possible that a user would like to back up in the history of that component and create a version of C containing one of those intermediate versions. In figure 25 we see two components that are shared by three different compositions. A problem may now occur when a user would like to create a version of composition 2 containing versions *a2* and *b2*. None of those versions have been created through composition 2. It may be that temporal constraints (in general or in composition 2) do not allow this combination of the versions. Even if the combination is allowed, there remains the question of where in the history of the binding table the version of this new combination should come.
Currently in the LINCKS system we do not allow a user to combine older versions of compositions in this arbitrary way. Formal work needs to be done, to resolve questions regarding temporal consistency (see section 9.2).

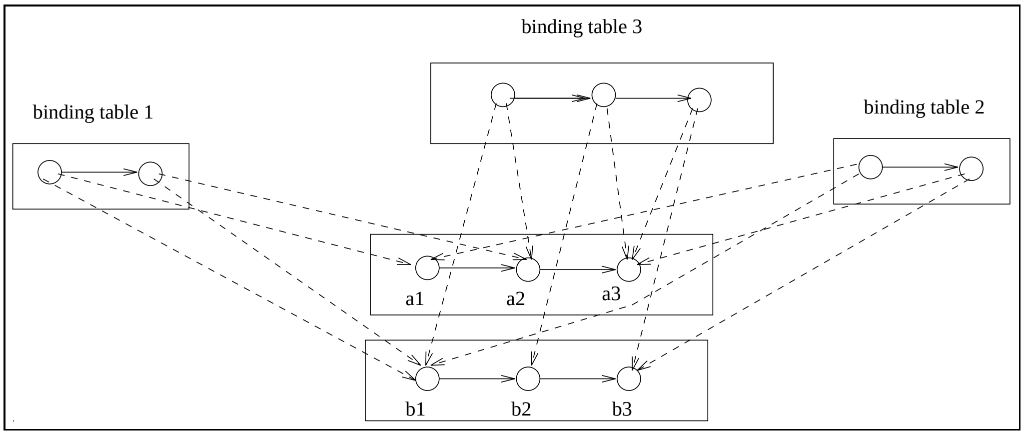


Figure 25: Binding tables and temporal consistency

6.2.3 Composite Object Structures

A *composite object structure* as implemented in LINCKS ([Tei90]) consists of two tree structures that represent a *view*[22] of a composite object[23]. The trees are called *composite object template* and *composite object display*. Composite object structures are built using presentation description information, found in the presentation description objects, and contents objects in the database ([Åbe89]).

The composite object template provides the structure and the corresponding access information for a composition type. It contains no actual data information, but provides the framework for both retrieving and storing the parts of a composition. The composite object templates for the different kinds of compositions are found in the presentation description objects.

The composite object display is built from a template together with the contents objects. It represents a view of a particular composition as it currently exists. It uses the information in the composite object template to link together the information needed from particular versions of several objects. The dynamically bound links are resolved by using the binding tables.

In figure 26 we see a composition that is built using a presentation description object and four data objects. Observe the difference between the composite object display and the composite object template. The composite object template gives us the information that from the root of the composition, we should follow the *a* and *b* links. The links can have multiple values. In the composite object display the root is instantiated with a particular version of one of the contents object. From this version we follow the *a* and *b* links. In this particular example there happens to be one *a* link and two *b* links. The links are resolved (if needed) and particular versions of the linked objects are added to the composition.

In LINCKS it is possible to have partially instantiated composite object displays. This happens when the composite object template forces the system

[22]A view of an object in the LINCKS system is a presentation of this object. It presents information contained in an object, not necessarily all information, in a well specified way (by the formatting information it contains).

[23]In general the implementation of the composite object structure in LINCKS does not have to represent a view of a composite object. It can also represent a view of an aggregation of objects that are linked together and where the links do not necessarily denote part-of. We can think here of hypertext-like links. However for this thesis we restrict ourselves to views of composite objects.

to follow a link which is not found (a 'missing component'). In that case the system puts a *place holder* in the composite object display. The place holder can be seen as an indication that the user can create something new or connect something existing there if desired.

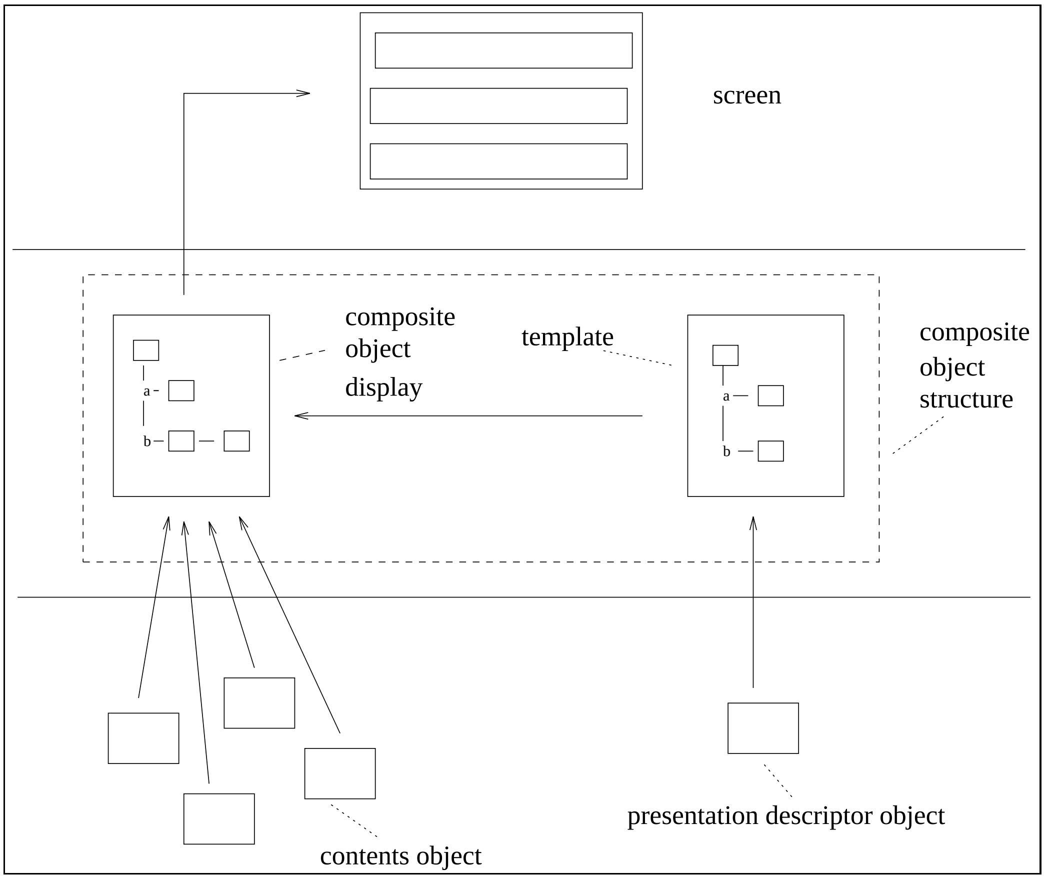


Figure 26: Composite object structure, contents object and presentation description object

We want to note here the difference between a view of a composite object (represented by the composite object structure), and the composite object itself (represented by the composite object representation versions). The composite object is a tree structure of objects linked together by the part-of relation. The view of a composite object is a tree structure of *parts* of objects linked together by the part-of relation. The view of the composite object is the *visible part* of the composite object. However the root and the component objects can have some hidden information. As a consequence, our mechanism is really a versioning mechanism for the composite objects, and not for the views of the composite objects. The changes in the object in a composition depend on visible information as well as hidden information.

In figure 27 we see a composite object having a root and three components. However we have two different views over the same composition. In view 1 we see that part a11 of the first component, part b11 of the second component, and part b12 of the third component, are visible. In view 2, parts a21, b21 and b22 are the visible parts of respectively components one, two and three. When we update now a visible part of a component in view 1, then that component changes. Therefore (in a strongly connected composition) the composition changes. If the visible parts of view 1 and view 2 are disjoint, then we have an update of the composition, while at the same time the visible part of the composition in view 2 is the same.

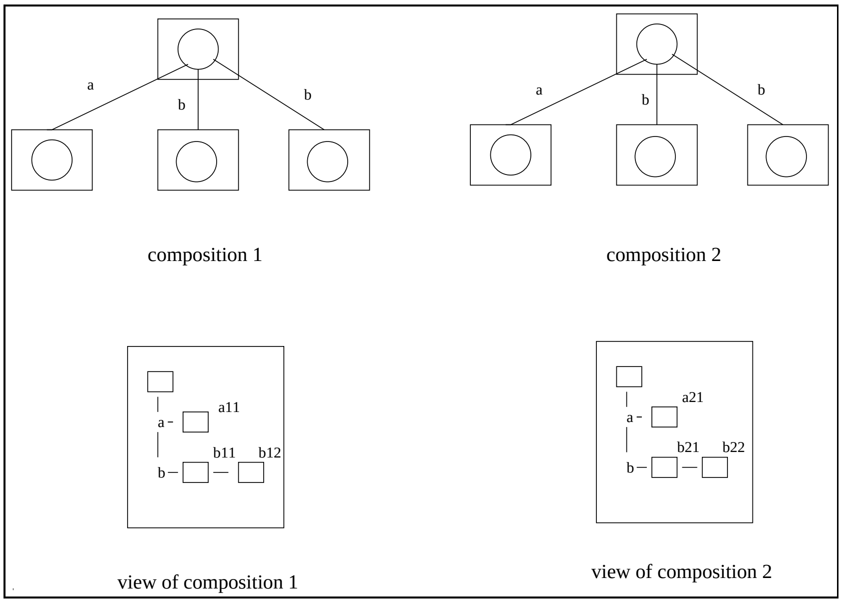


Figure 27: Composite objects and views of composite objects

It is clear that this difference between the composition and the view over the composition is a matter of granularity. When we make sure that views of a composition always consider the whole objects as visible (without hidden information) then the compositions and the views over the compositions coincide.

6.2.4 Implemented Functionality

The information about which binding table and presentation description object constitute a specific composition can be found in the root object of the composition. The root contains a list of pairs <presentation description object,binding table>, denoting the compositions for which this object is the root.

We provided three functions in the system which are used to do the version management of the compositions.

Get-Binding-Table-Version
IN : Presentation Description Object, Composition Root
OUT : Binding Table Version

Get-Binding-Table-Version searches for the binding table associated with the composition root and the presentation description object. If no binding table is found[24], a new binding table and an empty transient version are created. Otherwise we select the best binding table version. The best binding table version is (i) the latest version[25] (if there is only one), or (ii) a latest version owned by the user (if there is one), or (iii) the default version[26] for the dynamic binding mechanism. The user is notified when parallel latest versions exist.

This function is called upon access of a composition.

Get-Component/Root-Version
IN : Binding Table (Version)[27], Component/Root Object
OUT : Component Version

Get-Component/Root-Version gets the appropriate version for a component/root with respect to a binding table. If the function is called using a binding table version, then the function returns the version of the component in the binding table version. Otherwise the function returns the version of the component in the default version of the binding table. In this case, if a change has occurred in the component since the last access of the component through this composition, the binding table is updated.

[24]This means that this object is for the first time a root for a composition with as template the given presentation description object.

[25]with respect to the temporal history

[26]see section 2.2.

[27]The function can be called with a binding table or a binding table version.

This function is used in two ways. When the system builds up the composite object structure it calls the function for each dynamically bound component to get the appropriate component version and update the binding table if necessary. The second use is when the system propagates the changes of a component to all compositions containing that object visible on the screen. In that case the function is used to force update of the binding table.

Remove-Redundant-Components
IN : Binding Table Version, List of Versions of Components
OUT : Binding Table Version

The update procedure for the binding table in *Get-Component-Version* takes care of the addition of new components and the changing of old components. However components are never deleted (or never go out of the composition) in that procedure[28]. This function, *Remove-Redundant-Components*, removes the deleted components.

This function is called before a binding table is stored in the database.

Observe that after an explicit store for a composition, the binding table should also be stored.

6.3 The Composite Object Representations and the Synchronization Constraints

It is easy to see that the composite object representation mechanism as described above, satisfies the synchronization rules of theorem 2 and 3. We map the compositions or connecting objects to the composite object representations. As the composite object representations represent compositions which did in fact exist at a particular time, they represent partial collections[29]. Then the following holds for these collections. The composite object representation version is associated with versions of components which already existed at the creation time of the composite object representation version. This means that the synchronization rule of section 4.3 is satisfied. Indeed, the versions of the components are temporally before the version of the composition.

[28]Observe that this is actually no real problem. As the composite object representation contains both the presentation description object and the binding table, the deleted components are not reached through the template. However it seems 'cleaner' to remove the deleted components also from the binding table versions before they become working versions.

[29]A *partial collection* is a collection restricted to a subset of the total set of objects.

The case of parallel versions is taken care of as follows. First, when two users update the same composition at the same logical time, two parallel versions of the composite object representation are created. Both versions of the composite object representation are created later than each of their respective components.
Secondly, when there are several possibilities for delayed updating of a composition at access time, then the user decides whether he wants the parallelism in the history of the composite object representation or not (see former section). If the user decides not to introduce parallelism we have a new binding table version which is latest in the temporal history and for which the components existed before the composition. If the user decides to introduce parallelism, then parallel component versions have given rise to parallel versions of the composite object representation and again the components were created earlier than the composite object representation versions.
In all cases the synchronization rules and the remark regarding parallel versions at the end of section 4.3 are satisfied.

7 Scope of Propagation and Path Ambiguity

Katz and Chang [KaC90] discuss two problems related to change propagation. The first problem has to do with the *scope of the propagation*. The question posed is how far up in the composition hierarchy do we propagate the changes (see figure 28). Answers by other authors range from little (one level up) or no propagation, to propagate up to the root. In LINCKS we access an object always through a composition/composite object representation. Therefore we propagate the change of a component (if needed) to this composition or composite object representation. Propagation on higher levels is deferred until we access the higher level composition. In figure 29 the component is accessed through composite object representation 2. So the composition associated with composite object representation 2 is updated. The update of composite object representation 1 is deferred until access of composite object representation 1.

The second problem has to do with the fact that a composition may refer to a particular component via different paths. There is then an *ambiguity* in how many times a change in that component should be propagated to the composition (see figure 30). Most systems try to limit the effect to only one update to the composition. As accessing an object always happens through a composition and updating always occurs in workspaces, our solution is similar to the *group-check-in* of [KaC90]. New versions of several components can be created before a new version of the composition is created. We commit to the new version of the composition when an explicit store command for the composition is given. Therefore also in LINCKS ambiguity is resolved to induce only one update to the composition (see figure 31).

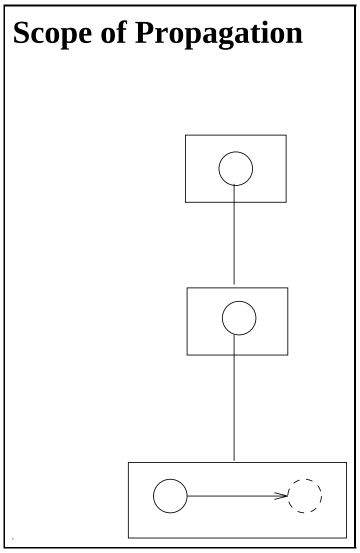


Figure 28: The Scope of Propagation Problem

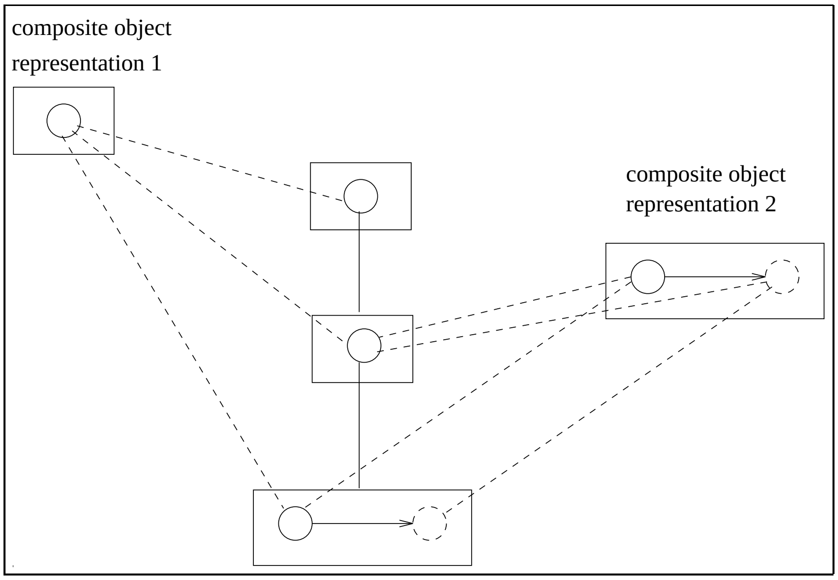


Figure 29: The Scope of Propagation Problem in LINCKS

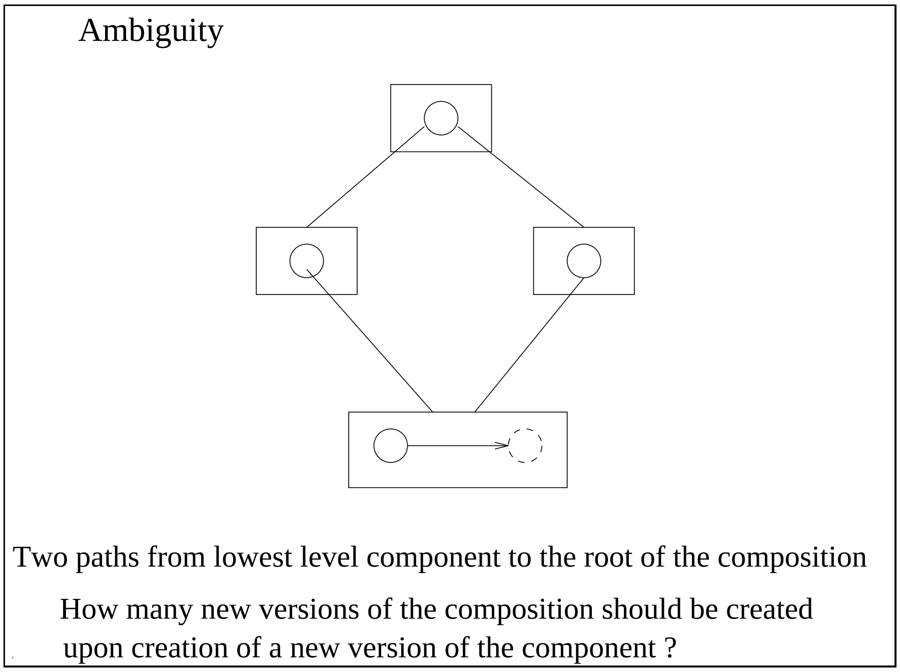


Figure 30: The Ambiguity Problem

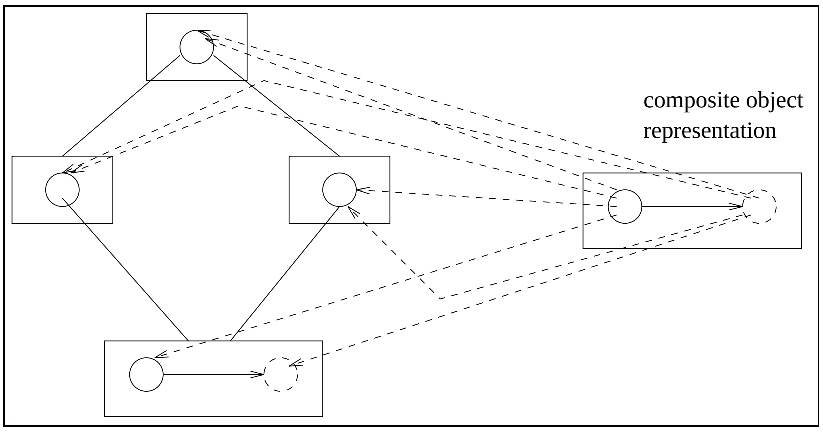


Figure 31: Solution to the Ambiguity Problem in LINCKS

8 Related Work

In [Zdo86] the author notes that a problem can occur which is similar to our strong and weak connection (the *percolation* property). However Zdonik argues that conceptually propagation of change always makes sense, although one might not want to do this always in practice (in which case information is lost). We argued by introducing the notions of strong and weak connection that the propagation even conceptually is not always desired. Zdonik notes that the propagation can be done using triggers. However it seems that one cannot then use dynamically bound links for the strongly connected components. Zdonik also recognizes the problem of parallel versions. To deal with the problem of consistency, *slices* are associated with transactions. A slice is a set of versions that has been produced during the transaction. Going back in time can only be done via slices, i.e. by denying the occurrence of a transaction. In this way one would always retrace to states of the database that really existed.

IRIS ([BeM88]) allows a particular feature for its version management which we do not allow in our model. In IRIS it is possible to go back in time and make a minor change to a version of an object (a minor bug fix for instance) and then say that it is actually the same version. In LINCKS we would create a new version. Dynamic binding is resolved by a *context* mechanism based on user defined rules in the form of *daemons* which may be triggered. With respect to composite objects they observe that it is sometimes desirable to propagate a change of a component to a composition. This can be done by specifying that a particular component is *significant* for implicit version creation. A problem which then still remains is the proliferation of new versions. A drawback is that when in the case of strong compositions, this mechanism is not used, then information is lost, while if it is used, the price is that dynamic binding is again useless.

In the ORION system the compositions are divided into four classes ([KBG89]) depending on whether components can be shared and whether a component can exist without the composition. However they do not seem to have a similar notion to our strong and weak connection. As support for the historical information of composite objects they introduce rules. It seems however that these rules tell us what part-of links (statically bound and dynamically bound) are permitted between two objects, but it still leaves us with cases where several developments are possible. Consequently we are unable to go back in time to show what a composition really looked like. Dynamic binding is resolved on the basis of a time stamp ordering on the creation of versions. ORION supports a flag based notification mechanism [BaK85] to notify users about updates. Each object has two time stamps : the time that the object was changed latest (CN) and the time when changes were last acknowledged (CA). An object is *implementation consistent* if its CN is before its CA. A composition is *reference consistent* if its CA is later than all the CN of its components. ORION-2 [Ki*91] also supports a message based notification approach where the system sends messages to notify users of potentially affected objects. The users can be notified immediately or at some later time specified by the user.

The Version Server in [CGK89] resolves dynamic binding by specifying *environments*. In the early version of the Version Server propagation from component to composition was carried out automatically to the root. There was no way in limiting the effect. In [GeK88] however the authors describe a graphical representation system for the compositions which can be used to hold a dialog with the user. An interesting notion to resolve the problem of ambiguity is *group-check-in*. This mechanism ensures the user that only one version of the composition is created. The problem of consistency is tackled by introducing a *layer* system. If two versions of different objects must be used together, then they are placed in the same layer. A *context* defines then a search order over the layers. A drawback of this method is that it is still possible to combine the layers in ways that do not make sense. There are no explicit constraints that state that particular versions of particular objects cannot exist at the same time.

In [DiL88] an object is seen as a set of versions with one current version. There is no support for propagation of change from components to compositions. Dynamic binding is resolved by introducing *environments*. Environments bind an object to a particular version or point to another environment to bind a particular object. Environments have some similarity with the composite object representations (without the propagation mechanism). Also environments have to be created by the user. The consistency problem is partly solved by making *clusters* of versions to select particular versions of a component who might be bound to a particular version of a component. (However to obtain complete consistency we would have to synchronize also the components.)

In [WKS89] composite objects are dependent in the sense of ORION. The following conditions are required for the component relationship. (i) the composite object inherits data of the component and is not allowed to update them; (ii) if an update of the component occurs, the update has to be propagated to the composite object which possibly has to be adapted afterwards; and (iii) updates of the composite object do not change data of its components, only relationships to components or between components are touched. This model is very close to the model we use. Condition (ii) is in our model as follows. In the strong connection the composition changes if the component changes. In the weak connection the composition can but does not have to change. In both cases the latest instance of the composition is related to the new instance of the component.

Rumbaugh [Rum88] introduces an interesting mechanism for controlling operation propagation across relationships. The intended operations were operations as copy, destroy, print, save, lock and display. However, it seems that this mechanism may also be used in propagation of change. It is based on assigning particular *propagation attributes* to operation-relationship pairs. The values for these attributes can be *none* (the relation is ignored for the operation), *propagate* (the operation is applied to the relation elements containing the source object and the operation is propagated to all related objects), *shallow* (the operation is applied to the relation elements containing the source object but the operation is not propagated to the related objects) and *inhibit* (this value suppresses any attempt to apply the operation no other attributes are examined and no propagation occurs). The last value is especially for the destroy operation, where it indicates an object that should not be destroyed because it is needed by another object. Each operation is applied to each object only once in each course of propagation. Strong compositions could use the propagate value connected with a composition relation and a 'create' operation. Weak compositions might use the shallow value.

Compositions are shared independent in the sense of ORION terminology in [AgJ89]. Versions of composite objects are not explicitly stored, but they are put together on the fly. This mechanism associates with each transaction a read and write set of versions. A correct configuration (composition) is associated with a consistent state of the database. Consistent states however are not only states as how the database really looked like at a specific point in time, but also states as how the database could have been with respect to different serialization orders of a set of transactions. The problem tackled in [AgJ89] is thus actually : given a set of transactions and a set of 'pre-creation' versions (1 for each object), find the correct configurations such that particular dependencies between transactions, derived from the read and write sets, are satisfied. Algorithms to find all correct configurations are provided in the general case and in the case of so called *regular systems*. An algorithm to determine whether a complete (i.e. involving every object in the database) configuration is correct, is also provided. However it is noted that the generalization of this algorithm to cope with non-complete configurations (the usual case for compositions) is a non-trivial exercise.

The problem of consistent databases, which can be seen as a generalization of the problem of consistent compositions is addressed in [CeJ90]. A database version is a representation of a real world state. The model includes *versioning* and *non-versioning transactions*. A versioning transaction creates a new database version. A non-versioning transaction queries or updates a database version, causing its evolution independent from the evolution of other database versions. The user is responsible for writing transactions which bring the database from an initial consistent state to a new consistent state. Database versions have version stamps. If a database version has a version stamp of the form i, then the version stamps of the database version derived from this version are of the form $i.j$. For each object a table relates a version of the object with a list of version stamps of the database versions. Default rules are used when several database versions share the same object versions. This mechanism allows us to obtain consistent compositions, but it is still the user who decides when to propagate changes from components to compositions. Also, the consistency depends on the ability of the user to write well-behaved transactions.

Ahmed and Navathe [AhN91] address the propagation discussion even further than strong and weak connection. They argue that when the *interface properties* of an object change, that we actually should create a new object. When the *internal assembly* changes, a new version should be created. A classification is made to determine the degree of *updatability* of an objects. Attributes can be *version-significant*, *nonversion-significant* or *invariant*. They are respectively attributes which give rise to creation of a new version upon update (strong attributes), attributes which can always be modified, and attributes which can never be modified. Compositions are defined to be version-significant (strong compositions). New versions of components give rise to new versions of the compositions. However, if a composition is itself also a component in another composition, the user can decide whether a new version of the higher level composition should be created or not. This mechanism prevents the proliferation of versions, but puts the burden on the user.

9 Conclusion

9.1 Relevance of the Work

We addressed the issue of maintaining historical information of composite objects in object-oriented database systems. We argued that at least two different kinds of history should be supported : temporal history and edit history. Whereas edit history is usually supported in other systems, temporal history is usually not. Most systems use time stamping instead. We discussed why we do not think time stamping to be appropriate.

In combining composite objects and historical information, we addressed two problems. The first problem has to do with the question whether changes in a component should induce the creation of a new version of a composition. In other systems this question is answered in different ways. ('Yes, but it is not practical or efficient to do this' or 'No, if the user would want this, let him do it explicitly'.) By introducing the notions of strong and weak connection, we allow a user to vary the answer to this question. The strong connection implies that a new version of the composition is always created upon a change, an addition or deletion of a component. A weak connection implies that a new version of the composition is always created upon an addition or deletion of a component, but not necessarily when a component changes. When the composition is not even weak, then it is possible for a composition to stay the same, even when components are added or deleted. We also provided some (non-)transitivity rules for the connections. Further we provided synchronization rules to maintain the different kinds of connections automatically in a database setting. The composite object representations satisfy these rules.

The second problem we addressed, concerns the problem of maintaining historical information of strongly connected compositions using the historical information of their components. We showed that the historical information of the components alone is not enough. An extra mechanism is needed. We introduced the notion of composite object representations which are essentially time slices over the compositions, and discussed an implementation in the LINCKS system. We showed how the scope of propagation is limited and how the path ambiguity problem is addressed.

9.2 Further Work

Further work is still needed to allow full user flexibility. We describe some of the concrete problems and topics which are part of ongoing and future work.

Differences between Strong and Weak Connection

Work remains to be done in further investigating behavioral differences between strongly and weakly connected composite objects. A strongly connected component is also weakly connected. We also know that the change propagation property holds for strong connections but not in general for weak connections. We want to investigate whether the change propagation property is the necessary and sufficient condition to characterize the strong connections as a subset of the weak connections or whether there are other essential behavioral differences[30].

Compositions with Strongly and Weakly Connected Components

We also plan to investigate more the interaction and/or combination of strong and weak connections. We will explore the behavior of compositions which have both strongly and weakly connected components. We will also investigate multi-level composition hierarchies with respect to when connections should be propagated within the composition hierarchy.

An example of a composition with both strongly and weakly connected parts could be a folder of documents (weakly connected) with a document list which is strongly connected to the folder. Thus the folder is considered to have changed if either (i) a document is added to or deleted from the folder, or (ii) if the document list is modified in any way. Interactions between these two causes of change in the composition (resulting from the fact that the composition has both strong and weakly connected components) need to be studied.

[30] Another difference between strong and weak connection, which is however related closely to the change propagation property, is the following. Let A be a composition with component B. Then the formula 'B=B@t' is *transient* for the version A@t for the weak connection and *characteristic* for the strong connection. We do not go into details here and refer to [Rön92] for definitions of transient and characteristic formulae with respect to versions.

For multi-level composition hierarchies we already made a start by giving (non-)transitivity properties in section 4.2.

Other Properties of the Part-Of Relation

The part-of relationship covers a set of relations which all have the composition-component intuition in common, but which differ in other respects. One of the properties is the strong and weak connection property. Other properties already mentioned (see chapter 1.3) are for instance (i) whether a component can be part of different compositions or if it is exclusive for a particular composition or (ii) whether components can exist only when they are part of a composition or not. These seem to be natural constraints for compositions giving rise to new classes. These constraints have an influence on the historical information of the objects. For instance if an object is an exclusive component and is going to change composition, then this implies a certain synchronization in the histories of the component and the different compositions. We will investigate further classification of compositions and the relation between the different classes and the temporal and edit history. We will also try to find simple synchronization rules to maintain these properties automatically.

Edit History and Composite Objects

Shared objects can change in several ways. One can take the object itself and create a new version. It is however also possible that an object changes as part of a composition. In the first case it is clear that we have two different versions of the object for which the second is a direct successor of the first in the edit history. In the second case things are less clear. Is the predecessor in the edit history the same as in the first case ? Is it the composite object ? Or is it the simple object but in the particular context of the composition ? The semantics and ramifications of these different possibilities need to be explored.

Temporal Consistency

It may be the case that some composition would like to introduce a composite object representation version which reflects a former change which is reflected in another composite object representation, but not in its own composite object representation. Currently this is not allowed in the LINCKS system. Formal work needs to be done, to resolve questions regarding temporal consistency. One question is for instance whether the new combination could have existed within this composition. It is possible that temporal constraints between different components make a combination of versions of the components inconsistent. In such a case we may want to copy component versions in order to allow the building of a temporally consistent composite object representation, rather then allowing a version of a composite object representation which combines non simultaneous versions of components. Even when the combination of component versions is allowed the question remains regarding how this new version of the composition should be integrated within the history of the composition. One possibility would be to insert the new version of the composition into the history as if it had been built at the time the components were current. Another possibility would be to treat it as the most recent version of the composition (which in terms of access it is). However this latter approach raises further questions regarding the temporal consistency of a composite object representation and its components.

User Defined Successor Relationships

We want to keep our approach as general as possible. A theoretical base has been provided for temporal and edit history. However we want it to be easy to translate our methods to similar methods for arbitrary partial orders between versions of objects. This is particularly useful when a user wants to define his own predecessor/successor relationship between versions. To be able to do this mapping between our histories and the user defined successor relationships we need to understand the structure and properties of the histories well. Using LITE[31] we will investigate this subject. With these results it should then be possible to define and classify successor relationships with respect to differences in properties and mappings between those relationships and the different kinds of histories (or substructures of the histories).

[31] Observe that the partial order between the versions in LITE already satisfies some *admissibility* constraints ([Rön92]). We might want to try to loosen those constraints too.

10 References

[AgJ89] Agrawal, R., Jagadish, H.V., 'On Correctly Configuring Versioned Objects', in *Proceedings of the International Conference on Very Large Data Bases - VLDB 89*, pp 367-374, 1989.

[Ah*84] Ahlsen, M., Björnerstedt, A., Britts, S., Hulten, C., Söderlund, L., 'Making type Changes Transparent', *Research Report*, SYSLAB report 22, SYSLAB-S, University of Stockholm, 1984.

[AhN91] Ahmed, R., Navathe, S.B., 'Version Management of Composite Objects in CAD Databases', in *Proceedings of the Conference on Modeling of Data - SIGMOD 91*, pp 218-227, 1991.

[An*90] Anderson, T.L., Berre, A.J., Mallison, M., Porter, H. H., Schneider, B., 'The Hypermodel Benchmark' in *Advances in Database Technologies, Proceedings of the Second International Conference on Extending DataBase Technology - EDBT 90* , pp 317-331, 1990.

[At*89] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S., 'The Object-Oriented Database System Manifesto', *Technical Report*, GIP-ALTAIR, No. 30-89, LeChesnay, France, 1989.

[Ba*87] Banerjee, J., Chou, H.-T., Garza, J.F., Kim, W., Woelk, D., Ballou, N., Kim, H.-J., 'Data Model Issues for Object-Oriented Applications', in *ACM Transactions on Office Information Systems*, Vol 5(1), pp 3-26, jan 1987.

[BaK85] Batory, D.S., Kim, W., 'Modeling Concepts for VLSI CAD Objects', in *ACM Transactions on Database Systems*, Vol 10 (3), pp 322-346, sept 1985.

[BeM88] Beech, D., Mahbod, B., 'Generalized Version Control in an Object-Oriented Database', in *Proceedings of the 4th IEEE Conference on Data Engineering*, pp 14-22, 1988.

[BeM91] Bertino, E., Martino, L., 'Object-Oriented Database Management Systems : Concepts and Issues', in *IEEE Computer*, pp 33-47, April 1991.

[CeJ90] Cellary, W., Jomier, G., 'Consistency of Versions in Object-Oriented Databases', in *Proceedings of the International Conference on Very Large Data Bases - VLDB 90*, pp 432-441, 1990.

[CGK89] Chang, E.E., Gedye, D., Katz, R.H., 'The Design and Implementation of a Version Server for Computer-Aided Design Data', in *Software - Practice and Experience*, Vol 19 (3), pp 199-222, march 1989.

[Cod70] Codd, E.F., 'A Relational Model for Large Shared Data Banks', in *Communications of the ACM*, Vol 13(6), pp 377-387, 1970.

[DiL88] Dittrich, K.R., Lorie, R.A., 'Version Support for Engineering Database Systems', in *IEEE Transactions on Software Engineering*, Vol 14 (4), pp 429-437, april 1988.

[GeK88] Gedye, D.M., Katz, R.H., 'Browsing the Chip Design Database', in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp 269-274, 1988

[Hal92] Hall, T., *Maintaining a Command History in LINCKS*, undergraduate thesis, LiTH-IDA-Ex-9221, Department of Computer Science, Linköping University, 1992.

[ILE88] Iris, M.A., Litowitz, B.E., Evens, M., 'Problems with the part-whole relation', in *Relational Models of the Lexicon; Representation of Knowledge in Semantic Networks*, ed. Evens, pp 261-288, 1988.

[KaC90] Katz, R.H., Chang, E., 'Managing Change in a Computer-Aided Design Database', in *Readings in Object Oriented Database Systems*, eds. Zdonik, Maier, pp 400-407, 1990.

[Kat90] Katz, R.H., 'Toward a Unified Framework for Version Modeling in Engineering Databases', in *ACM Computing Surveys*, Vol 22 (4), pp 375-408, dec 1990.

[KBG89] Kim, W., Bertino, E., Garza, J.F., 'Composite Objects Revisited', in *Proceedings of the Conference on Management of Data - SIGMOD 89*, SIGMOD Rec., Vol 18(2), pp 337-347, 1989.

[KCB86] Katz, R.H., Chang, E., Bhateja, R., 'Version Modeling Concepts for Computer-Aided Design Databases', in *Proceedings of the Conference on Modeling of Data - SIGMOD 86*, pp 379-386, 1986.

[KiC88] Kim, W., Chou, H.-T., 'Versions of Schema for Object-Oriented Databases', in *Proceedings of the International Conference on Very Large Data Bases - VLDB 88*, pp 148-159, 1988.

[Ki*91] Kim, W., Ballou, N., Garza, J.F., Woelk, D., 'A Distributed Object-Oriented Database System Supporting Shared and Private Databases', in *ACM Transactions on Information Systems*, Vol 9(1), pp 31-51, jan 1991.

[La*91] Lamb, C., Landis, G., Orenstein, J., Weinreb, D., 'The ObjectStore Database System', in *Communications of the ACM*, Vol 34(10), pp 50-63, oct 1991.

[Lan91] Lang, E., 'The LILOG Ontology from a Linguistic Point of View', in *Text Understanding in LILOG*, eds. Herzog, Rollinger, Lecture Notes in AI, 546, pp 464-481, 1991.

[McS91] MacKenzie, L.E. Jr., Snodgrass, R.T., 'Evaluation of Relational Algebras Incorporating the Time Dimension in Databases', in *ACM Computing Surveys*, Vol 23 (4), pp 501-543, dec 1991.

[NgR89] Nguyen, G.T., Rieu, D., 'Schema Evolution in Object-Oriented Database Systems', in *Data and Knowledge Engineering*, Vol 4(1989), pp 43-67, 1989.

[NiT88] Nierstrasz, O.M., Tsichritzis, D.C., 'Integrated Office Systems' in *Object-Oriented Concepts, Databases and Applications,* eds. Kim, Lochovsky, pp 199-215, 1988.

[Pad86] Padgham, L., 'Linköpings Intelligent Knowledge Communication System', in *IFIP Working Conference on Methods and Tools for Office Systems*, pp 1-15, Pisa, Italy, 1986.

[Pad88] Padgham, L., 'NODE : a database for use by intelligent systems', in *Proceedings of the International Symposium on Methodologies for Intelligent Systems - ISMIS 88,* pp 190-199, Torino, Italy, 1988.

[PeS87] Penney, D.J., Stein, J., 'Class modification in the GemStone Object-Oriented DBMS', in *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications - OOPSLA 87*, pp 111-117, 1987.

[Rum88] Rumbaugh, J., 'Controlling Propagation of Operations using Attributes on Relations', in *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications - OOPSLA 88*, pp 285-296, 1988.

[Rön90] Rönnquist, R., 'A Logic for Propagation Based Characterisation of Process Behaviour', in *Proceedings of the International Symposium on Methodologies for Intelligent Systems - ISMIS 90,* pp 297-304, 1990.

[Rön92] Rönnquist, R., *Theory and Practice of Tense-bound Object References*, Ph.D. thesis, nr 270, Department of Computer Science, Linköping University, 1992.

[SkZ87] Skarra, A.H., Zdonik, S.B., 'Type Evolution in an Object Oriented Database', in *Research Directions in Object Oriented Programming,* eds. Shriver, Wegner, pp 393-415, 1987.

[SnA86] Snodgrass, R., Ahn, I., 'Temporal Databases', in *IEEE Computer,* Vol 19, no. 9, pp 35-42, September 1986.

[Sno90] Snodgrass, R., 'Temporal Databases : Status and Research Directions', in *SIGMOD RECORD*, Vol 19, no. 4, pp 83-89, December 1990.

[St*90] Stonebraker, M., Rowe, L.A., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P., Beech, D., 'Third-Generation Database System Manifesto', in *SIGMOD RECORD*, Vol 19(3), pp 31-43, 1990.

[Tei90] Teitenberg, T.N., 'Preliminary Documentation of the Application Interface Manager (AIM) for the LINCKS System', *Internal Report*, System Documentation 34.1, Department of Computer Science, Linköping University, 1990.

[WCH87] Winston, M.E., Chaffin, R., Herrmann, D., 'A Taxonomy of Part-Whole Relations', in *Cognitive Science*, Vol 11, pp 417-444, 1987.

[WKS89] Wilkes, W., Klahold, P., Schlageter, G., 'Complex and Composite Objects in CAD/CAM Databases', *Proceedings of the 5th International Conference on Data Engineering*, pp 443-450, 1989.

[Zdo86] Zdonik, S.B., 'Version Management in an Object-Oriented Database', in *Proceedings of the International Workshop on Advanced Programming Environments*, pp 405-422, Trondheim (Norway), 1986.

[Åbe89] Åberg, P., *Design of a Multiple View Presentation and Interaction Manager*, Lic. Thesis, No 177, Department of Computer Science, Linköping University, 1989.

11 Appendix

In this appendix we present the structure charts for the implemented functionality.

We use the following conventions.

- Arrows with the source filled denote flags, while otherwise they denote data flow.
- When an input parameter is changed during the execution of a function (i.e. is an in/out parameter), we have not registered this parameter also as output parameter.
- Text in italics denote already available function calls in the LINCKS system. We mentioned the main parameters.