Locking Objects and Classes in Multiversion Object-Oriented Databases

Wojciech Cellary, Waldemar Wieczerzycki
Franco-Polish School of New Information and Communication Technologies
ul. Mansfelda 4, 60-854 Poznan, POLAND
phone: (48)(61)48.34.06
e-mail: cellary@efp.poz.edu.pl, wieczerzycki@efp.poz.edu.pl

Abstract

In this paper a new approach to locking multiversion objects and classes is proposed, called the *stamp locking* approach. The main notion of this approach is a *stamp lock* defined as an extension of a classical lock in such a way that it contains the information about the position of locked nodes in the inheritance and the version derivation hierarchies. The main advantage of this method is simple locking strategy following from the lack of intentional locks concerning these hierarchies. In most cases, setting one stamp lock only, pointing to the nodes or to the subtree roots is sufficient to lock a hierarchy nodes or subtrees. To determine stamp lock compatibility, it is sufficient to compare lock modes and appropriate pointers. Advantages of the stamp locking method become particularly beneficial in the case of a database containing many classes and many versions of objects. This is a typical case of design and software engineering databases where classes, being the artifacts of the design process, have usually many superclasses and objects are available in many versions.

1. Introduction

Recently databases address non-traditional domains, such as computer aided design and management (CAD/CAM), software engineering and office information. Databases devoted to these domains need to support new functions. Most of them are provided by databases that have adopted object-oriented paradigm extended by object versions. They are called "multiversion object-oriented databases" (*MOODB*). *MOODBs*, on the one hand, support all traditional database functions, such as concurrency control, recovery, access authorization, data distribution, etc., and on the other, provide object-oriented features, such as object encapsulation and identification, class inheritance, object version derivation, etc. One of the important *MOODB* concept is a class of objects. A *class* groups objects, called its *instances*, that have the same structure and behaviour, i.e., that are defined by the same sets of attributes and methods, and recognize the same messages. When a message is sent to an instance, the corresponding method is found in the definition of the class. Classes constitute a hierarchy, called *inheritance hierarchy* that represents the *IS-A* relationship. If two nodes are linked, the lower level node is a specialization of the higher level node, and conversely, the higher level node is a generalization of the lower level node. The properties of a class, i.e. its attributes and methods, are inherited by all the classes of any lower level.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

CIKM '93 - 11/93/D.C., USA

© 1993 ACM 0-89791-626-3/93/0011 ....$1.50

In *MOODBs*, after creation of a class instance, i.e., an object, its versions may be created. From the initial object version new versions may be derived, which in turn may become the parents of next new versions, etc. Versions of an object are organized as a hierarchy, called the *version derivation hierarchy*. It captures the evolution of the object and partially orders its versions [4, 7, 8].

Classes of versionable instances require proper management. The main problem is how to maintain consistency of the database, which is related to two aspects: concurrency and versioning. The concurrency aspect of the *MOODB* consistency problem is similar to the one in traditional databases. It follows from concurrent transaction processing. Versioning aspect of database consistency concerns identification of versions of different object that go together.

The problem of concurrency control in *MOODBs* cannot be solved by the use of methods addressed to classical databases, because they do not concern new semantic relationships between classes, objects and object versions, i.e class instantiation and inheritance, and version derivation. These relationships impose additional constraints on concurrent data access. A transaction accessing a particular class virtually accesses all its instances and all its subclasses. The virtual access to an object by a transaction does not mean that this object is read or modified by it, but it means that a concurrent update of the object by other transactions has to be excluded.

In general, we distinguish two types of operations which involve inheritance subtrees: schema changes and queries. Most *MOODBs* allow more than twenty types of dynamic changes to the database schema, without requiring system shutdown or database reorganization [1]. These include adding or dropping an attribute or a method to/from a class, changing the domain of an attribute, changing the inheritance of an attribute or a method, adding or dropping a class, adding or dropping a superclass to a class, etc. The semantic of queries is impacted by the inheritance hierarchy in two ways. One is that the search space for query against a class C may be only the instances of C, or it may encompass the instances of the class subtree rooted by C. The second is that the domain D of an attribute of a class C is really the class D and all subclasses of D. This means that the search space for a query against a class includes class subtrees rooted by the domain class of each of its attributes.

Virtual access to nodes occurs also in the object version derivation hierarchy. A transaction accessing an object version may, in some cases, virtually access all the versions derived from it and come into a conflict with another one accessing a derived object version directly. Another problem is the update of an object version which is not a leaf of the derivation hierarchy. If the update has to be propagated to all

the derived object versions, the access to the whole derivation subtree by other transaction has to be excluded.

It may seem that to solve this problem the classical hierarchical locking method may be adapted [6]. As is well known, the main notion of this method is a granule, being the lockable unit. Three types of granules are distinguished: the whole database, relations and tuples, which are vertices of the granularity hierarchy. When a particular granule is locked, all the granules nested in it (i.e., its successors in the granularity hierarchy) are locked implicitly. The concept of hierarchical locking is implemented by the use of intentional locks, which make it possible to detect the conflicts between transactions on a higher level of granularity hierarchy than the level where basic locks are set. The main rule of the hierarchical locking protocol is to set intentional locks on all the predecessors of a granule being basically locked.

Two attempts to adapt the classical hierarchical locking method to the requirements of objects are known from the literature [2, 5]. Both of them concern the inheritance hierarchy and none of them concerns the version derivation hierarchy. In [2] hierarchical locking is applied to two types of granules only: a class and its instances. If an access of a transaction to a class concerns also its subclasses, all of them have to be locked explicitly by the basic locks. Their number is as big as the number of class subclasses at any level of the inheritance hierarchy. In [5] hierarchical locking is applied to the class-instance hierarchy and the class inheritance hierarchy. Before basic locking a class or a class with its subclasses, all its superclasses have to be intentionally locked. In a case of a transaction accessing the leaves of the inheritance hierarchy, it leads to a great number of intentional locks to be set.

In fact it is difficult to adapt the classical hierarchical locking method to classes of multiversion objects, because it was constructed under the assumption that there is one hierarchy only of a small depth. In object-oriented databases the inheritance and version derivation hierarchies may be composed of hundreds levels. Thus, setting intentional locks from the root to a leaf of a hierarchy via hundreds intermediary nodes cannot be efficient. From this point of view, the approaches proposed in [2] and [5] are unsatisfactory.

In this paper a new approach to hierarchical locking in *MOODBs* is proposed, called the *stamp locking* approach. It concerns inheritance and version derivation hierarchies. The main notion of this approach is a *stamp lock* defined as an extension of a classical lock. It is defined in such a way that it contains information about the position of the locked nodes in both hierarchies. Thus, in most cases, to lock related nodes or subtrees of nodes of one or both hierarchies it is sufficient to set a single stamp lock only.

The paper is organized as follows. In Section 2 the general concept of stamp locking methods is presented. In Sections 3 object versioning model is presented. In Section 4 the stamp locking method is described. Section 5 concludes the paper.

2. The Concept of Stamp Locking Methods

The concept of stamp locking methods is presented by the use of an abstract hierarchy of nodes (Fig. 1). The main idea is to extend the notion of a lock in such a way that it contains information on the position of a locked subtree in the whole hierarchy. The extended lock is called *stamp lock* and denoted *SL*.

A stamp lock is a pair:

$$SL = (lm, ns),$$

where *lm* denotes a *lock mode* and *ns* denotes a *node stamp*.

Lock modes describe the properties of stamp locks. They correspond to the lock modes used in the classical locking algorithms, namely, shared and exclusive lock modes. A node stamp is a sequence of numbers constructed in such a way that it makes it possible to determine all the node ancestors in the hierarchy. If a node is the *n*-th child of its parent whose node stamp is *p*, then the child node is stamped *p.n*. The root node is stamped *0*. For example, stamp lock *SL = (lm, 0.2)* locks, in the *lm* mode, all the nodes composing the subtree rooted by the node stamped *0.2*. This subtree of nodes is called *stamp lock scope*. In Fig. 1, the scope of *SL = (lm, 0.2)* is surrounded by the dashed line.

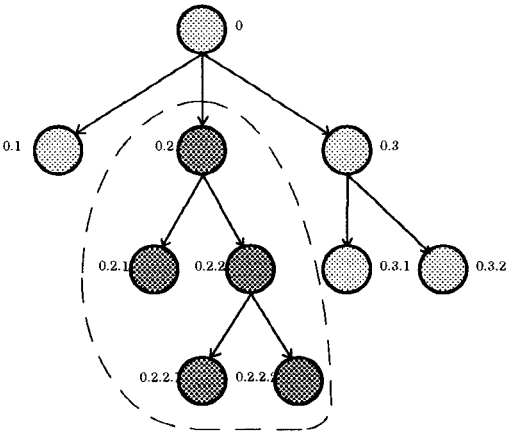

Figure 1. An abstract hierarchy

Compatibility of two stamp locks is determined by the following rule:

> *Two stamp locks are compatible if and only if their lock modes are compatible, or their scopes have no common nodes.*

Compatibility of lock modes is determined in a classical way: shared lock modes are mutually compatible, while exclusive lock modes are incompatible with both exclusive and shared lock modes. As follows from the above rule, in a case of incompatible lock modes, the corresponding stamp lock scopes have to be compared. Despite lock mode incompatibility, the stamp locks may be compatible, providing their scopes have no common nodes. All the possible relationships between the scopes of two stamp locks: SL_1 (dashed line) and SL_2 (dotted line) are given in Fig. 2. The scopes may be equal (Fig. 2a), different (Fig. 2b), the scope of SL_1 may include the scope of SL_2 (Fig. 2c), or the scope of SL_1 may be included in the scope of SL_2 (Fig. 2d).

Compatibility of stamp locks SL_1 and SL_2 may be summarized as follows: in cases a), c) and d) to be compatible the stamp lock modes have to be compatible, while in case b) stamp locks are always compatible, no matter if their lock modes are compatible or not.

To examine the relationship between the scopes of two stamp locks it is sufficient to compare their node stamps:

1. The scopes of two stamp locks SL_1 and SL_2 are equal (cf. Fig. 2a) if and only if their node stamps are identical;
2. The scope of a stamp lock SL_1 includes the scope of a SL_2 (cf. Fig. 2c) if and only if the node stamp of SL_1 is a head[1] of node stamp SL_2 ;
3. The scope of a stamp lock SL_1 is included in the scope of a SL_2 (cf. Fig. 2d) if and only if the node stamp of SL_2 is a head of node stamp SL_1;
4. The scopes of two stamp locks SL_1 and SL_2 are different (cf. Fig. 2b) if and only if cases 1,2 and 3 do not occur.

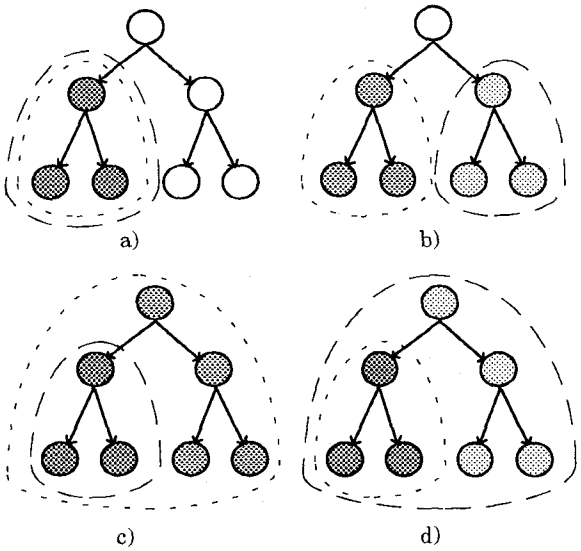

Figure 2. Comparison of scopes of two stamp locks

To compare node stamps we define three operators.

Let $ns_1 = a_1,a_2,...,a_m$ and $ns_2 = b_1,b_2,...,b_n$ be two node stamps. We say that:

- node stamp ns_1 is different from stamp ns_2 , denoted ns_1 != ns_2 , if and only if

$$m <> n \vee (m = n \wedge \exists_{i \le m} a_i <> b_i) ;$$

- node stamp ns_1 includes stamp ns_2 , denoted ns_1 >> ns_2 , if and only if

$$m > n \wedge \forall_{i \le n} a_i = b_i ;$$

- node stamp ns_1 is included in stamp ns_2 , denoted ns_1 <= ns_2 , if and only if

$$m \le n \wedge \forall_{i \le m} a_i = b_i ;$$

[1]Node stamp $ns_1 = a_1,a_2,...,a_m$ is a *head* of node stamp $ns_2 = b_1,b_2,...,b_n$ if and only if $m < n \wedge \forall_{i \le m} a_i = b_i$;

As an example consider four node stamps $ns_1 = 0$, $ns_2 = 0.1$, $ns_3 = 0.1.2$ and $ns_4 = 0.2$ (cf. Fig. 2). Node stamp ns_1 includes stamp ns_1 (ns_2 >> ns_1), because ns_1 is a head of stamp ns_2. Node stamp ns_2 is included in stamp ns_3 (ns_2 <= ns_3), because ns_2 is a head of ns_3. Node stamps ns_2 and ns_4 are different (ns_2 != ns_4), because their second elements are different. Note that from the fact that one node stamp does not include another one, we may not conclude that the first one is included in the second one. Node stamp ns_2 neither includes stamp ns_4, nor ns_4 includes ns_2 (cf. Fig 2 b).

Applying operators defined above to the node stamps composing two stamp locks, one may construct a logical rule which determines stamp lock compatibility. Considering all the stamp lock combinations it is possible to construct stamp lock compatibility matrix:

$$C_m: (SL_s, SL_r) \rightarrow R,$$

where R is a logical rule composed of operators defined above and node stamps, being their operands. Evaluating these rules makes it possible to determine the compatibility or incompatibility of locks compared. As an example consider the compatibility of two exclusive stamp locks:

$$SL_s = (X, ns_1) \quad \text{and} \quad SL_r = (X, ns_2).$$

As we know, exclusive lock modes are incompatible. Thus, the compatibility of stamp locks SL_s and SL_r may be determined by the comparison of their scopes. The scopes must be different (cf. Fig. 2 b), that means node stamp ns_1 neither includes stamp ns_2, nor it is included in stamp ns_2. We may express it by the following rule:

$$r: \quad \neg\ (ns_1 <= ns_2) \wedge \neg\ (ns_1 >> ns_2).$$

If the rule r evaluates to *true*, then the stamp locks SL_s and SL_r are compatible.

The concept of stamp locking method proposed, may be extended to more then one hierarchy, providing they are orthogonal to each other, that means the hierarchies are composed of nodes of different types. To enable simultaneous locking in many orthogonal hierarchies it is necessary to extend the notion of the stamp lock in such a way, that it contains many node stamps, corresponding to different hierarchies. As a consequence, the notion of stamp lock scope is also extended, because each orthogonal hierarchy introduces its new dimension. It is also necessary to modify logical rules determining stamp lock compatibility. In a case of n hierarchies, the problem of stamp lock scope comparison resembles finding an intersection between two figures having edges orthogonal to axis of n-dimensional space. The intersection is not empty if shadows of these figures are not distinct on each of n axis. In a similar way, we may say that the intersection between the scopes of two stamp locks is empty (i.e. it does not contain any node), if corresponding lock scopes are different in at least one hierarchy concerned. It means that two stamp locks having incompatible lock modes are compatible, if their scopes are different in at least one of the orthogonal hierarchies.

Thus, in case of two orthogonal hierarchies, the stamp lock definition has to be extended to the following triple:

$$SL = (lm, ns_1, ns_2),$$

where lm is a stamp lock mode, ns_1 and ns_2 are node stamps concerning the first and the second hierarchy, respectively. In Fig. 3 two sample hierarchies and a scope of a stamp lock SL_s = *(lm, 0.1, 0.2)* are presented.

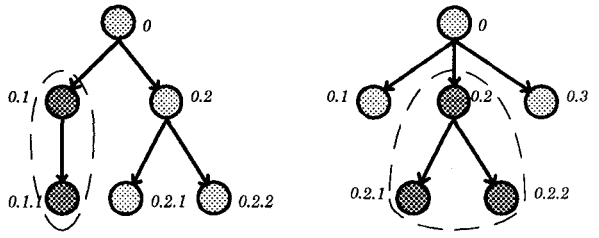

Figure 3. Scope of a stamp lock concerning two orthogonal hierarchies

To determine the compatibility of stamp locks concerning two orthogonal hierarchies we consider two following exclusive stamp locks:

$SL_s = (X, ns_{1a}, ns_{2a})$ and $SL_r = (X, ns_{1b}, ns_{2b})$.

To decide if these locks are compatible or not, one has to compare their scopes in both hierarchies. The scopes have to be different in at least one hierarchy. We may express it by the following logical rule:

$$\neg\ (ns_{1a} <= ns_{1b}) \wedge \neg\ (ns_{1a} >> ns_{1b}) \vee$$
$$\neg\ (ns_{2a} <= ns_{2b}) \wedge \neg\ (ns_{2a} >> ns_{2b}).$$

3. Database Version Approach

The concept of stamp locking may be applied to different versioning models. In the paper we assume the database version approach, which was originally introduced in [3] as a new paradigm for maintaining consistency of versions in object-oriented databases, and which is currently under implementation for the O_2 object-oriented DBMS. The main concept of this approach is that of a *database version* which comprises a version of **each** multiversion object stored in the database. Some objects may be hidden in a particular database version by the use of the *nil* values of their versions. In the database version approach, a database version is a unit of consistency and versioning. It is a unit of consistency, because each object version contained in a database version must be consistent with the versions of all the other objects contained in it. It is a unit of versioning, because an object version cannot appear outside a database version. To create a new object version, a new database version has to be created, where the new object version appears in the context of versions of all the other objects and respects the consistency constraints imposed. Database versions are logically isolated from each other, i.e., any changes made in a database version have no effect on the others.

To operate on database versions, *dbv-transactions* are used, while to operate on object versions inside database versions, *object-transactions* are used. A dbv-transaction is used to derive a new database version, called a *child*, from an existing one, called its *parent*. To derive a child means to make a logical copy of all the object versions contained in the parent. Once created, the child database version evolves independently of its parent; also its parent is not prevented from evolving if it is admitted to by the application.

To efficiently implement database versions, and to avoid version value redundancy, database versions are organized as a tree reflecting derivation history, and are identified by *version stamps*. A version stamp which is syntactically identical with node stamp (cf. Section 2) makes it possible to easily identify the path in the derivation tree from a given database version to the root, i.e., to identify all the ancestors of a given database version. A multiversion object is implemented as a set of object version values and a control data structure called *association table*. Each row of the association table of a multiversion object associates an object version value with one or several database versions. Some database versions are associated explicitly, i.e., their version stamps appear explicitly in the association table. Others are associated implicitly: if the version stamp of a database version does not appear in an association table, this means that it shares an object version value with its parent, which in turn may share it with its parent, etc. This rule gives an important advantage: to derive a new database version, it is sufficient to register its version stamp in the system. Just after derivation, this version stamp does not appear in any association table, so automatically the new database version shares version values of all the objects with its parent. As an example consider the database version derivation hierarchy given in Fig. 4. The multiversion database contains seven database versions stamped *0, 0.1, 0.1.1, 0.1.2, 0.1.3, 0.1.1.1* and *0.1.1.2* and two multiversion objects *A* and *B*. Object *A* appears in two versions: a_0 (nil version) and a_1, in database versions *0* and *0.1*, respectively. Version a_1 is shared by all the database versions composing the derivation subtree rooted by *0.1*. Object *B* appears in five versions: b_0, b_1, b_2, b_3 and b_4, in database versions *0*, *0.1*, *0.1.1*, *0.1.3* and *0.1.1.2*, respectively. The database version *0.1.2* shares object version b_1 with the database version *0.1*, while database *0.1.1.1* shares version b_2 with database *0.1.1*.

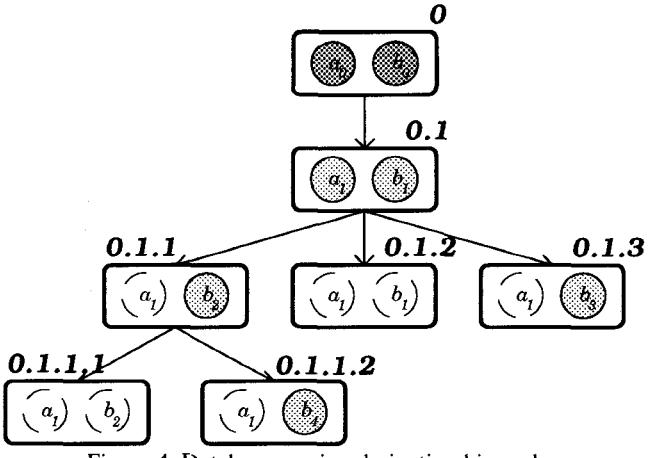

Figure 4. Database version derivation hierarchy

To update shared version value in a database version *d*, the following simple algorithm is used. First, a new row is added to the association table, associating the new version value and the version stamp of the database version *d*. Then, in the original row concerning the old version value, the version stamp of *d* is replaced by the version stamps of those of its immediate successors (children) that do not explicitly appear in any row of the association table. In this way, all the successors of the database version *d* immediate and not which implicitly shared the old version value with database version *d* will continue to be associated with it. The same algorithm is also used to create and delete objects in a database version. Creation of an object in a

database version consists in updating its *nil* version value by a given one; deletion of an object in a database version consists in updating its value by *nil*. Creation of a new multiversion object in the database consists of the creation of its association table with one row associating the *nil* version value with the root database version.

The versioning mechanism described above permits two object transactions addressed to two different database versions to run in parallel. They do not conflict and need not be serialized, even if both write the object version whose value is shared by both database versions addressed. This follows from the logical isolation of database versions: the update of a shared version value in one database version gives birth to a new one, while preserving the old one as explained above. Two object transactions addressed to the same database version are serialized in exactly the same manner as in a monoversion database.

4. Stamp Locking Method for Multiversion Objects

In this section we present a stamp locking method applied explicitly to two hierarchies: the database version derivation and the inheritance hierarchy, and implicitly to the object version derivation hierarchy. The inheritance hierarchy is orthogonal to both database version and object version derivation hierarchy because it is composed of classes, while the others of database or object versions.

The database version derivation hierarchy is constructed according to the rules presented in Section 3. To identify its single nodes, i.e. database versions, or the subtrees of its nodes, version stamps *vs* are used.

An object version derivation hierarchy binds different versions of the same object. By a "different version" we mean the one that is not shared by its parent. It may, however, be shared by its child. In other words, "different versions" are explicitly associated in the association table. There are as many different object version derivation hierarchies as the number of multiversion objects in the database. They, however, are not independent of one another. Each of them may be considered as a particular projection of the database version derivation hierarchy. Consider as an example Fig. 5. The database version derivation hierarchy is drawn by the use of the solid line, while the version derivation hierarchy for a particular object *O* is drawn by the use of the dashed line. The object version derivation hierarchy is reduced to four nodes stamped: *0, 0.1, 0.1.2.2* and *0.2*, respectively. To identify the nodes of an object version derivation hierarchy database version stamps *vs* are used.

The inheritance hierarchy binds classes, which are dealt by the method in two ways: as an abstraction of its instances and as an abstraction of its subclasses. In the first case, a class comprises its intent, i.e., its schema, and its extend, i.e., the multiversion objects being its instances. To lock both the class intent and extent it is sufficient to set one stamp lock only. One stamp lock is enough, because to update a class intent, i.e., to update its schema, the access to all its instances has to be forbidden; also to update the class extent, i.e., to update one or more instances of a class, any modification of the class schema must be excluded.

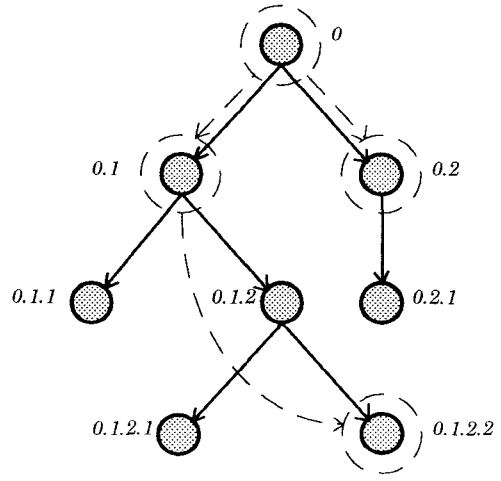

Figure 5. The database version and object version derivation hierarchies

A class considered as an abstraction of all its subclasses is a hierarchy, called the inheritance hierarchy. The method provides stamp locks which concern inheritance subtrees and all the instances of the classes included in these subtrees. To identify a single node of the inheritance hierarchy or a subtree of its nodes, *inheritance stamps* are used, which are syntactically identical with the node stamps (cf. Section 2). To distinguish between version stamps *vs* and inheritance stamps *is* we use slashes instead of dots as stamp separators. An example of an inheritance hierarchy with inheritance stamps assigned to classes is given in Fig. 6. In the database five classes *C1, C2, C3, C4* and *C5*, being the subclasses of the *Object* class are distinguished. Class *C3* is stamped *0/1/2*, which means that it is a direct subclass of a class stamped *0/1*, i.e., class *C1*, which in turn is a subclass of the root class stamped *0*.

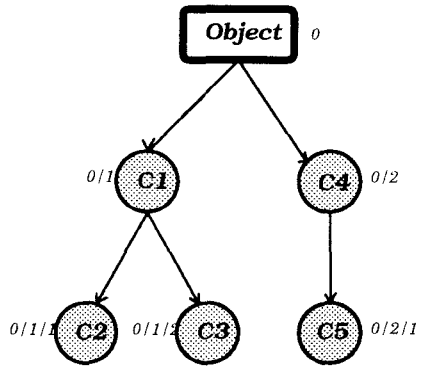

Figure 6. Class inheritance hierarchy

By the combination of particular subtrees and nodes of the inheritance, database version and object version hierarchies the following lockable granules are distinguished:

1) the whole multiversion database,
2) a subtree of database versions,
3) a single database version,
4) a class subtree in a database version subtree,
5) a class subtree in a single database version,
6) a single class in a database version subtree,

7) a single class in a single database version,
8) a multiversion object
9) a subtree of object versions,
10) a single version of a single object.

In the method, stamp locks are set on the multiversion database and multiversion objects. A lock set on the multiversion database points to nodes or node subtrees of both inheritance and database version hierarchies. Thus, it is used to lock first seven granules, which correspond to different combinations of nodes and node subtrees of both hierarchies. A lock set on a multiversion object points to its single version or a version derivation subtree. It is used to lock last three granules. Because each multiversion object is an instance of a particular class, this class must be somehow protected against locking it by the use of incompatible stamp lock on the multiversion database. To this end, before setting a lock on a multiversion object, a single intentional lock set on the multiversion database is required. This intentional stamp lock, which contains the inheritance stamp *is*, identifies the class whose instance is concerned by the corresponding basic stamp lock set on the multiversion object. Thus, there is no need to include the inheritance stamp *is* to stamp locks set on multiversion objects.

A stamp lock set on the multiversion database is now defined as a triple:

(1) $SL = (lm, vs, is)$,

while a stamp lock set on the multiversion object is defined as a pair:

(2) $SL = (lm, vs)$,

where *lm* is a lock mode, *vs* is a version stamp of the database version subtree (1) or object version subtree (2) being concerned, and *is* is an inheritance stamp of the class (class subtree) which all or particular instance is locked.

Depending on the lock mode *lm*, stamp locks may be grouped in four following ways:

- exclusive and shared stamp locks;
- non-hierarchical and hierarchical stamp locks in the database version derivation tree. Non-hierarchical stamp locks provide class or object locking in a single database version, while hierarchical stamp locks provide class or object locking in a database version derivation subtree;
- non-hierarchical and hierarchical stamp locks in the inheritance tree. Non-hierarchical stamp locks provide locking single classes, while hierarchical provide locking class subtrees, i.e., classes together with all their subclasses.
- basic and intentional locks. Basic locks are set on the multiversion database and on multiversion objects, while intentional locks are set on the multiversion database only, before setting corresponding basic locks on multiversion objects.

Twelve stamp locks set on the multiversion database are distinguished:

(x, vs, is), *(s, vs, is)*,
(xi, vs, is), *(si, vs, is)*,
(ix, vs, is), *(is, vs, is)*,

(X, vs, is), *(S, vs, is)*,
(XI, vs, is), *(SI, vs, is)*,
(IX, vs, is), *(IS, vs, is)*;

and four stamp locks set on multiversion objects:

(x, vs), *(s, vs)*,
(X, vs), *(S, vs)*,

where *x* means "exclusive", *s* means "shared", lock modes written in upper case mean "hierarchical in the database version derivation tree", while written in lower case mean "non-hierarchical in the database version derivation tree", lock modes with suffix *i* mean "hierarchical in the inheritance tree", while lock modes without suffix *i* mean "non-hierarchical in the inheritance tree", lock modes with prefix *i* mean "intentional", while lock modes without prefix *i* mean "basic".

Stamp locks *(x, vs, is)* and *(s, vs, is)* concern a single class whose inheritance stamp is *is* in a single database version stamped *vs*.

Stamp locks *(x, vs)* and *(s, vs)* concern a single object version contained in database version stamped *vs*.

Stamp locks *(X, vs, is)* and *(S, vs, is)* concern a single class whose inheritance stamp is *is* in a subtree of database versions rooted by *vs*. In the particular case of $vs = 0$, locks *(X, 0, is)* and *(S, 0, is)* concern class *is* in the whole multiversion database.

Stamp locks *(X, vs)* and *(S, vs)* concern a single object versions in a subtree of database versions rooted by *vs*. In the particular case of $vs = 0$, locks *(X, 0)* and *(S, 0)* concern the whole multiversion object.

Stamp locks *(xi, vs, is)* and *(si, vs, is)* concern a class whose inheritance stamp is *is* together with all its subclasses in a single database version stamped *vs*. In the particular case of $is = 0$, locks *(xi, vs, 0)* and *(si, vs, 0)* concern all the classes contained in a single database version, i.e. they lock a database version *vs*.

Stamp locks *(XI, vs, is)* and *(SI, vs, is)* concern a class whose inheritance stamp is *is* together with all its subclasses in a subtree of database versions rooted by *vs*. In the particular case of $vs = 0$ and $is = 0$, locks *(XI, 0, 0)* and *(SI, 0, 0)* concern all the classes in all the database versions, i.e. they lock the whole multiversion database.

Stamp locks *(ix, vs, is)* and *(is, vs, is)* express the intention of non-hierarchical locking at least one instance of a class *is* in database version *vs*, by the use of stamp locks *(x, vs)* and *(s, vs)*, respectively. Stamp locks *(ix, vs, is)* and *(is, vs, is)* protect against setting incompatible basic lock on the multiversion database.

Stamp locks *(IX, vs, is)* and *(IS, vs, is)* express the intention of hierarchical locking at least one instance of a class *is* in database version subtree *vs*, by the use of stamp locks *(X, vs)* and *(S, vs)*, respectively. Stamp locks *(IX, vs, is)* and *(IS, vs, is)* protect against setting incompatible basic lock on the multiversion database.

As an example, consider the inheritance hierarchy given in Fig. 6. Assume one instance *A* of class *C1*, two instances *B* and *C* of class *C2*, tree instaneces *D*, *E* and *F* of class *C3*, one instance *G* of class *C4* and one instance *H* of class *C5*. Transaction *T* sets stamp lock *(XI, 0.1, 0/1)*. It is hierarchical in both: the database version and the inheritance tree. Thus, it concerns class *C1* (whose inheritance stamp is *0/1*) with all its subclasses, i.e. classes *C2* and *C3*, in the database version subtree rooted by *0.1*. The scope of the stamp lock considered is surrounded in Fig. 7 by a dotted line. In Fig. 8 classes locked explicitly in both hierarchies are surrounded by a thick line, classes locked explicitly in the database version hierarchy and implicitly in the inheritance hierarchy are surrounded by a thin solid line, classes

locked implicitly in the database version hierarchy and explicitly in the inheritance hierarchy are surrounded by a dashed line, while classes locked implicitly in both hierarchies are surrounded by a dotted line.

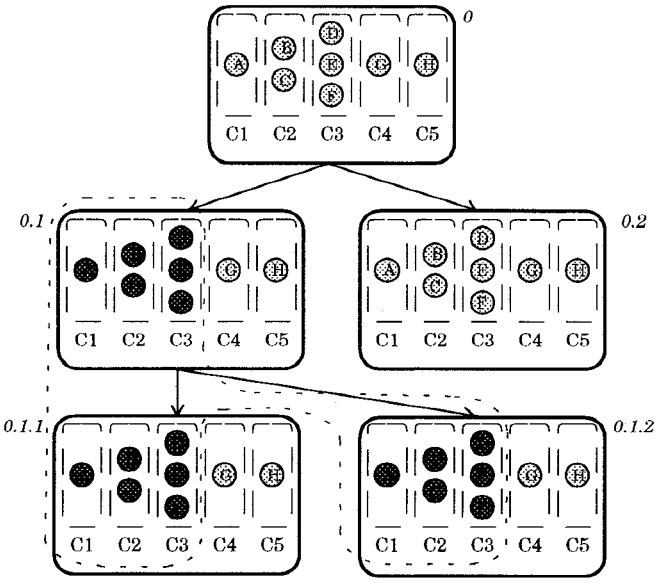

Figure 7. The scope of stamp lock *SL=(XI, 0.1, 0/1)*

Compatibility of stamp locks set on the multiversion database is determined as follows. Shared stamp locks *S, s, SI, si, IS, is* are always compatible. Intentional locks *IX, ix, IS, is* are also mutually compatible. All the other stamp locks are potentially incompatible. In the database version tree four stamp lock type combinations are possible: (non-hierarchical, non-hierarchical), (hierarchical, non-hierarchical), (non-hierarchical, hierarchical) and (hierarchical, hierarchical). Similarly, in the inheritance tree four stamp lock type combinations are possible. Because every pair of stamp locks concern simultaneously both hierarchies, 4x4 = 16 combinations of pairs: *stamp_lock_granted, stamp_lock_requested* have to be considered.

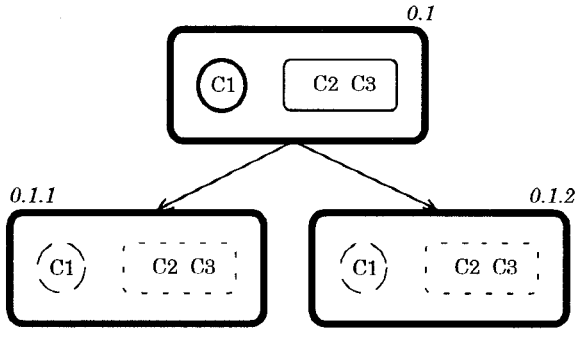

Figure 8. Classes explicitly and implicitly locked in derivation and inheritance trees

1. Both stamp locks are hierarchical in both trees: *XI* and *XI*, *SI* and *XI*, *XI* and *SI*. The scopes of these locks are subtrees of the database version tree and the inheritance tree. The scope intersection is empty if at least one of the following conditions is observed:
   - the inheritance subtree concerned by the first stamp lock does not include the inheritance subtree concerned by the second one, and vice versa, or
   - database version subtree concerned by the first stamp lock does not include the database version subtree concerned by the second one, and vice versa.

   The above conditions may be formalized in the form of the following rule:

   r1: $\neg\ (is_1 \leq\ is_2 \vee\ is_1 >> is_2) \vee\ \neg\ (vs_1 \leq\ vs_2 \vee\ vs_1 >> vs_2)$.
2. Both stamp locks are non-hierarchical in both trees: *x* and *x*, *s* and *x*, *x* and *s*, *x* and *ix*, *ix* and *x*, *s* and *ix*, *ix* and *s*, *is* and *x*, *x* and *is*. The scopes of these locks are single classes and single database versions. The scope intersection is empty if the stamp locks concern different classes or different database versions. It may be expressed by the following rule:

   r2: $is_1\ != is_2\ \vee\ \ vs_1\ != vs_2$
3. Both stamp locks are non-hierarchical in the inheritance tree and hierarchical in the database version tree: *X* and *X*, *S* and *X*, *X* and *S*, *X* and *IX*, *IX* and *X*, *X* and *IS*, *IS* and *X*, *S* and *IX*, *IX* and *S*. The scopes of these locks are single classes and subtrees of database version tree. The scope intersection is empty if the stamp locks concern different classes or the subtree of database versions concerned by the first stamp lock does not include the subtree concerned by the second one, and vice versa. It may be expressed by the following rule:

   r3: $is_1\ != is_2\ \vee\ \neg\ (vs_1 \leq\ vs_2 \vee\ vs_1 >> vs_2)$.
4. Both stamp locks are hierarchical in the inheritance tree and non-hierarchical in the database version tree: *xi* and *xi, xi* and *si, si* and *xi*. The scopes of these locks are inheritance subtrees and single database versions. The scope intersection is empty if the inheritance subtree concerned by the first stamp lock does not include the inheritance subtree concerned by the second one, and vice versa, or the stamp locks concern different database versions. It may be expressed by the following rule:

   r4: $\neg\ (is_1 \leq\ is_2 \vee\ is_1 >> is_2) \vee\ \ vs_1\ != vs_2$.
5. The first stamp lock is non-hierarchical in the inheritance tree and hierarchical in the database version tree, while the second one is non-hierarchical in both hierarchies: *X* and *x*, *X* and *ix*, *X* and *s*, *X* and *is*, *IX* and *x*, *IX* and *s*, *S* and *x*, *S* and *IX*, *IS* and *x*. The scope of the first stamp lock is a single class and a database version subtree, while the scope of the second one is a single class and a single database version. The scope intersection is empty if the stamp locks concern different classes, or the database subtree concerned by the first stamp lock does not include the database version concerned by the second one. It may be expressed by the following rule:

   r5: $is_1\ != is_2\ \vee\ \neg\ (vs_1 \leq\ vs_2)$.
6. The symmetric combination to the one presented above: first stamp lock is non-hierarchical in both trees, while the second one is non-hierarchical in the inheritance tree and hierarchical in the database version tree. The stamp lock compatibility may be expressed by the following rule:

   r6: $is_1\ != is_2\ \vee\ \neg\ (vs_2 \leq\ vs_1)$.

7. The first stamp lock is non-hierarchical in the inheritance tree and hierarchical in the database version tree, while the second one is hierarchical in both hierarchies: *X* and *XI*, *X* and *SI*, *S* and *XI*, *IX* and *XI*, *IX* and *SI*, *IS* and *XI*. The scope of the first stamp lock is a single class and a database version subtree, while the scope of the second one is a inheritance subtree and a database version subtree. The scope intersection is empty if at least one of the following conditions is observed:
   - the class concerned by the first stamp lock is not a subclass of the class concerned by the second one, or
   - the database version subtree concerned by the first stamp lock does not include the database version subtree concerned by the second one, and vice versa.

   It may be expressed by the following rule:

   r7: $\neg\ (is_2 \leq\ is_1) \vee\ \neg\ (vs_1 \leq\ vs_2 \vee\ vs_1 >> vs_2)$.
8. The symmetric combination to the one presented above. The stamp lock compatibility may be expressed by the following rule:

   r8: $\neg\ (is_1 \leq\ is_2) \vee\ \neg\ (vs_1 \leq\ vs_2 \vee\ vs_1 >> vs_2)$.
9. The first stamp lock is non-hierarchical in both hierarchies, while the second one is hierarchical in both hierarchies: *x* and *XI*, *s* and *SI*, *ix* and *XI*, *ix* and *SI*, *s* and *XI*, *is* and *XI*. The scope of the first stamp lock is a single class and a single database version, while the scope of the second one is a inheritance subtree and a database version subtree. The scope intersection is empty if the class concerned by the first stamp lock is not a subclass of the object concerned by the second lock, or the database version concerned by the first stamp lock is not included in the database version subtree concerned by the second lock. It may be expressed by the following rule:

   r9: $\neg\ (is_2 \leq\ is_1) \vee\ \neg\ (vs_2 \leq\ vs_1)$.
10. The symmetric combination to the one presented above. The stamp lock compatibility may be expressed by the following rule:

    r10: $\neg\ (is_1 \leq\ is_2) \vee\ \neg\ (vs_1 \leq\ vs_2)$.
11. The first stamp lock is non-hierarchical in the inheritance tree and hierarchical in the database tree, while the second one is hierarchical in the inheritance tree and non-hierarchical in the database version tree: *X* and *xi*, *X* and *si*, *IX* and *xi*, *IX* and *si*, *S* and *xi*, *IS* and *xi*. The scope of the first stamp lock is a single class and a database version subtree, while the scope of the second one is a inheritance subtree and a single database version. The scope intersection is empty if the class concerned by the first stamp lock is not a subclass of the class concerned by the second lock, or the database version subtree concerned by the first stamp lock does not include the database version concerned by the first lock. It may be expressed by the following rule:

    r11: $\neg\ (is_2 \leq\ is_1) \vee\ \neg\ (vs_1 \leq\ vs_2)$.
12. The symmetric combination to the one presented above. The stamp lock compatibility may be expressed by the following rule:

    r12: $\neg\ (is_1 \leq\ is_2) \vee\ \neg\ (vs_2 \leq\ vs_1)$.
13. The first stamp lock is non-hierarchical in both hierarchies, while the second one is hierarchical in the inheritance tree and non-hierarchical in the database version tree: *x* and *xi*, *x* and *si*, *ix* and *xi*, *ix* and *si*, *s* and *xi*, *is* and *xi*. The scope of the first stamp lock is a single class and a single database version, while the scope of the second one is a inheritance subtree and a single database version. The scope intersection is empty if the class concerned by the first stamp lock is not a subclass of the class concerned by the second lock, or the database versions concerned by the stamp locks are different. It may be expressed by the following rule:

    r13: $\neg\ (is_2 \leq\ is_1) \vee\ vs_1 != vs_2$.
14. The symmetric combination to the one presented above. The stamp lock compatibility may be expressed by the following rule:

    r14: $\neg\ (is_1 \leq\ is_2) \vee\ vs_1 != vs_2$.
15. The first stamp lock is hierarchical in both hierarchies, while the second one is hierarchical in the inheritance tree and non-hierarchical in the database version tree: *XI* and *xi*, *XI* and *si*, *SI* and *xi*. The scope of the first stamp lock is a inheritance subtree and a database version subtree, while the scope of the second one is a inheritance subtree and a single database version. The scope intersection is empty if one of the following conditions is observed:
    - the inheritance subtree concerned by the first stamp lock does not include the inheritance subtree concerned by the second one, and vice versa, or
    - the database version subtree concerned by the first stamp lock does not include the database version concerned by the second one.

    It may be expressed by the following rule:

    r15: $\neg\ (is_1 \leq\ is_2 \vee\ is_1 >> is_2) \vee\ \neg\ (vs_1 \leq\ vs_2)$.
16. The symmetric combination to the one presented above. The stamp lock compatibility may be expressed by the following rule:

    r16: $\neg\ (is_1 \leq\ is_2 \vee\ is_1 >> is_2) \vee\ \neg\ (vs_2 \leq\ vs_1)$.

The compatibility of stamp locks set on multiversion objects is determined in a analogous way. Shared stamp locks *S*, and *s* are always compatible. All the other stamp locks are potentially incompatible. Because stamp locks set on multiversion object concern the database version tree only, four *(stamp_lock_granted, stamp_lock_requested)* combinations have to be considered. In result, four new rules are defined:

r17: $\neg\ (vs_1 \leq\ vs_2 \vee\ vs_1 >> vs_2)$

for two hierarchical stamp locks,

r18: $vs_1 != vs_2$

for two non-hierarchical stamp locks,

r19: $\neg\ (vs_1 \leq\ vs_2)$

for hierarchical and non-hierarchical stamp locks, and

r20: $\neg\ (vs_2 \leq\ vs_1)$

for non-hierarchical and hierarchical stamp locks.

The compatibility matrix for stamp locks set on the multiversion database is given in Fig. 9 (exclusive locks requested) and in Fig. 10 (shared locks requested). The compatibility matrix for stamp locks set on multiversion objects is given in Fig. 11. In the case of compatible stamp locks the word **true** is put to the respective matrix field. In the case of potentially incompatible stamp locks a reference to rules **r1, r2, ..., r16** is put, respectively for all 16 combinations of stamp lock types set on the multiversion database, and to rules **r17, ..., r20**,

respectively for four combinations of stamp lock types set on multiversion objects.

SL_s	(IX, vs_2, is_2)	(x, vs_2, is_2)	(XI, vs_2, is_2)	(xi, vs_2, is_2)	(IX, vs_2, is_2)	(ix, vs_2, is_2)
(X, vs_1, is_1)	r3	r5	r7	r11	r3	r5
(ix, vs_1, is_1)	r6	r2	r9	r13	r6	r2
(XI, vs_1)	r8	r10	r1	r15	r8	r10
(xi, vs_1, is_1)	r12	r14	r16	r4	r12	r14
(IX, vs_1, is_1)	r3	r5	r7	r11	true	true
(ix, vs_1, is_1)	r6	r2	r9	r13	true	true
(S, vs_1, is_1)	r3	r5	r7	r11	r3	r5
(s, vs_1, is_1)	r6	r2	r9	r13	r6	r2
(SI, vs_1, is_1)	r8	r10	r1	r15	r8	r10
(si, vs_1, is_1)	r12	r14	r16	r4	r12	r14
(IS, vs_1, is_1)	r3	r5	r7	r11	true	true
(is, vs_1, is_1)	r6	r2	r9	r13	true	true

Figure 9. Lock compatibility matrix

SL_s	(X, vs_1, is_2)	(x, vs_1, is_2)	(XI, vs_1, is_2)	(xi, vs_1, is_2)	(IX, vs_1, is_2)	(ix, vs_1, is_2)
(X, vs_1, is_2)	r3	r5	r7	r11	r3	r5
(x, vs_1, is_2)	r6	r2	r9	r13	r6	r2
(XI, vs_1)	r8	r10	r1	r15	r8	r10
(xi, vs_1, is_2)	r12	r14	r16	r4	r12	r14
(IX, vs_1, is_2)	r3	r5	r7	r11	true	true
(ix, vs_1, is_2)	r6	r2	r9	r13	true	true
(S, vs_1, is_2)	true	true	true	true	true	true
(s, vs_1, is_2)	true	true	true	true	true	true
(SI, vs_1)	true	true	true	true	true	true
(si, vs_1, is_2)	true	true	true	true	true	true
(IS, vs_1)	true	true	true	true	true	true
(is, vs_1, is_2)	true	true	true	true	true	true

Figure 10. Lock compatibility matrix

$SL_s \setminus SL_r$	(X, vs_2)	(x, vs_2)	(S, vs_2)	(s, vs_2)
(X, vs_1)	r17	r19	r17	r19
(x, vs_1)	r20	r18	r20	r18
(S, vs_1)	r17	r19	true	true
(s, vs_1)	r17	r19	true	true

Figure 11. Lock compatibility matrix

To illustrate the way of determining stamp lock compatibility consider two stamp locks: $SL_s = (X, vs_1, is_1)$ and $SL_r = (X, vs_2, is_2)$. Referring to the compatibility matrix given in Fig. 9 we find out that the compatibility of stamp locks considered is determined by the rule **r1**. The rule says that stamp locks SL_s and SL_r are compatible if they concern different classes (classes having different inheritance stamps) or their scopes in the database version tree are different.

Now, consider two stamp lock pairs:

$$SL_{s1} = (IX, vs_1, is), \quad SL_{r1} = (ix, vs_2, is)$$

and

$$SL_{s2} = (X, vs_1), \quad SL_{r2} = (x, vs_2)$$

Stamp locks from the first pair, set on the multiversion database, concern the same class *is*. As follows from Fig. 9, they are compatible. Stamp locks from the second pair are set on multiversion objects, providing corresponding locks from the first pair are granted. If they concern different instances of class *is* - they are also granted. Otherwise, their compatibility is determined by rule **r19** (cf. Fig. 11). The rule says that locks may be granted if the database version vs_2 is not included in the subtree of database versions rooted vs_1.

The locking protocol depends on the size of a granule being locked and requires setting one or two stamp locks only. If a class or a class subtree in a single database version or database version subtree is locked - one stamp lock on the multiversion database is required. If a single object version or an object version subtree is locked - two the following stamp locks are required:

1. intentional stamp lock on the multiversion database with inheritance stamp *is* pointing to a single class whose particular instance will be locked,
2. basic stamp lock on the multiversion object with version stamp *vs* pointing to its single version or version subtree.

To illustrate this protocol assume a transaction T_1 which updates all the instances of class *C1* (cf. Fig. 6) and its subclasses in a single database version stamped *0.1* (cf. Fig. 7). It has to set the following stamp lock on the multiversion database:

$$SL_1 = (xi, 0.1, 0/1).$$

Now, assume transaction T_2 which updates all versions of the object *O*, being an instance of class *C4*. It has to set two following stamp locks:

$$SL_2 = (IX, 0, 0/2) \text{ and } SL_3 = (X, 0),$$

on the multiversion database and multiversion object *O*, respectively. As follows from the evaluation of rule **r12** stamp lock SL_2 is compatible with the stamp lock SL_2. Stamp lock SL_3 is the only lock set on object *O*.

Finally, consider transaction T_3 which reads a single version *C3* instance in database version *0.1*. It has to set two stamp locks. The first one:

$$SL_4 = (is, 0.1, 0/1/2)$$

will not be granted, however, because of its incompatibility with the stamp lock SL_1 set by T_1. It follows from the evaluation of rule **r14**.

6. Conclusions

The stamp locking method presented in this paper may be efficiently used in object-oriented databases to solve the problem of concurrent transaction execution. The main advantage of this method is simple locking strategy following from the lack of intentional locks concerning the inheritance and the version derivation hierarchies. In most cases, to lock hierarchy subtrees it is sufficient to set one stamp lock only, whose node stamps identify the subtree roots. To determine stamp lock compatibility, it is sufficient to compare stamp lock modes and scopes.

Advantages of the stamp locking method become particularly beneficial in the case of a database containing many calasses and many versions of objects. This is a typical case of design and software engineering databases where classes, being the artifacts of the design process, have usually many superclasses and objects are available in many versions.

Future work will be focused on taking into account the composition hierarchy in the same hierarchical way and extending the method for directed acyclic graphs instead of trees of object versions and classes.

References

[1] Banerjee J., Kim W., Kim H.J., Korth H.F., *"Semantics and Implementation of Schema Evolution in Object-Oriented Databases"*, Proc. ACM SIGMOD Conf., San Francisco, Calif., 1987.

[2] Cart M., Ferrie J. *"Integrating Concurrency Control into Object-Oriented Database System"*, EDBT Proc., Venice, Italy, 1990.

[3] Cellary W., Jomier G., *"Consistency of Versions in Object-Oriented Databases"*, Proc. 16th VLDB Conf., Brisbane, Aug. 1990, pp. 432-441.

[4] Chou H., Kim W., *"A Unifying Framework for Version Control in CAD Environment"*, 12 VLDB Conf., Kyoto, Aug. 1986.

[5] Garza J., Kim W., *"Transaction Management in an Object-Oriented Database System"*, Proc. ACM SIGMOD Conf., June 1988.

[6] Gray J. *"Notes on Database Operating Systems"*, Operating Systems: An Advanced Course, Springer-Verlag, 1978.

[7] Hubel C., Kafer W., Sutter B., *"Controlling Cooperation Through Design-Object Specification, a Database-oriented Approach"* Proc. of the European Design Automation Conf., Brussels, Belgium, 1992.

[8] Zdonik S.B. *"Version Management in an Object-Oriented Database"*, Int. Works. on Advanced Programming Environments, Norway 1986, pp. 139-200.