Database Model for Web-Based Cooperative Applications

Waldemar Wieczerzycki
Dept. of Information Technology
The Poznań University of Economics
Mansfelda 4, 60-854 Poznan, Poland
+48 61 848.05.49
wiecz@kti.ae.poznan.pl

ABSTRACT

In this paper we propose a model of a database that could become a kernel of cooperative database applications. First, we propose a new data model *CDM* (Collaborative Data Model) that is oriented for the specificity of multiuser environments, in particular: cooperation scenarios, cooperation techniques and cooperation management. Second, we propose to apply to databases supporting collaboration so called multiuser transactions. Multiuser transactions are flat transactions in which, in comparison to classical ACID transactions, the isolation property is relaxed.

Keywords

CSCW, object-oriented databases, data model, transaction model.

1. INTRODUCTION

A common feature of the majority of collaborative systems is that they require functions and mechanisms naturally available in database management systems, e.g. data persistency, access authorization, concurrency control, consistency checking and assuring, data recovery after failures, etc. Notice, however, that these functions are generally implemented in collaborative systems from scratch, without any reference to the database technology. Some systems provide gateways to classical databases, however these databases are autonomous and external to them, thus database access is organized in a conventional manner.

Since the theory and technology of classical databases is very mature, commonly accepted and verified over many years, the following question naturally arises: can we apply this technology in collaborative systems, instead of re-implementing database functions from scratch and embedding them in collaborative systems? In other words: can we develop collaborative systems as database applications, thus probably saving time normally spent on re-implementation of selected database functions? As usually we can obviously try, but there is one substantial drawback we have to take into account. The classical database paradigm assumes namely that database users are totally isolated.

CIKM '99 11/99 Kansas City, MO, USA

In such situation, in order to develop collaborative database applications, we have to extend database technology. The required extensions should be applied simultaneously to both: data modeling techniques and transaction management algorithms. Former techniques have to facilitate modeling data structures that are specific to cooperation processes, while the latter techniques have to support human interaction and exchange of non-committed data.

There are many data models proposed in the literature that are addressed to advanced domains of database applications, in particular to computer aided design (CAD) and computer aided software engineering (CASE). Most of them provide versioning mechanisms that are necessary to model: data revisions, variants and alternatives [1, 3, 4, 5, 9, 10]. These models substantially support individual design and development activities of database users. However, they do not sufficiently support group activities. It follows from the common assumption that database users communicate only via committed data. Since the users are totally isolated by the database system, each of them has an impression that the system is dedicated to him. When users collaborate to achieve a common goal, this approach is obviously too restrictive. Collaborators have to communicate directly before they agree on a data value.

As mentioned before, parallel to data model extensions, transaction model and transaction management techniques have to be extended. There are two possible directions. The first one consists in avoiding the concept of transaction and transaction management mechanisms. Non-transactional databases, however, are generally unsafe, and it is very difficult to preserve the consistency of information stored in them. The second direction aims at avoiding ACID transactions, and propose new transaction models which are more oriented for advanced database applications, especially for collaborative applications, thus preserving all advantages of transactional systems.

There are many advanced transaction models proposed in the literature [6, 7, 8, 11], There are also transaction models supporting cooperation between transactions. The most general approach proposes the cooperative transaction hierarchy [12] that allows associating transactions encompassed by a transaction group with individual designers. Taking into account the needs of generally understood collaborative work, cooperative transaction hierarchies are very promising, since transactions from the same group are not isolated mutually and can correspond to different, though somehow related tasks.

An attempt to apply hierarchical transactions to databases has some disadvantages. Contrarily to flat transactions, hierarchical transactions require sophisticated transaction management methods and, as a consequence, additional system overhead which reduces its performance. Moreover, hierarchical transactions are

still not sufficient, considering expectations of collaborating users, since in many situations the transaction correctness criterion restricts wide cooperation. Finally, they are not so reliable as flat transactions, since in practice commercial databases use the latter ones.

In this paper we propose a solution of problems mentioned above by the use of a new database model. First, we propose a new data model *CDM (Collaborative Data Model)* that is oriented for the specificity of cooperation scenarios, cooperation techniques and cooperation management. Second, we propose to apply to databases supporting collaboration so called *multiuser transactions*. Multiuser transactions are flat transactions in which, in comparison to classical ACID transactions, the isolation property is relaxed. It is worth to emphasize that both: data model and transaction model are strictly related to each other. Most of concepts used in the *CDM* model match the basic concepts of the transaction model and transaction management techniques, and vice versa.

Finally, it is also worth to underline that both the data model and transaction model have been elaborated parallel to the development of the prototype collaborative system, called *Agora* [2]. Thus, the proposed approach is not purely theoretical, but instead, it reflects the problems and solutions that occurred during the implementation of *Agora*.

The paper is structured in the following way. In section 2 the CDM data structures are proposed. In section 3 related work concerning multiuser transaction model is briefly presented. Next, the model is combined with the *CDM* model. In section 4 the *Agora* prototype is mentioned. More details concerning the *Agora* system can be found in [2]. Section 5 contains concluding remarks.

2. CDM MODEL

2.1 Basic Notions

In the *CDM* model a database is viewed as a set of domains. The *domain* is a set of database objects operated by a group of collaborating users. The users create (modify) domain objects using cooperative database applications associated with the domain. The users can also directly address domain objects by the use of ad hoc queries. In this case, however, before the first object operation, a respective domain has to be explicitly selected.

Every domain is composed of two disjoint subsets of objects: local objects and global objects. First subset contains objects that are available only in the encompassing database domain. These objects are created and modified by cooperative applications corresponding to the domain. Second subset is composed of objects simultaneously available in all database domains. In other words, the subset of global objects is the intersection of all database domains - it will be further called the *database core*.

The database core is a communication mean between database domains. It is composed of non-versionable objects containing basic information concerning the database, that can be potentially useful to all database users, no matter which domain they address (e.g. a list of domains and applications associated with them, a list of database users with the information necessary to contact them by the use of available teleconferencing tools). Moreover, the database core can store verified and commonly accepted final products of group activities that can become available to all database users, e.g. technical documentation, budget project of the enterprise.

Objects contained in the database core are read-only for the majority of database users. They can be modified only by sufficiently privileged users (i.e. database administrators), in response to requests sent by users working in different domains. Modifications of database core consist in removing and adding objects only. This approach aims at avoidance of conflicts between users who do not cooperate, i.e. users who access core objects through different domains.

The notions introduced so far are illustrated in Fig. 1. The database is composed of eight domains having a common core. Domains: *d_1* and *d_2* are assigned to cooperative application *CA1*, which is used for collaborative document writing. Users of the *d_1* domain co-author a journal paper, while users of the *d_2* domain work on a marketing leaflet. Domains: *d_4* and *d_5* are assigned to cooperative application *CA2*, which is used for collaborative software design. Users of the *d_4* domain develop a customer evidence program, while users of the *d_5* domain try to implement a program supporting finances of the enterprise. Four other domains: *d_3, d_6, d_7* and *d_8* are assigned to application *CA3*, which is used for workflow management, and thus supports business processes of the enterprise.

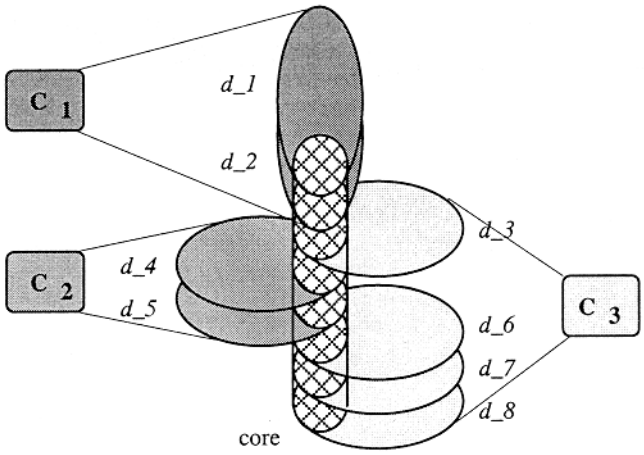

Figure 1. Database core and domains

Local domain objects can be further divided into so called domain content and domain abstract. The *domain content* groups objects created and frequently modified by team members in order to achieve the assumed outputs of cooperative work. Due to multi-stage, multi-thread and multi-variant specificity of the cooperation, the domain content can be versioned. Every version of the domain content will be further called a *context*. Thus, the domain content is the only versioning unit available in our approach. As a consequence, users perceive the domain as a set of contexts augmented by the abstract.

The *domain abstract* is as subset of non-versionable domain objects playing the role of domain content generalization. It is used to support team members assigned to the respective domain: to coordinate the cooperation, to derive commonly agreed starting assumptions, to store read-only input objects of cooperative work, to present advances in elementary task execution, etc. For example, a team co-authoring a book stores in the domain abstract: a book title, table and outline of book contents,

assumptions concerning book size and structure, reference list, figures and paragraphs taken from documents previously written that can be useful in current work (directly or after modifications). Moreover, the abstract contains also meta-texts that will not be included in the final version of a book. They are exchanged between team members for the purpose of efficient collaboration, as well as for mutual awareness and notification, e.g. comments and doubts concerning already written paragraphs, ideas concerning future work, information on recently prepared new sections.

A database composed of two domains: *D1* and *D2* is illustrated in Fig. 2. Domain *D1* contains three disjoint subsets of objects: the database core (shared with the domain *D2*), the domain abstract *A1* and the domain content *C1*. The content *C1* is versioned - in the *D1* domain the following contexts are available: *c_11, c_12, c_13* and *c_14*. Similarly, the domain *D2* contains the core, the abstract *A2* and the contents *C2*, which is available in five contexts: *c_21, c_22, c_23, c_24* and *c_25*.

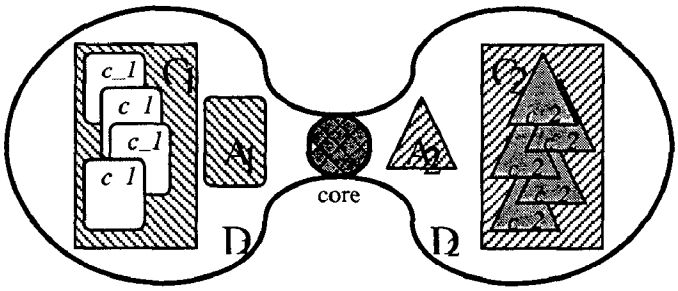

Figure 2. Abstract, content and contexts

Now we focus a bit more on domain content versioning, i.e. on the creation of new contexts in the scope of a database domain. The first context, called *root context*, is created in a particular way based on a selection of objects included in the abstract (typically abstract is created on the very beginning of cooperative work). There are two possible selection strategies. First, the indicated object may be moved from abstract to the root context. Second, the indicated object may be physically copied from the abstract to the root context. In this case a new object is created which initially has the same value as the source object.

After the root context has been created, new non-root contexts can be created by their derivation from already existing contexts. This operation consists in logical copying of all objects included in the indicated base context, providing the objects have committed values. Thus, modifications introduced by non-committed transactions are not taken into account. Next, the derived context can continue its evolution independently from the base context, due to modifications addressed to its objects. Notice, that the number of object versions included in base and derived context remains the same.

Since, except of the root context, every context is derived exactly from one base context, contexts of the same domain constitute a hierarchy. It is illustrated in Fig. 3. The domain content is composed of five contexts: *c_0, c_1, c_2, c_3* and *c_4*. The root context *c_0* contains two objects moved from the abstract, represented by a rectangle and a circle. The rectangle is available in two physical versions explicitly appearing in contexts: *c_0* and *c_2*. The circle is available in three versions appearing in contexts: *c_0, c_1* and *c_3*. Notice, that *c_4* context shares versions of both objects with its base context *c_2*.

With such assumptions concerning data model (isolated domain contents are the only versioning granules) a natural question arises that concerns the size of a consistency unit. Similarly to other versioning models, in the *CDM* model the entire database is not consistent, because in general it does not represent correctly any fragment of the real world. Thus, in the *CDM* model a single context extended by the respective abstract and database core is a unit of consistency. This assumption can only be violated by particular type of contexts that we introduce in Section 2.3.

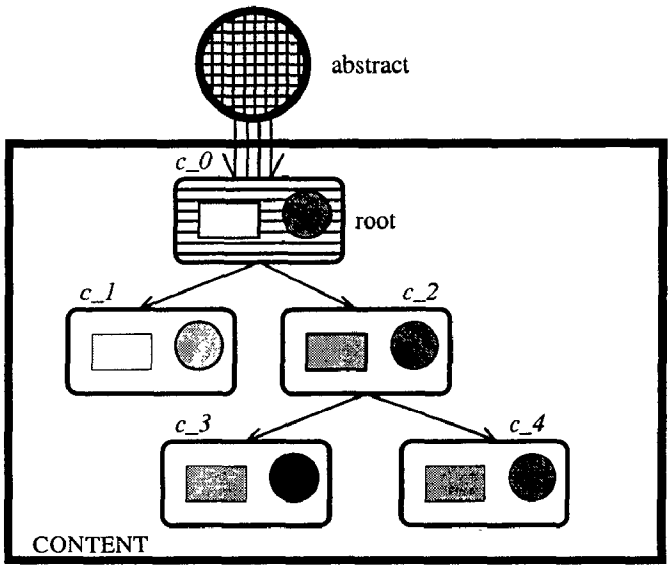

Figure 3. Hierarchy of contexts

2.2 Formal Model

In this section we define CDM model in a formal way. Let O denote a set of all database objects. This set is composed of two subsets: non-versioned objects NVO and versioned (multiversion) objects VO, such that:

$$O = NVO \cup VO \; ; \; NVO \cap VO = \varnothing$$

Every non-versioned object $NVO = (oid, val) \in NVO$ is a pair composed of object identifier *oid* and object value *val*. Every versioned object $VO = (oid, V) \in VO$ is a pair composed of object identifier *oid* and a finite set of object versions $V = \{V_1, V_2, ..., V_n\}$. Every object version $V \in V$ is in turn a pair composed of an identifier (defined later on in this subsection) and object version value *val*. In both cases, i.e. non-versioned and versioned objects, a value *val* may be composed or simple. A composed value contains at least one object identifier *oid* pointing to another object. Non-versioned object may point only to non-versioned objects, while object version may point to both versioned and non-versioned objects. A simple value does not contain object identifiers. Object $O \in O$ becomes a composite object if its versions have composite values.

Domain D is defined as a quadruple:

$$D = (did, DC, A, C),$$

composed of the domain identifier *did*, a set of non-versioned objects from the database core DC, a set of non-versioned objects

from the domain abstract A and a set of versioned objects from the domain content C, for which the following holds:

- $\mathbf{DC} \cup \mathbf{A} \subseteq \mathbf{NVO}$, $\mathbf{C} \subseteq \mathbf{VO}$,
- $\mathbf{DC} \cap \mathbf{A} = \varnothing$,
- Domains belong to the set **D**. Every pair (D_i, D_j) of different elements of this set $(i \neq j)$ fulfills the condition:
- $\mathbf{DC}_i = \mathbf{DC}_j \ \wedge \ (\mathbf{A}_i \cup \mathbf{C}_i) \cap (\mathbf{A}_j \cup \mathbf{C}_j) = \varnothing$
- Every composite objects $O_1 \in \mathbf{C}$ can point only to objects O_i $(i \neq 1)$ appearing in the same domain, i.e. $O_i \in \mathbf{DC} \cup \mathbf{A} \cup \mathbf{C}$; every composite object $O_2 \in \mathbf{A}$ can point only to objects O_j $(j \neq 2)$ appearing either in database core or abstract of the same domain, i.e. $O_j \in \mathbf{DC} \cup \mathbf{A}$; finally, every composite object $O_3 \in \mathbf{DC}$ can point only to objects O_k $(k \neq 3)$ from the database core, i.e. $O_k \in \mathbf{DC}$.

A database context CX is defined as a triple:

$$CX = (cid, D, \mathbf{ver}),$$

where *cid* denotes the context identifier, $D \in \mathbf{D}$ denotes the domain, while ***ver*** is a set of versions of objects from the content C of the domain D. Every context CX has to fulfill the following conditions:

- if $V \in \mathbf{ver}$, then object VO, with V being its version, belongs to the content C of the domain D,
- $card\ [\ C\ ] = card\ [ver]$,
- for every object belonging to the domain content $VO = (oid, V) \in C$, there is exactly one its version $V_i \in V$, which belongs to the set ***ver***, i.e. $V_i \in \mathbf{ver}$, in particular *nil* version.

Now we can define object version identifier as a pair *(oid, cid)*. As a consequence, object version is a triple $V = (oid, cid, val)$. Object version identifier *(oid, cid)* represents version identity. Thus, we consider two object versions: $V = (oid, cid, val)$ and $V' = (oid', cid', val')$ as identical, if $oid = oid'$, $cid = cid'$ and $val = val'$. Notice, that two versions of the same object, belonging to two different contexts of the same domain D are never identical. They can be at most equal, if their values are equal. Formally, two object versions: $V = (oid, cid, val)$ and $V' = (oid', cid', val')$ are equal, if $oid = oid'$ and $val = val'$.

Having basic concepts of the CDM model formally introduced, we can finally define a database as a finite set of contexts CX_i, augmented by a finite set of abstracts A_i and the database core ***DC***. The database fulfills the following conditions:

- Every non-versioned object *NVO* belongs either to the core ***DC*** or to exactly one abstract A_i;
- Every versioned object belongs to exactly one content C_i of the domain D_i;
- For every object version V there exists exactly one database context $CX = (cid, D, \mathbf{ver})$ such that $V \in \mathbf{ver}$.

2.3 Context types

The specificity of collaborative work in the database environment, in particular: long duration of database transactions, the need to store transient stages of work which sometimes may be inconsistent, as well as different levels of the cooperation intensity and scope, imply the necessity of distinguishing different types of contexts. In the *CDM* model we classify domain context in three orthogonal ways, considering respectively: context life-time, context consistency and semantic relationships between contexts.

Taking into account context life-time, we distinguish persistent contexts and temporary contexts. A *persistent context* is stored in the database directly after the commitment of a transaction that has created this context. Next, the context is accessible to all other transactions executed in the same domain. A persistent context may become a base for context derivation. It can be removed from the database only as a result of explicit delete context operation, invoked by a transaction different than the transaction that has created it.

Temporary context life-time may not exceed the duration of a transaction related to this context. We mean here a transaction that has explicitly created the context during its execution, or a transaction that has implicitly created the context, as a result of a particular database operation that requires temporary context derivation. A temporary context may be addressed only by the transaction related to it (i.e. a temporary context behaves like a private context of a single transaction). A temporary context may be promoted to a persistent context, thus gaining the features mentioned above.

Temporary contexts are used to achieve the following purposes: (1) to increase the level of concurrency among transactions, (2) to automatically merge contexts, and (3) to partially roll-back transactions. Now we will focus a bit more on these purposes.

Concurrency control. If many transactions simultaneously address the same context, then in case of incompatible operations an access conflict arises that implies a suspension or a roll-back of one of conflicting transactions. Trying to avoid conflicts one of transactions may derive a temporary context, logically move all its uncommitted operations from the base context to the temporary context, and re-address all future operation to the temporary context. As result, the level of concurrency is potentially increased. It follows from the isolation of the re-addressed transaction which execution is continued in a private context. Notice, that since a derived context (temporary context) becomes initially a logical copy of its parent, the execution environment of the re-addressed transaction is preserved. Notice also, that there is no need for lock setting in a temporary context, since it is private, thus the system overhead related to concurrency control is reduced.

Temporary contexts are particularly useful in case of read-only transaction (queries), hypothetical reasoning transactions and conditional transactions (pre-condition and post-condition). Transactions of this type usually do not modify a database or their modifications are temporary, i.e. the modifications are removed during transaction commitment. Since they are usually long-duration its reasonable to address them to a temporary context that in most cases can be simply removed from the system during transaction commitment.

Context merge. Cooperating database users sometimes deliberately decide to temporary isolate their work by addressing different (often private) contexts. Then, in agreed time moment, they decide to merge their mutual efforts by the creation of a new context, containing selected object versions from previously accessed base contexts. This process may be supported by the database system that can apply a specific merging algorithm that avoids a tedious task of object version comparison done manually by respective users. A context resulting from automatic merge

operation has to be temporary, since in general it requires users' verification (authorization), after which it may be promoted to a persistent context.

Partial transaction roll-back is related to so called save points. Defining a save point in the *CDM* model consists in deriving by the database system a new temporary, isolated and inconsistent context, directly from the context addressed by the transaction which requests save point definition. Since this context is not visible to any transaction (thus, it can not be modified), it stores a frozen image of its base context. A transaction may define many save points during its execution. Every time the same derivation mechanism is applied. If a transaction is committed, save points are useless. Thus, corresponding contexts are simply removed by the database system.

If a transaction requests a roll back, first a respective context is identified by the database system. Next all contexts derived after the identified context are automatically removed. Finally, the historical state of the base context is restored using objects from the context that represents a save point. This operation consists in copying to the base context all values of objects that have been exclusively locked by the transaction (it suggests that they could be modified after the save point definition). Other objects do not change. Earlier save points are potentially still useful, thus corresponding contexts remain unchanged.

The second way of context classification, orthogonal to the one considered above, distinguishes consistent and inconsistent contexts. As mentioned before, in *CDM* model a single context extended by a domain abstract and the database core is a unit of consistency. It concerns, however, only consistent contexts, since inconsistent contexts do not contribute in database consistency units. A *consistent context*, augmented by a respective abstract and database core, does not violate integrity constraints defined for the corresponding database domain and it reflects the expectations of users responsible for information stored in this context.

Contrarily, *inconsistent context* does not fulfill the requirements stated above. It can be created as a result of particular operations automatically performed by the database system. Furthermore, initially consistent context may also become inconsistent, as result of deliberate operations done by a database user, who is aware of context inconsistency, however, because of some reasons, decides to keep the context in the database. For example, a user writing a journal paper during many hours is aware that the paper lacks cross-references, conclusions and final review, however, he decides to keep the paper in the database and to resume his work next day, aiming at the promotion of the context into consistent one.

Classically a transaction can be addressed only to a consistent subset of data stored in the database; as result of its execution a transaction moves this subset of data from one consistent state to another consistent state. It means that classical transaction might not be addressed to inconsistent context of the *CDM* model. It also might not violate a context consistency. In such situation, in the proposed approach, besides classical transactions with the consistency property, we distinguish two particular transaction types that do not preserve this property: inconsistent transactions and verifying transactions. An *inconsistent transaction* is addressed to a consistent subset of *CDM* data (i.e. inconsistent context extended by an abstract and database core) moving it into inconsistent state. A *verifying transaction* is addressed to an inconsistent subset of data moving it into a consistent state (as result of its commitment).

The third way of context classification refers to semantic relationships between contexts. We distinguish isolated and linked contexts. Up till now, speaking about contexts we have meant only isolated contexts. An *isolated context* is logically independent from all other contexts. Logical independence concerns also isolated contexts from the same domain, in particular a base context and its descendants. Two transactions addressed to two isolated contexts never conflict, even if they operate on the same multiversion object, whose value is physically shared.

Linked contexts have at least one link explicitly defined in the database. A link is a semantic relationship of a particular type binding a pair of contexts. A link between two contexts causes in general that an execution of an operation in one context automatically triggers an execution of a derived operation in the second context. Linked contexts are particularly important for cooperative database applications. Mutual operation triggering supports mutual awareness of collaborating users, concerning the state of current work and its dynamic evolution. It also supports users' notification about the occurrence of system events that are important to the users.

Contrarily to isolated contexts, two transactions addressed to two linked contexts may fall into conflicts. To solve this problem so called multiuser transactions (cf. section 3) are used.

Links between contexts are directed. In the most general case, links are bi-directional and of different type. As a consequence, for two linked contexts *CX1* and *CX2*, operations performed in context *CX1* may trigger in context *CX2* operations different than operations triggered in context *CX1* by operations done in context *CX2*.

Links between contexts can be global or limited. *Global links* concern all context objects, while *limited links* concern only subsets of context objects.

Among basic link types one can distinguish strong and weak update propagation. Strong propagation is an extreme type of link. It consists in exact copying of updates from one context to the other. Weak propagation concerns only objects physically shared between contexts. This link can be very useful for cooperating users. As an example, consider user *U* who derives a new private version of a co-authored book and starts to modify its second chapter. The base and derived contexts are bound by a global, weak propagation link. Thus, the user *U* can observe the evolution of other chapters, because updates introduced to them by colleagues are immediately propagated to the user *U* context. There are many other types of links, i.e. propagation of object creation, removal.

In case of well-defined cooperative applications the semantics of links between contexts may be much more complex, which reflects the specificity of collaboration processes supported by the application and application functionality. For example, in case of collaborative writing application, a modification of a paragraph in a particular context, instead of update propagation, triggers only changing a color of this paragraph in linked context.

3. RELATED WORK

In this section we briefly present a transaction model especially elaborated for databases supporting web-based collaborative database applications. The model has been developed parallel to the *CDM* model. Most of the concepts used in the transaction model are strictly related to the *CDM* concepts introduced in the previous section. More details concerning the transaction model can be found in [13].

3.1 Multiuser Transactions

The multiuser transaction model is inspired by the natural perception, that a team of intensively cooperating users can be considered as a single virtual user, who has more than one brain trying to achieve the assumed goal, and more than two hands operating on keyboards.

Depending on whether database users collaborate or do not, and how tight is their collaboration, we distinguish two levels of users' grouping: conferences and teams. A *conference Ci* groups users who aim at achieving the common goal, e.g. to write a co-authored book. Users belonging to the same conference can communicate with each other and be informed about progress of common work by the use of typical conferencing tools, like message exchange, negotiation, etc. Conferences are logically independent, i.e. a user working in the scope of a single conference is not influenced by work being done in other conferences. It is possible for a single user to participate simultaneously in many conferences, thus the intersection between two conferences need not be empty. In this case, however, actions performed in one conference are logically independent from actions performed in other conferences.

Tightly collaborating users of the same conference are grouped into the same team *Ti*. Thus, a *team* is a subset of users of a corresponding conference, with the restriction that a single user *Ui* belongs in the scope of a single conference exactly to one team, in particular, to a single-user team. Of course, if he is involved in many conferences, say *n*, then he belongs to *n* teams.

A *multiuser transaction* is a flat, ordered set of database operations performed by users of the same team, which is atomic, consistent and durable. In other words, a multiuser transaction is the only unit of communication between a virtual user representing members of a single team, and the database management system.

Two multiuser transactions from two different conferences behave in the classical way, which means that they work in mutual isolation, and they are serialized by database management system. In case of access conflicts, resulting from attempts to operate on the same data item in incompatible modes, one of transactions is suspended or aborted, depending on the concurrency control policy.

Two multiuser transactions from the same conference behave in a non-classical way, which means that the isolation property is partially relaxed for them. In case of access conflicts, so called *negotiation mechanism* is triggered by DBMS, which informs users assigned to both transactions about the conflict, giving them details concerning operations which have caused it. Then, the users can consult their intended operations using conferencing mechanisms provided by the system, and negotiate on how to resolve their mutual problem. If commonly agreed, they can undertake one of actions provided by the system [13], in order to avoid access conflict. If they succeed, transactions can be continued, otherwise classical mechanisms have to be applied.

A particular mechanism is used in case of operations of the same multiuser transaction, if different users perform them, and the operations are conflicting in a classical meaning. There is no isolation between operations of different users, however in this situation so called *notification mechanism* is triggered by DBMS, which aims to keep the users assigned to the same transaction aware of operations done by other users. We have to stress that it concerns only the situation when a user accesses data previously accessed by other users, and the modes of those two accesses are incompatible in a classical meaning. After notification, users assigned to the same transaction continue their work, as if nothing happened.

3.2 Operations on Multiuser Transactions

We distinguish the following operations: initialize, commit, abort, connect, disconnect, merge and split.

initialize

Every multiuser transaction can be created explicitly by the *initialize()* operation, invoked by an arbitrary user who is a participant of at least one conference. The user, however, can not belong to any other team already working in the scope of the conference concerned, i.e. the conference in which the initialize operation is called. After transaction creation, this user becomes automatically so-called *transaction leader*.

initialize() operation can also be triggered automatically, directly after one of the members of team *Tn* performs explicit *commit(TMn)* operation, or implicit *auto-commit(TMn)* database operation. All consecutive transactions of the same team are executed in a serial order.

commit and *abort*

commit(TMn) is a classical DBMS operation that commits multiuser transaction *TMn*. *abort(TMn)* is another classical DBMS operation that aborts transaction *TMn*. These operations do not require further explanation.

connect and *disconnect*

After a multiuser transaction is initialized by the transaction leader, other team members can be attached to this transaction at any moment of the transaction execution, by the use of explicit *connect(TMn)* operation, which is performed in asynchronous manner. Once connected to the transaction, any member of a team can perform *disconnect(TMn)* operation, providing there is still at least one user assigned to the transaction *TMn*. *disconnect(TMn)* brakes the link between transaction *TMn* and the user.

merge

Transaction *TMj* can merge into transaction *TMi* by the use of *merge(TMi)* operation, providing the members of a team assigned to *TMi* allow for it. After this operation, transaction *TMj* is logically removed from the system, i.e. operation *abort(TMj)* is automatically triggered by the DBMS, and all *TMj* operations are logically re-done by transaction *TMi*. These actions are only logical, since in fact at the system level operations of *TMj* are just added to the list of *TMi* operations, and *TMi* continues its execution. However, the number of users assigned to *TMi* is now increased. It means, that until the end of *TMi* execution, the team assigned previously to *TMj* is merged into the team assigned to

TMi. Of course, *merge(Ti)* operation is only allowed in the scope of the same conference. *merge(TMi)* can be useful when an access conflict between two teams of the same conference arises.

split

Similarly to *merge* operation, *split(UT)* operation can be used in order to avoid access conflicts. Contrarily to *merge* operation that is always feasible, providing members of the other team allow it, *split(UT)* operation can be done only in particular contexts. *split(UT)* operation causes that a single multiuser transaction *TMi* is split into two transactions: *TMi* and *TMj*. After *split()* operation, a subset of team members *UT* being operation argument, originally assigned to *TMi*, is re-assigned to newly created transaction *TMj*. Also all operations performed by re-assigned users are logically removed from transaction *TMi* and redone in transaction *TMj* directly after its creation.

The two transactions resulting from *split* operation are related to each other in such a way that they can be either both committed or both aborted. It is not possible to abort one of them and commit the other, since in this case the atomicity property of the original transaction (i.e. the transaction before split operation was requested) would be violated.

As mentioned before, *split* operation can be executed in particular contexts only. Speaking very briefly, a transaction can be split if two sub-teams, which intend to separate their further actions, have operated on disjoint subsets of data, before *split* operation is requested. If the intersection between the data accessed is not empty, *split* operation is still possible, providing the two sub-teams have accessed the data in compatible modes.

Finally we combine multiuser transactions with the *CDM* model presented in section 2. It can be achieved in a very straightforward way. Every conference *C* from the transaction model is bound to a single domain of the *CDM* model. Thus conferences preserve their isolation property, since most of their data, except the database core, are physically disjoint. Remember that the database core is read-only, thus access conflicts between conferences are not possible.

Teams working in the scope of the same conference, i.e. multiuser transactions, are addressed to contexts of the same domain extended by the domain abstract. There are two possible approaches. First, teams can operate on the same database contexts. In this case access conflicts can occur frequently, thus negotiation mechanism and problem solving mechanisms presented in this section can be very useful. Second, teams can operate on different database contexts. In this case access conflicts happen rarely. They can occur when abstract objects are simultaneously accessed or when contexts addressed are bound by semantic relationships (e.g. update propagation).

4. PROTOTYPE APPLICATION

Most of the concepts introduced in previous sections have been implemented in *Agora* prototype system [2]. *Agora* is composed of two functional parts. The first one is a virtual conference tool, and the second one is a support for collaborative document writing. *Agora* provides negotiators (i.e. conference participants) with an arbitrary number of conferences and arbitrary number of collaboratively written versionable documents, with the restriction that only one document can be associated with a single conference. All negotiators discuss and present their positions by exchanging electronic messages. Each negotiator of a conference sees all the messages exchanged. A negotiator can be involved in several negotiations simultaneously, i.e. he can virtually attend different conferences. Negotiations in different conferences may concern different topics, different aspects of the same topic, or the same topic discussed by different partners.

The part of *Agora* devoted to support collaborative writing is required to prepare a final document, which is a result of negotiations [14]. This common document is seen and accessible to all the negotiators. When a negotiator writes or modifies a paragraph of the document and commits changes, it becomes instantaneously visible to other negotiators. Next, any negotiator can modify this paragraph. *Agora* provides versioning mechanisms that additionally facilitate collaborative writing.

Agora has been implemented in *Java* language and connected to the *Oracle* database management system through *Java Database Connectivity* interface (*JDBC*) to provide persistency of both documents and negotiation history. The use of *Java* and *JDBC* provides *Agora* with platform independence, concerning hardware, operating systems and database management system.

Agora architecture is client-server. Both clients and the server are implemented as *Java* objects which communicate by *Remote Method Invocation* (*RMI*). The *Agora* server is connected to a database management system by *Java Database Connectivity* interface.

Concurrency control mechanisms provided by *Oracle* DBMS are overridden in *Agora* server, what is necessary to validate the concept of multiuser transactions and new concurrency control mechanisms proposed in this paper.

Every document version is stored in one database table, thus if a document is available in *n* versions, then *n* tables have to be created. The first version of a document is entirely represented in a respective table, while in case of derived document versions only differences in comparison to the parent document version are represented, i.e. paragraphs explicitly modified in the child document version. This aims to avoid redundancy that can be really painful in case of documents having many slightly different versions.

Every paragraph of a document version is stored in a single row of a corresponding table, which is composed of a paragraph content and its layout attributes. Paragraph content is modeled by a single attribute of long raw type. It means that a paragraph can contain not only pure text but also multimedia data (pictures, sounds, etc.). Layout attributes contain typical information about the way a paragraph is visualized to the users, e.g. color, font, size, indent.

5. CONCLUSIONS

A particular model of a database supporting collaborative applications has been proposed in this paper that is based on the *CDM* data model and the multiuser transaction model. The proposed database model is very straightforward and natural, on one hand, and allows practically unrestricted collaboration among members of the same team, on the other hand. The basic assumption of this model is that collaborating users try to solve their access conflicts at a higher level than the level of a database management system, as it happens classically. The DBMS presents to the users available information on access conflicts.

Then the users can negotiate, presenting their intentions concerning future work, and choose one of proposed transaction management mechanisms which aim at conflict avoidance. During collaborative work, DBMS supports flexible versioning mechanisms and wide information exchange between team members, on one hand, and users' awareness and notification, on the other hand. These functions are extremely important in case of every cooperative system.

The proposed database model supports three typical cooperation strategies: sequential, reciprocal and parallel. These strategies applied to one of the most popular group activities - collaborative writing - are illustrated in Fig.4. In case of *sequential strategy* (Fig. 4a), a user who continues work of his co-author simply derives a new *CDM* context from colleague's context, i.e. contexts are ordered and a derivation tree is reduced to a list of contexts. In case of *reciprocal strategy* (Fig. 4b), all users address the same context. If they want to freeze a particular stage of their collaborative work, they derive a new context and re-address all their consecutive operations to this new context. In case of *parallel strategy* (Fig. 4c), users derive private contexts from the same base context. In order to support users' awareness and notification, these new contexts could be linked by mutual semantic relationships. When required, e.g. when assumed milestone of cooperation is reached, users can automatically merge the results achieved by the creation of a shared context that in general is inconsistent and requires verification transaction. Next, users can repeatedly continue the aforementioned steps towards next milestones.

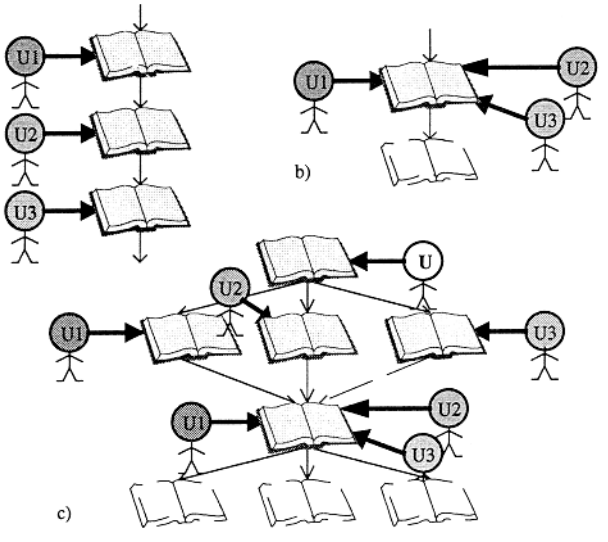

Figure 4. Different cooperation strategies

The proposed approach was verified in *Agora* prototype which is a Web-based multi-user conferencing system, offering conference participants a flexible tool for collaborative document writing. It is worth to emphasize that *Agora* is implemented on the top of a conventional and commercial database system, i.e. *Oracle* RDBMS, instead of another prototype specifically designed to support object versioning mechanisms. It makes the achieved results more reliable.

6. REFERENCES

[1] Agraval R. et al., Object Versioning in Ode, IEEE, 1991.

[2] Cellary W., Picard W., Wieczerzycki W., Web-Based Business-to-Business Negotiation Support, Proc. of the Int. Conference on Trends in Electronic Commerce TrEC-98, Hamburg, Germany, 1998.

[3] Cellary W., Vossen G. and Jomier G., Multiversion object constellations: A new approach to support a designer's database work, Engineering with Computers, vol. 10, 1994, pp. 230-244

[4] Chou H., Kim W., A Unifying Framework for Version Control in CAD Environment, Proc. of the 12 Intl. Conf. on Very Large Databases, Kyoto, 1986.

[5] Chou H. T., Kim W., Versions and Change Notification in Object-Oriented Database System, Proc. of the Design Automation Conf., 1988.

[6] Chrysanthis P.K., Ramamtitham K.: Acta - A framework for specifying and reasoning about transaction structure and behavior, Proc. of ACM-SIGMOD Int. Conference on Management of Data, 1990.

[7] Elmagarmid A. (ed.): Database Transaction Models, Morgan Kaufmann, 1992.

[8] Garcia-Molina H, Salem K.: Sagas, Proc. of the ACM Conf. on Management of Data, 1987.

[9] Katz R. H., Toward a Unified Framework for Version Modeling in Engineering Databases, ACM Computing Surveys, Vol. 22, No 4, 1990.

[10] Lamb C., Landis G., Orenstein J., Weinreb D., The ObjectStore Database System, Communications of the ACM, Vol. 34, No. 10, 1991.

[11] Moss J. E.: Nested Transactions: An Approach to Reliable Distributed Computing, The MIT Press, 1985.

[12] Nodine M., Zdonik S.: Cooperative transaction hierarchies: A transaction model to support design applications, Proc. of VLDB Conf., 1984.

[13] W. Wieczerzycki, Multiuser Transactions for Collaborative Database Applications, Proc. of 9th International Conference on Database and Expert Systems Applications - DEXA'98, Vienna, pp. 145-154, 1998.

[14] Wieczerzycki W., Advanced Transaction Management Mechanisms for Document Databases, 3rd World Conf. on Integrated Design and Process Technology - IDPT'98, Berlin, Germany, pp. 31-38, 1998.