A Model of Concurrency in Object-Oriented Databases

Daniela Rosu
McMaster University

September 2001

Abstract

The most commonly used model for concurrency control in traditional database systems represents transactions as streams of partially ordered operations that are scheduled for execution by a central or distributed transaction manager. The various scheduling strategies are proved correct by using the serializability theory. This theory, in its classical form, operates successfully on systems with flat (non-nested) transactions but is unable to represent conveniently complex (nested) computations, inherent to the object-oriented paradigm.

This report presents a more general model for transaction management which permits nested computations and a technique for proving the correctness of the schedules designed by the concurrency control mechanism. All objects in the database are assumed to have their own concurrency control mechanisms that schedule the operations local to the objects according to the order devised by the central transaction manager. The classical serializability theory is extended with an abstract model for computations allowing for arbitrary operations and nondeterministic programs.

Contents

1 Introduction

A database describes at some level of detail a part of the real world in terms of software entities (or objects) and their internal structure and interrelationships. These software entities can be recorded on any storage medium and should reflect at any point in time the state of the part of the world they represent.

1.1 Evolution of Database Models

The systems first designed for managing data and often regarded as precursors of the modern database systems were the *file management systems* which were performing basic operations on files independent of the data contained in them. Their successors are the *data definition* products which appeared in early 60's produced by companies such as IBM, General Electric and Honeywell and later transformed in simple databases that permitted the access of multiple users and used COBOL as a language for data definition and manipulation. These systems laid the foundation for what became eventually the *network* database systems which consist of *record types* related by *one-to-many* relationships. A stricter version of the network model is the *hierarchical* model where record types are organized in a tree-like structure.

An important step in providing more flexibility and physical data independence was the introduction by Codd [8] of the *relational data model*, by far the most popular database model in use. The relational model is simple and elegant with sound underlying theory based on concepts of relational algebra and first-order predicate calculus. The most important research efforts that emerged in commercial products are System R, Ingres and DB2 which made crucial steps in defining and implementing transactions, concurrency control and query mechanisms. The next sections provide a more closer look at the systems that emerged by combining the traditional database principles with to the object-oriented trend in programming languages.

1.2 Object-Oriented Database Models

Database systems were traditionally regarded as collection of objects modeled as tables or hierarchies on which transactions could perform two basic operations: *read* and *write (update)* and the relational model was one of the first database models that was not only general enough to be a powerful modeling tool but also had a solid mathematical foundation. However, the need to model an increasing number of non-standard applications has pointed out some of the major drawbacks of the relational model, namely the impossibility of defining inheritance relationships and the inability to provide a powerful modeling technique to associate complex behavior to entities at schema level. Moreover, in the relational model, the semantics of entity (object) behavior is difficult to understand and use since the attributes of a conceptual object may be distributed among many tables and there is only a limited set of primary types that can by used in defining them [9, 15, 18].

As such, one of the major goals of object-oriented data models is to provide tools for modeling complex objects, defining behavioral abstractions and introducing inheritance relationships among sets of objects. The object-oriented database management systems (OODBMSs) have many characteristics in common with the object-oriented programming languages and in fact were defined on the basis of existing object-oriented programming languages. In these systems information is organized in terms of *classes* and *instances* of these classes. Classes define the structure of data in terms of *attributes* and the *methods* used to manipulate this structure. The methods in a class can invoke other methods defined in the same class (performing local operations) or in other classes in the database (by sending messages to some instances of these other classes)[3, 4, 10]. The invoked methods can themselves call other methods, in this way causing the execution of the original method to exhibit arbitrary *nesting* .

The pioneers of the object-oriented approach to databases were based on untyped interpreted languages (e.g. Gemstone was based on the SmallTalk paradigm and Orion used Lisp's functional computing style)[25]. However, the present trend is to use the strongly-typed paradigm and static binding for security reasons (e.g. Ontos, the Vbase successor, uses an C++ interface and OO-SQL as a query language)[25].

As mentioned before, object-oriented databases merge object-oriented programming paradigms with database capabilities. The first element of this closely linked couple, *object-orientation*, appeared from the need to have a closer representation of the world around us and more modeling techniques. The new techniques allow the hiding of implementation details by *encapsulation*, *referentially sharing* objects, *schema evolution* by adding brand new elements or elements obtained by specializing existing elements through the *inheritance* mechanism, and *uniquely identifying* objects by means of unique internal labels.

The second part of the OODBMSs paradigm reflects the requests the applications had from their environment, some of the most important being the persistence of the object being manipulated and concurrent access to them.

The paradigm presented above is concisely summarized in [15] as:
Object-oriented databases = Object orientation + Database capabilities,
where
Object Orientation = Abstract data typing + Inheritance + Object identity
and
Database capabilities = Persistency + Concurrency + Transactions Recovery + Querying + Versioning + Integrity + Security + Performance.

Being an extension and a combination of two important concepts, the object-oriented databases have a huge potential that lies in the successful integration of these two technologies. It may seem a simple problem, but the practical situation is not ideal. Although object-orientation is associated theoretically with all the three notions listed above: abstract data types, inheritance and object identity, as pointed out in [15], none of the existing object-oriented programming languages completely exhibits them.

1.3 Approaches to Object-Oriented Databases

For a better understanding of the state-of-the art in the OODBMSs it is worth having a short overview of different strategies used for implementing them:

1. *Using new data model and language approach.* This is the most aggressive approach and examples of projects that employed it are FAD, Galileo and SIM (Unisys Corporation).

2. *Adding object-oriented capabilities to an existing database.* Languages such as C++ and ObjectPascal have evolved from conventional languages C and Pascal, respectively, by incorporating object-oriented concepts. Much in the same way standard database languages such as SQL can be extended with object-oriented constructs. Informix, Oracle and Borland have incorporated in their relational database systems object-oriented capabilities and Hewlett-Packard, Versant and Ontos developed their own dialects of Object-SQL [25].

3. *Adding database capabilities to an existing object-oriented language.* As a example, OPAL language extended SmallTalk with database management classes and primitives [25].

4. *Providing extendible libraries for object-oriented database management.* Ontos, Versant, Object Design and others [25] introduced C++ libraries for database management that include among other things methods for *start, abort and commit* transactions, exception handling and object clustering.

5. *Using application-specific products with an object-oriented model.* The most important application of next-generation OODBMSs is probably the *intelligent office* and OfficeIQ [15]provides just that, a persistent and concurrently shared object-oriented environment that organize not only the massive and heterogeneous amount of data but also the *flow* of information in the system.

1.4 Purpose

The problem of controlling access to shared data is common to all system that allow many programs to run concurrently. The main issues are guaranteeing that the data is kept consistent and that the user programs have a consistent view of the data. This problem has received considerable attention in database research and many theories for proving the correctness of concurrency control protocols have been developed.

The most important theory is *serializability theory* [5, 6, 7, 17] in which executions scheduled by a concurrency control protocol are considered correct if they are *equivalent* to a serial execution of the user programs submitted to the scheduler. This approach, despite its elegance, its more suitable for classical

systems and can no longer cope with the challenging issues raised by object-orientation paradigm. In the classical database systems programs have only two layers, the basic operations on data items, such as read and write a record in a table, and the transaction that issued these operations.

Starting from the premise that the serializability theory can be improved to accommodate nested transactions and user-defined commutativity and concurrency issues, this paper aims to extend the existing theory with new concepts to describe the computations in object-oriented databases and proof methods for concurrency control protocols. To achieve this goal a new conceptual model is needed that will be able to describe complex computations, equivalence of different executions and methods for proving the correctness of a given concurrency control protocol using *serializability* as a correctness criterion.

This paper follows the classical approach in that it deals with **executions** rather than with **specifications** of programs as in concurrent programming proofs. However, in our model the executions contain also the states of the objects and the values returned by the operations invoked at any object. The execution-based approach is preserved but it is the database designer/user who defines the correctness criterion and the conflict specifications.

The main component of our approach consists of a list of properties that can be used to characterize correct computations of the transactions in the database system and a set of transformations that can be used in proving the correctness of schedules generated by concurrency control protocols.

2 Transactions

The notion of *transaction* was first introduced in the context of relational databases in order to handle the concurrent access to the shared data and the fault recovery (software or hardware). The users interacted with the database by running sets of read and write operations organized in programs that received the name *transactions* to distinguish them from the programs generated with programming languages. The classical database environment appeared more than 30 years ago and research in this field was quite extensive with important fundamental results [1, 5, 7, 20, 22].

The relational, network and hierarchical systems treat transactions as streams of read and write operations with no support for complex user programs which are inherent to object-oriented systems. As such, the model of executing transactions has to be generalized by permitting *nested transactions* and by extending the set of admissible operations from *read* and *write* to a set of *arbitrary operations* [2, 3, 4, 16].

As opposed to the traditional approach to processing transactions, in object-oriented databases conflicts among execution hierarchies are determined dynamically during the execution phase. Thus, the protocols used in transaction management have to be more flexible than the ones used in traditional systems and account for *object semantics, nested executions* and *dynamic conflicts*. However, the main characteristics of transactions are valid for all database systems and represent the most important concern of any Database Transaction Manager (DTM).

2.1 ACID Properties of Transactions

The system, namely the transaction manager, guarantees that all transactions execute correctly and the database is kept consistent after any of the transactions terminates. In the literature a transaction that executed and terminated normally is referred to as a *validated* or *committed* transaction, while a transaction that did not finish normally, regardless of the type of failure (software or hardware), is said to be *aborted*.

The properties that a transaction has to have are often referred to in the literature as the ACID (Atomicity, Consistency, Isolation and Durability) properties [14, 15].

- *Atomicity:* refers to the *all or nothing* property. The *all* part of the statement refers to the fact that for a committed transaction the transaction manager arranges for all the changes that were requested by the transaction to be reflected in the current state of the database and for all the other concurrent transactions in the system to have access to those changes. If the transaction aborts because of conflicts with other transactions that are running in the same time or because of hardware failure then none of the changes has to take place — this is what *nothing* refers to.

6

- *Consistency:* a transaction has to transform a database from one consistent state to another consistent state. Consistency conditions are given by a set of rules or constraints that apply to entities in the classical models and by consistency assertions in the object-oriented database systems. Basically, in OODBMSs every object is responsible for maintaining its own consistency and the transaction manager is responsible for those actions regarding schema evolution, e.g eliminating or altering classes.

- *Isolation:* means that a transaction is executed only if there are no other concurrent transaction running in the system. In practice, this property means that a transaction's return values must be the same or equivalent to those obtained by executing that transaction without any interference with other transactions running in the system (logical isolation). The systems that require that the return values of transactions be *the same* with the ones that would be obtained if all transactions run in physical isolation are said to be *serializability enforcing*. However, in OODBMSs the environment is more flexible and the programmers can opt for defining their own equivalence assertions allowing for a degree of concurrency never attained in traditional systems. In this work we will look even deeper into the notion of concurrency control: how the responsibility for it is moved to the programmer/user side and how the availability of active objects (objects that can exhibit internal concurrency) has challenged the classical paradigms .

- *Durability:* means that the effect on the database of successful transaction will survive subsequent system failures such as disk, volatile memory or communication failure. In traditional database systems this is enforced by having the changes written first into some non-volatile memory before the transaction is committed and having them subsequently propagated to all versions of the database in use. This kind of mechanism is however difficult to implement in systems that are using a cache strategy for their clients.

If the agreement on the preservation of the above properties in the OODBMS is almost unanimous, some questions arise in terms of whether transaction management should be based or not on new paradigms that would fit better the object-oriented models. The transaction management used in the traditional databases was developed for user programs that were supposed to be relatively short, flat (with no sub-transactions) with no data exchange between concurrent transactions. As a result, the transaction manager was *batch oriented* and its techniques can no longer cope with the demands it is facing in the object-oriented environment.

2.2 Formal Definition of Transactions

From the user's point of view, a transaction is the program through which he or she manipulates the database. From the viewpoint of concurrency control

theory, a transaction is a representation of a program's execution as a sequence of operations on the database's objects.

Definition 2.1 *Two operations are said to conflict if their order of execution matters for the overall result of the transactions that issued them. Two operations that do not conflict are said to commute.*

Definition 2.2 *Given a set of operations t, a transaction is a partial order $\langle t, <_t \rangle$ such that:*

1. *t is a set of operations.*

2. *The relation $<_t$ orders at least all conflicting operations in t.*

In what follows we will refer to a transaction by using only t, when there is no confusion. As it appears in the definition, operations in a transaction are only partially ordered. This is because a program can have many operations whose order of execution does not matter as long as all strictly ordered operations are executed conform to the order defined on the transaction. For a better understanding of this concept let us look at the following example:

Example 2.1 *Consider that we have a transaction t that issues 3 operations that read values from objects x, y and z and one operation that writes the result in one of object u's attributes. The stream of operations, as they appear in the transaction's program body is:*

$t : read\_op(x); read\_op(y); read\_op(z); modify\_op(u)$

Because the programmer did not impose any restrictions on the order of the operations, all read operations can be performed concurrently on x, y and z and then the result can be finally written on u. This can be expressed in terms of partial order in the following way:

$read\_op(x) \searrow$
$read\_op(y) \longrightarrow modify\_op(z)$
$read\_op(z) \nearrow$

As it can be easily seen, an useful way of representing transactions graphically would be by using directed acyclic graphs with nodes labeled by operation names.

Note 2.1
- *Introduction of recursivity involves a more complicated treatment and is not considered in this work.*

- *The above representation of transactions does not capture every observable feature of a transaction's execution, e.g. does not specify the return values of the operations. The protocols that use this type of information do not make the subject of this work and are briefly discussed in a later section.*

2.3 Transaction Management

Transaction management or concurrency control is the activity of coordinating the access of transactions to the shared data with the goal of preventing transactions to interfere with each other. There are system in which transactions need to cooperate and exchange information, but in what follows interference among transactions is regarded as an event that must be avoided.

Throught this work we will refer to the complex mechanism performing the concurrency control as the Transaction Manager. In the literature, sometimes a distinction is made between the entity that receives transaction requests and the entity dispatching the operations for execution, but a more complex description of these entities is beyond the scope of this work and does not bring any benefit to the concepts being presented.

The solutions to the concurrency control problem are the schedules devised by the Transaction Manager so they need to be evaluated to make sure that they meet the *correctness criterion* chosen by the database designer. In traditional databases, the most common correctness criterion is *serializability-based* and the most used control mechanisms are based on *locking* [6, 7, 22].

The object-oriented paradigm has changed the classical concurrency control approaches in the sense that objects contain in their specification semantic information whose correct exploitation can lead to a better usage of the system. In this work objects are also assumed to have their internal concurrency control mechanism that ensures that the operations local to a particular object are executed in a *correct* order. The local object scheduler also ensures that executing the operations dispached by the Transaction Manager observes the consistency assertion established by the system designer or users in the case of systems that allow schema evolution. The overall design of an object-oriented system must ensure that the separately correct schedules merge into a global schedule that will be correct across all objects, without deadlocks and cyclic restarts. Because the physical location of objects is irrelevant, the same concurrency control can be used in both centralized and distributed versions of the object-oriented database.

The safest and simplest way to execute transactions is by scheduling them serially, which means that a new transaction can start executing only after the one that preceded it either committed successfully or was aborted. It is the safest way because while a transaction is executing all resource are available to it exclusively and there is no possibility of conflicting access to them. It is also the most straight forward way of scheduling transactions, but nevertheless the poorest in terms of performance (utilization of resources).

To maximize the utilization of resources the optimal way to execute transactions is by having them run concurrently with the interleaving of operations controlled by the Transaction Manager. OODBSs are particularly suitable for concurrency since every object can manage itself an associated queue of operations and in concurrent object-oriented languages the objects can have more than one internal thread of control.

However, concurrency means that there is a possibility of transaction

interference and thus consistency violation. The following types of inconsistency may arise:

- *Dirty read:* occurs when a transaction is allowed to read a value of an object's attribute that was modified by another transaction that has not yet committed and is a violation of the atomicity property.

- *Lost update:* happens when the durability property is violated in the sense that an aborted transaction t_1 that had access to data modified by another transaction t_2 will erase by its roll-back the previous modifications thus canceling the duration property of t_2.

- *Unrepeatable read:* occurs when a transaction that re-reads a data item will get a different value from the one that was read first because meantime another transaction had access to that data item and modified it.

The inconsistencies listed above need to be eliminated by regulating the way transactions interfere in the system. This is the job of the transaction manager or of the central transaction manager and individual object transaction managers in the case of OODBMSs. The regulating process should ensure that a transaction is allowed to access a data item only when there is no possibility of interference with other concurrent transactions. The most common regulatory method is known as *serialization process*.

2.4 Correctness Criteria

The traditional serializability theory defines the correct executions to be those equivalent to the *serial* executions of the same transactions and provides a graph-based tool for identifying executions that meet the requirements of the correctness criterion.

General-Serializability and Non-Serializability Correctness Criteria
Strict serializability-based correctness criteria are an elegant and efficient way of verifying that both the data consistency and transaction scheduling correctness requirements are met. In an effort to overcome the inherent limitations of this traditional concurrency model, many researchers have come up with alternative correctness criteria, including *abstract serializability* and *relaxation of serializability* [23, 24].

The former uses different definitions for conflicts among operations (e.g. serial dependency and recoverability). Badrinath and Ramamrithan [4], use the semantic information about operations to identify the property they call *recoverability*. In their model two operations that conventionally do not commute but whose respective results are can be recovered in case that one of them aborts are said to be recoverable with respect to each other. Two recoverable operations are allowed to execute in parallel, but the transactions invoking these operations must commit in an order consistent with the order in which the operations are performed. If $operation_i$ of transaction t_i is executed after $operation_j$ of transaction t_j and $operation_j$ is *recoverable* relative

to $operation_i$, then t_i must commit before t_j. The term *recoverability* is also discussed by Hadzilacos [10] and Bernstein et al. [6], their models imposing that no transaction commits before any transaction on which it depends.

In [6], *recoverability* is defined as following: for any two operations o_1 and o_2, if o_2 immediately follows o_1 and $return(o_2, state(o_1, s)) = return(o_2, s)$ then o_2 is said to be *recoverable* relative to operation o_1. $return(o, s)$ is the return value produced by operation o applied to a data item in state s and $state(o, s)$ is the state of a data item, initially in state s, after operation o is applied to it. The recoverability property introduces a *commit-dependency* relation on the set of operations available on objects which can be represented by a directed graph. The correctness criterion in this model requires that the commit dependency graph is acyclic, which is equivalent to ensuring the serializability of transaction executions.

Another approach towards incorporating semantic information in correctness criteria is due to Agrawal et al. [2] who relaxed the *absolute atomicity* property of transactions using the notion of *relative atomicity*. The novelty of this approach consists in partitioning each transaction in *atomic units*, sequences of consecutive operations that must be executed observing the strict atomicity property discussed in the introductory chapter. For all transactions in the system it is constructed a *relative atomicity specification matrix A* whose elements a_{ij} are sets, each representing a partition in *atomic units* of transaction t_i relative to transaction t_j. The elements of a_{ij} can be represented as $a_{ij}^1, a_{ij}^2, \ldots, a_{ij}^n$, where a_{ij}^k is the *kth* atomic unit of transaction t_i relative to transaction t_j and represents a set of consecutive operations of t_i that cannot be interleaved with any operation of t_j.

The correctness criterion is *relative serializability* which states that a schedule of all transactions in the system is *relatively serial*, thus correct, if for any transactions t_i and t_j, if an operation o of t_i is interleaved with an atomic unit a_{ji}^k for some k, then o does not depend on any operation p in the atomic unit a_{ji}^k and any operation q in the atomic unit a_{ji}^k does not depend on o. The schedules considered correct are those equivalent to some relatively serial schedules and are said to be *relatively serializable*. When relative atomicity is restricted to strict atomicity the above criterion becomes the serializability correctness criterion.

2.5   Concurrency Control Strategies

All correctness criteria detailed above have their direct application in transaction management, more specifically in proving the soundness of different techniques for controlling concurrency in the system. Traditionally, concurrency control strategies are classified in the literature as being either *optimistic* or *pessimistic*.

Optimistic Scheduling
Optimistic scheduling is based on the idea that user transactions are running on copies of the data without asking for access permission [11, 19]. As such, a transaction is allowed to start performing its operations without verifying whether

there are conflicts of accessing shared data with other transactions running in the system. However, before committing, a transaction must pass a validation checking for the changes it initiated to take place in the real database. This test is done usually by *backward validation*, which means that the Transaction Manager checks that the changes initiated by the transaction that requested the commit **were** not invalidated by other concurrent transactions. Another possible approach is *forward validation*, in which it is checked whether the changes initiated by the transaction **will** invalidate results of other concurrent transactions. If a transaction passes the test then it is committed and its changes become permanent. If the test fails, the transaction is aborted and later restarted.

Because of the relatively high probability of aborting transactions this strategy is useful if:

- There is little data sharing among transactions.

- It can be tolerated that transactions have to run several times before committing.

The probability that a transaction is in conflict with other transactions increases with the length of the transaction and with the number of objects being accessed. Thus, the optimistic concurrency control is cost-effective only when the probability of aborting transactions is low. Pessimistic techniques are more robust than the optimistic ones, but it must be stressed that their increase in the rate of transactions successfully committed is attained by making other transactions wait, thus increasing the average time for successful completion.

Pessimistic Scheduling
Pessimistic methods do not permit any transaction to access data before the system checks that there is no other transaction that requested access to that data. If the access requested cannot be granted, the transaction is blocked in a waiting list. Most pessimistic concurrency control algorithms use locking mechanisms for synchronizing the transactions running in the system. Two of the simplest and mostly used locking types in traditional databases are *read lock* and *write lock*.

The read lock is a shared one in the sense that it allows multiple transactions to access the data for reading or in terms of objects to perform methods that only retrieve the values of certain attributes and do not alter them. The write lock grants exclusive access to data so that only one transaction at a time can perform operations that imply changes in the values of attributes.

3 A Concurrency Model for OODBMSs

The classical serializability theory has been used successfully to prove the correctness of most concurrency algorithms used in traditional databases, but needs to undergo some changes in order to accommodate the nested computations inherent to OODBMSs.

3.1 Related work

Previous research that has focused on the theory of nested transactions assumes only centralized versions of a Database Transaction Manager (DTM), or at most a distributed version of a DTM, while in OODB systems each object has its own scheduler responsible for maintaining the consistency of the object. Even further, there are no explicit integrity constraints that can be checked when assessing the consistency of the database. As such, all methods are assumed to produce correct results and preserve the consistency of the database when run atomically or observing the concurrency rules imposed by the programmer.

Nested transactions as defined for traditional databases have received much attention and several important models have been proposed, some of them using decomposition into levels, others developing new proof techniques. All approaches model transactions computations as trees or forests but the techniques used in proving the correctness of certain protocols vary substantially.

This paper extends the model developed by Beeri at al. [5] for nested transactions in traditional databases. In their model transactions are modeled as trees of subsequent method calls and the Transaction Manager it is assumed to work at all levels of the computation and there is no support for the object-oriented paradigm. We extended the model introducing the object-oriented concepts and allowing each object to perform its own concurrency control compatible to the general strategy devised by the central Transaction Manager. Our model also operates with methods (operations) as opposed to approaches that explicitly model the initiation and completion of an operation.

3.2 The Model

In this paper, an *object-oriented database management system* is defined as a collection of classes and instances of these classes. A *class* contains the definitions of the variables that will take values in the instances of this class together with the methods used to access these variables. A *method execution* is considered to be a partial order of submethod calls. It is assumed that the database environment allows for *extensibility*, permitting users to dynamically modify the class definitions. As such, we can regard class definitions as objects accessible by the users of the database. Both the *class objects* and the *instance objects* (objects derived by instantiating a class object) will be referred to as *objects* thus providing uniform treatment for all objects in the database.

The objects are assumed to be autonomous entities, internally concurrent, with full control over the methods they are running at any time. They

are also assumed to be organized in a hierarchy. Relationships among methods are described in terms of standard tree terminology (recursive method calls are not allowed)with a method *m2* being a *child* of a method *m1* if *m1* invokes *m2*. The notions of *parent* and *ancestor* are defined in the same manner. In this environment the methods of one object can invoke only methods of objects that are lower the hierarchy and every object inherits the methods of all its ancestors.

As mentioned before, in a OODBMS a *transaction* (a user program) consists of a series of method invocations on different objects, which in turn can invoke other methods on different objects, this leading to a tree structure of method calls. As in traditional database systems a transaction is represented as a partially ordered set of method calls which are related among them by conflict, commutativity and concurrency conditions specified by the OODBMS designer or by the users who extend the database with new objects.

Assumption:
(1) Methods on different objects are regarded as non-conflicting, unless otherwise specified.

In every object, each method is implemented as a sequence of methods local to the object or possibly invoked on other objects, referred to using the tree terminology as *lower level* methods. However, in a different transaction execution tree the methods that are regarded in the current transaction as descendents can occur at a higher level than the methods currently invoking them.

3.3 Transaction Management

Synchronization in an OODBMS is a common effort of the central DTM and of local DTMs corresponding to each object in the database and as such extending the serializability theory from classical and multi-layered nested systems to the object-oriented systems is a challenging problem. Two of the most important aspects to be dealt with are the absence of explicit integrity constraints and the need to design a mechanism through which the central scheduler and the object schedulers can communicate and synchronize their decisions.

One of the first problems that arise when switching conceptually from the classical paradigms to the object-oriented paradigm is the concept of *commutativity*, used as a means of conveniently reordering the operations. In the classical theory commutativity can be defined as a static property, known from the beginning of the database life and valid until the end, but in the object-oriented world commutativity is a state-dependent property.

Definition 3.1 *Two method invokations on the same object* commute *if and only the return values of those two methods does not depend on the order in which they are executed.*

The model used in this paper allows methods to run concurrently do the *commutativity* property can be restated as follows:

Definition 3.2 *Two methods m_1 and m_2 are* concurrent *if all the following three executions are equivalent: m_1 followed by m_2, m_2 followed by m_1 and m_1 and m_2 simultaneously.*

The above definition corresponds to paradigm π_8 defined by Janicki and Koutny in [12,13]. It is further assumed that the commutativity relationship between the operations can be derived based on the specifications contained in class objects corresponding to the instance objects on which these operations are to be performed.

Definition 3.3 *If two operations do not commute they are said to* conflict *with each other.*

The most important implication of this definition of commutativity is that in OODBMSs the reordering of lower level methods can be performed in a more flexible manner. The simplest approach is to reverse the order of the commuting ancestors and the whole subtrees underlying their computation. The above definition also allows a reordering that permits certain interleavings.

classical approach   under certain conditions in the new model
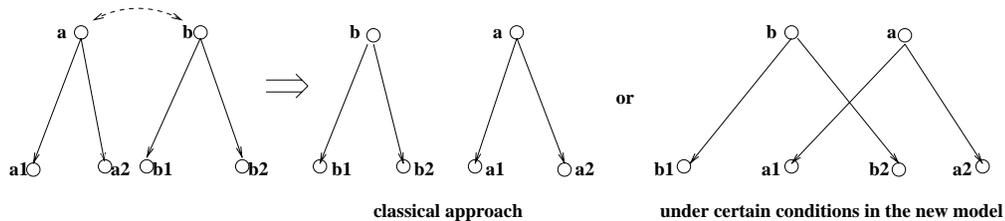
Figure 1:

Also in contrast to the classical theory — which uses only transactions, methods and their partial order and interference (commutativity or conflict)— the model presented in this paper will also use the states of the objects and the value returned by the invoked methods.

All user programs are directed to a central DTM that will dispatch the method calls to the objects involved in computation. As such, the DTM functions as an interface between the persistent objects stored in the database and the users. As user transactions may run concurrently and some of the possible computations generated by different orderings of method executions can be incorrect, there is a need for the central DTM to schedule the main flow of control. The object-based DTMs have the responsibility to execute the method calls they receive in a order consistent with the ordering devised by the central DTM and consistent in the same time with the synchronization constraints local to the object.

This paper uses the most common correctness criterion for concurrency control, the *serializability* property. A serial execution of the leafs of a method means technically that the respective method runs atomically with respect to the other methods and the whole subtree can be pruned from the complete tree of the computation. As such, the computations (schedules of transaction executions) accepted as *correct* are those equivalent to *serial computations*. The proof paradigm is to define a set of general transformation whose iterative application on a given computation may result in a serial computation. If the transformed computation resulted after the transformation process is a serial one, the initial computation is regarded as correct.

The role of the central and object-based schedulers is to restrict the allowable computations to a subset of the correct ones. This role is accomplished by the schedulers by allowing only computations that exhibit certain properties that can guarantee that computations observing them can be transformed to equivalent serial computations.

An execution (computation) is considered to order its subsequent method calls in a partial-ordered manner and the commutativity and the concurrently runnable properties of methods are used to serialize the leaves of the computational tree.Basically, after each pruning of a subtree the computational tree becomes less nested and easier to reason about. If the serialization process reaches the root then one can claim that the transaction whose computation has been serialized is equivalent to a serial execution and thus is correct.

The main difficulty in synchronizing transactions in object-oriented databases resides in the fact that simply serializing executions on every object will not necessarily imply that the transaction as a whole is serial. It might happen that two objects schedule their local method calls corresponding to different serial orders of the transactions that issued them, thus making it impossible to establish a global serial order of the transactions in the database.

In [5] it is assumed that the central DTM functions at every level of the computation but this is no longer true in the model presented in this paper because each method is dispatched to an object and it is scheduled locally. The scheduling algorithm at object level needs to be compatible to the schedule as seen by the central DTM and the conditions that need to be met are specified in the concurrency protocol that will be discussed later on.

Example 3.1 *Assume that methods a and b conflict and a must execute before b, but a_2 and b_1 are concurrently runnable. As such, instead of executing all submethods serially, i.e. $a_1 \rightarrow a_2 \rightarrow b_2 \rightarrow b_2$ the system may run $a_1 \rightarrow a_2 \parallel b_1 \rightarrow b_2$, where $a_2 \parallel b_1$ means that a_2 and 4 b_1 execute concurrently.*

3.4 Computational Models

As stated at the very beginning the notion of transaction is central to the theory of concurrency in databases, but in our model it is the definition of methods
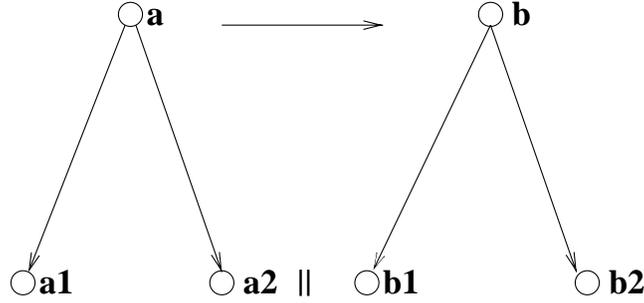
Figure 2:

that lies at the basis of all concepts and constructs that will be introduced in this chapter.

In what follows we will refer to the set of return values as the set of allowable return values for the functions as defined in the language used for accessing the database, augmented with the NULL value that means that the function does not return any value.

Definition 3.4 *A method call is represented by a tuple* $(m_{id}, m_{parent_{id}}, s_{initial,o}^{m_{id}}, s_{final,o}^{m_{id}}, <^{m_{id}}, r^{m_{id}})$, *where:*
(1) m_{id} is the method's id;
(2) $m_{parent_{id}}$ is the id of the method invoking m_{id};
(3) $s_{initial,o}^{m_{id}}$ and $s_{final,o}^{m_{id}}$ are the states of the object o processing the method prior and after its execution, respectively;
(4) r^{id} is the value returned by m_{id};
(5) $<^{id}$ is the partial order of the submethods in the computation tree of m_{id}.

We use m_{id} as an abreviation for the tuple describing a call for a method with id m_{id}.

Definition 3.5 *Let M be the set of all method calls. For m_1 and m_2 method calls in M, m_1 is said to precede m_2 in M, denoted by $m_1 < m_2$, if m_1 must execute before m_2.*

Example 3.2 *Let m be a method call with 2 direct descendents m_1 and m_2, where m_1 is a leaf method and m_2 is a complex method with 2 submethods m_{21} and m_{22}.*

The set of methods that defines the computation of m is $S_m = \{m, m_1, m_2, m_{21}, m_{22}\}$. Suppose that the only restriction imposed by the programmer is $m_1 < m_2$. Thus, the partial order describing a possible execution of method m is $\{m < m_1, m < m_2, m_1 < m_2, m_1 < m_{21}, m_1 < m_{22}\}$.
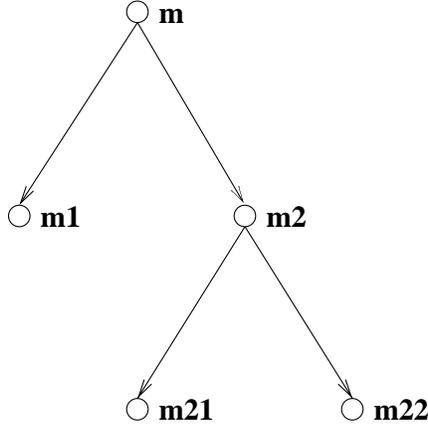
Figure 3:

Let $tree(m) = (S_m, \rightarrow)$ be a tree that defines the method m, where S_m is the set of all submethods, plus the root.

The set $leaves(m)$ of all leaf methods of m is $\{m_1, m_{21}, m_{22}\}$. Let $(leaves(m), <^{leaves(m)})$ be a partial order such that $<^{Leafs(m)}$ orders all conflicting leafs. Then $<^m |_{leaves(m)} = <^{leaves(m)}$.

Let us use m as an abreviation for $(tree(m), <_m)$, where $<_m$ is a partial order satisfying the following properties:

(a) $<_m |_{leaves(m)} = <^m_{leaves(m)}$;

(b) $\forall x, y \in S_m$, if $x <_m y$ then there exists submethods s_x and s_y of x and y, respectivelly such that $s_x <_m s_y$.

Considering M to be the set of all methods let us define $tree(M) = \underset{m \in M}{\cup} leaves(m)$. Let $S_M = \underset{m \in M}{\cup} S_m$. The partial order $<$ on a transaction over the set M of methods is defined such that:

1. $< |_{S_m} = <_m$

2. $<$ orders at least all conflicting elements of $\underset{m \in M}{\cup} leaves(m)$.

The central DTM and all object level DTM will restrict the partial order $<$ from $(M, <)$ to a total order preserving all *precede* restrictions. This property is usually referred to in the literature as the *order preserving atomic* property. The *order preserving* part describes the requirement of preserving all $m_i < m_j$ pairs and *atomic* means that when exchanging the order of two commuting operations (methods) the whole subtrees underlying their computations are exchanged in the global tree.

A1 (Atomicity preservation) All computations allowed by the DTM are order preserving atomic.

Intuitively, the tuples generated by the *precede* relation cannot be changed or deleted from the total ordered execution of the global tree without affecting the control structure intended by the users. The leaf method calls are executed by the objects on which they are invoked and return a value (a proper value or a NULL type value) and when all leafs of an internal node have completed, the method call that labels the node completes itself returning a value and so on up to the root.

A2 (Downward order compatibility) Let m_1 and m_2 be two methods that are internal nodes in the global tree. If m_1 precedes m_2 in the flow of control of one transaction then all descendants of m_1 must precede all descendants of m_2 at all objects invoked in their computation.

Intuitively, Axiom A2, states that the time ordering as devised by the programmer be respected and this can be expressed formally as follows: let $m_j = (m_{j-1}, s^{m_j}_{initial,o}, s^{m_j}_{final,o}, <^{m_j}, r^{m_j})$ be a method labeling an internal node in the general computational tree and $m_{j_1} = (m_j, s^{m_{j_1}}_{initial,o_{l_{j_1}}}, s^{m_{j_1}}_{final,o_{l_{j_1}}}, <^{m_{j_1}}, r^{m_{j_1}})$ and $m_{j_2} = (m_j, s^{m_{j_2}}_{initial,o_{l_{j_2}}}, s^{m_{j_2}}_{final,o_{l_{j_2}}}, <^{m_{j_2}}, r^{m_{j_2}})$ two of its submethods. If for any submethods $m^{j_1}_p$ of m_{j_1} and $m^{j_2}_q$ of m_{j_2}, $m^{j_1}_p$ precede $m^{j_2}_q$ then in the partial order describing the general computation tree it can be added that $m_{j_1} <^{m_j} m_{j_2}$ (m_{j_1} precede m_{j_2}) without altering its acyclicity property.

Definition 3.6 *A transaction is described by a tuple* $(M, <)$ *where* M *is the set of methods composing the computation tree of transaction* M *and* $<$ *is the partial order relating the method calls in the tree.*

Definition 3.7 *An execution of transaction* $(M, <)$ *is serial if* $<$ *is a total order.*

Assumption: The number of transactions in the system and the number of method calls in a transactions are finite.
        This assumption allows the proof of serializability to consider that all transactions submitted to the central scheduler are immediate descendants of an initial transaction that initialized the system, thus the following proofs will ensure the existence of a single computational tree. In this way, the correctness of a schedule is equivalent to proving the existence of a total order on the global tree.

3.5   Correctness of Computations

Definition 3.8 *Two transactions (computations) are equivalent if starting in the same initial state they leave all the objects involved in equivalent final states.*

Note 3.1
The programmer may state for each object a set of equivalence states with the purpose of defining state-based commutativity *and* concurrently runnable *assertions.*

Example 3.3 *In a bounded buffer with capacity* n [21], *all possible combinations of less than* n *elements make up the equivalence class* partial *and all combinations (states) with* n *elements fall in the* full *class.*

This treatment of states accommodates both the explicit states assertions and *condition-defined* states approaches.

Definition 3.9 *(Separation)* A node m (a method call m) is *separated* from the rest of the nodes in the general tree if its submethod calls at all objects involved in its computation are executed without being interleaved with method calls whose least common ancestor with m is at higher level than m.

Using the above definition, a serial execution is an execution in which all the methods in its computational tree are separated.

Example 3.4 *The following example presents an execution in which one of the methods is not separated in the general tree.*
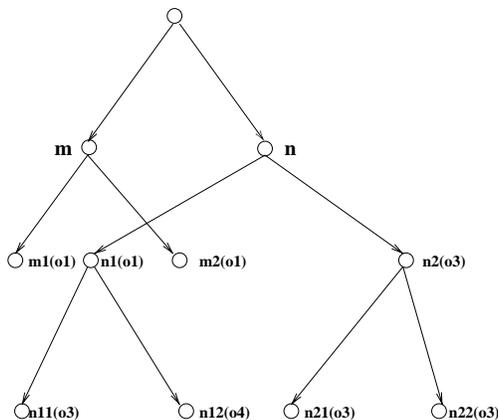

Figure 4:

The execution of m_1 and m_2 at object o_1 is interleaved with n_1 which is a descendent of n. *The least common ancestor of* m *and n_1 is at a higher level than* m *in the general tree. However, in an alternative execution method* m *can be separated:*
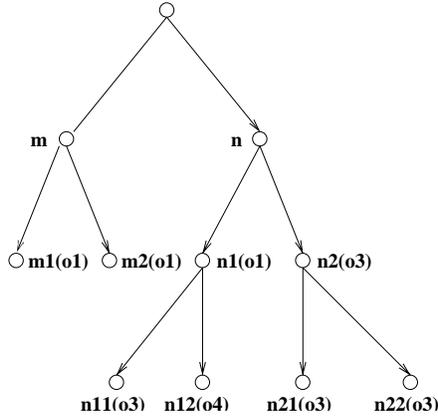
Figure 5:

Definition 3.10 *(2-method separation) Two method executions are separated if their descendants are not interleaved at any object involved in their computation.*

Note 3.2

    1. *2-method separation does not imply that the two method executions which are separated from each other are separated from the rest of the methods in the general tree. It can be that a third method execution is interleaved with both of them on some objects.*

    2. *interleaving among descendants of the same method calls can occur under certain conditions.*

Definition 3.11 *(Serialization) A method execution is serialized if all its subtrees are executed serially, which is equivalent to saying that there is a total order on its set of method calls.*

Commutativity

    Commutativity is the most important property of two methods and it is the basis of the most powerful serializing technique, which consists of reversing the order of commuting operations. As it will be used in this paper, commutativity is a state-based property, user-defined and local to an object.

        Intuitively, by repeatedly applying the commutativity property at object level one can serialize the whole general tree. Thus the initial computation that takes advantage of the internal concurrency of objects and of the opportunity of running descendent methods simultaneously at different objects can be proved *correct*.

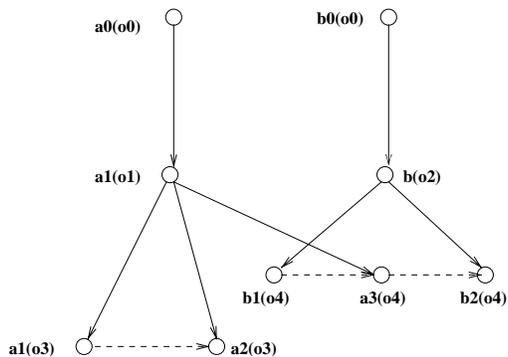**Example 3.5** *Let b_1 and a_3 be two commuting methods object o_4.*

Figure 6:

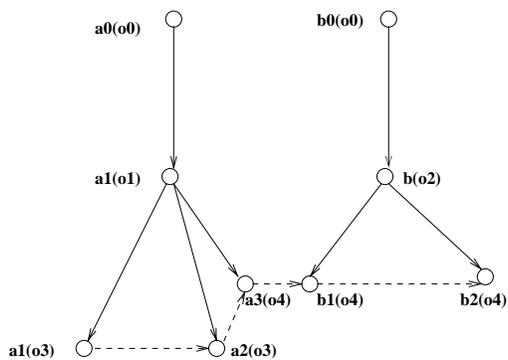*Then their order of execution can be reversed leading to*

Figure 7:

which is a serial execution not only of a_3, b_1 and b_2 on object o_4 but also a serial execution of a_0 and b_0 at object o_0.

Example 3.6 *Assume that the partial order of a method a contains only $(a_1, a_2), (a_1, a_3), (a_1, a_4)$ and at object o_2 a_2 and a_3 are concurrently runnable. An optimal execution of submethods of method a is: a_1 executes first, then a_2 and a_3 execute concurrently at o_2 and both concurrently with a_4.*
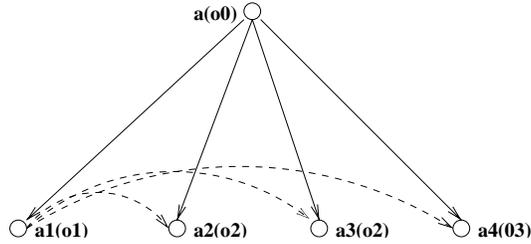
Figure 8:

This execution can be proved correct by extending the partial order to a total order. The extension is not unique. All possible orderings are $a_1 < a_2 < a_3 < a_4$, $a_1 < a_3 < a_2 < a_4$, $a_1 < a_4 < a_2 < a_3$, and $a_1 < a_4 < a_3 < a_2$.

Note 3.3
Commutativity is in this model a user-defined property and it is not inferred by strictly reasoning about the semantics of the methods involved.

Proposition 3.1 *Reversing two commuting leaf methods not ordered in the general tree preserve the partial order of the general tree and leads to an equivalent computation.*

Proof: It is sufficient to prove that the partial order of the method which is the least common ancestor of the two commuting leaves is preserved and its computation in the new ordering is equivalent to the old one.

Let m_i and m_j be two methods whose least common ancestor is m. If m_i and m_j commute at object o then both $m_i \rightarrow m_j$ and $m_j \rightarrow m_i$ executions leave o in equivalent states $s(m_j, o, s(m_i, o, s_{initial,o})) \equiv s(m_i, o, s(m_j, o, s_{initial,o}))$, where $s_{initial,o}$ is the state of o before executing any of m_j and m_i. The partial order on the computation of m is not altered because m_j and m_i did not exchange places with any other methods. By assumption, m_j and m_i were not ordered in the general tree (they were not descendants of conflicting methods that would impose either $m_j < m_i$ or $m_i < m_j$) so reversing their order of execution at o does not conflict with any pre-existent ordering of them. Except for o, the rest of the objects are not affected by this reversal so the new computation of m is equivalent to the old one, thus reversing two not ordered leafs leads to an equivalent computation. $\square$

An implicit assumption not stated very often in the serializability literature is that two complex methods commute if their executions are regarded as atomic.

Example 3.7 *Consider two methods decrease_price_5%() and decrease_price_%() that commute at object o in a database representing a store. Suppose that they are implemented as read_price() $\rightarrow$ modify_price()*
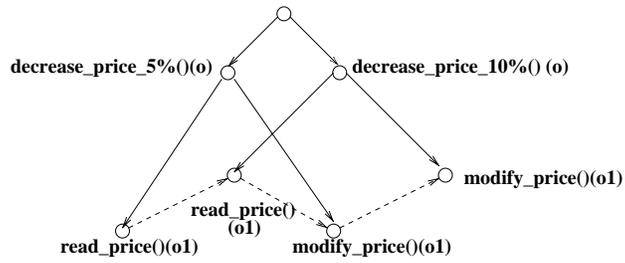
23

Figure 9:

This execution is not acceptable because the result will be only a decrease in price with 10% instead of 14.5%. A correct execution is :
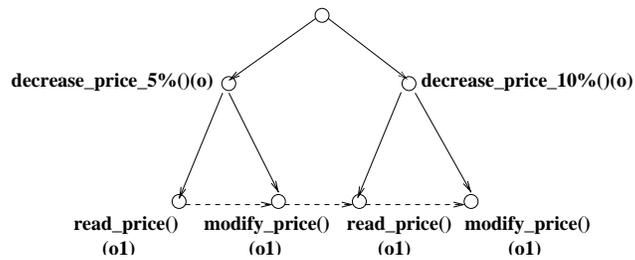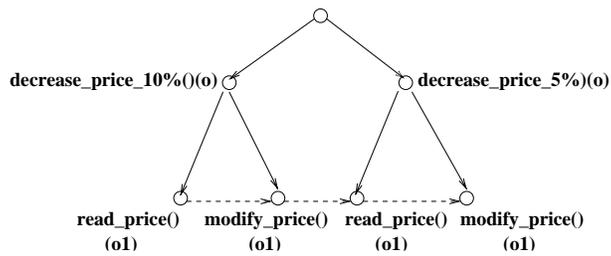

Figure 10:

And a correct reversal is

Figure 11:

This observation leads to a condition for reversing two commuting methods.

Proposition 3.2 *Let* m *and* n *be two methods that commute at object* o *and are not ordered in the global tree. If their execution is reversed such that all leaves of one method at any object involved in execution are executed before any leaf of the other method then the partial order of the general tree is preserved and the resulting execution is equivalent with the execution before reversal.*

Note 3.4
The partial order within m *and* n *must be the same as in the old execution.*

Proof: At object o, $s(m, o, s(n, o, s_{initial,o})) \equiv s(n, o, s(m, o, s_{initial,o}))$. By the implicit assumption derived from the commutativity of two complex methods, the sub-methods commute at any object involved in computation $s(m_i, o_k, s(n_j, o_k, s_{initial,o_k})) \equiv s(n_j, o_k, s(m_i, o_k, s_{initial,o_k}))$, where m_i and n_j are sub-methods of m and n, respectively, at object o_k. This condition ensures that all possible conflicting sub-methods are ordered such that the execution of the first method at one object completes before the other method starts modifying that object.

The new partial order resulted from these subtree reversals is consistent with the initial order of the global tree as no new pair of ordered methods was added. The fact that all leaves of one method must finish executing at one object before any other leaf of the other method can start executing at that object will not interfere with the partial order of the last method introducing only a possible delay in execution.

As stated at the beginning, all objects involved in execution are left by both orderings in equivalent states so a reversed execution of two internal nodes satisfying the conditions in the proposition leads to an equivalent execution. $\square$

Note 3.5
The above proposition makes no assumption on the relative position of the two commuting complex methods but only requires that no compulsory ordering be between them. The compulsory ordering can come from the programmer's request or because the two methods are descendants of two conflicting methods, in which case they inherit the ordering of their parents.

In the traditional systems that assume flat transactions (no nesting), commutativity alone is sufficient for reordering operations in the techniques used for proving serializability of executions. In the case of nested method execution commutativity, even in its flexible user-defined version, is no longer sufficient. Using commutativity we can reorder the leaves of a method executions to serialize the execution above the leaves, but there is need for a technique that will ease the reasoning about the whole tree of the computation being analyzed.

The two transformations that will be applied in order to change a given computation tree to an equivalent, possibly serial one, are *reduction* and

expansion. These two techniques work on the basis of a very important assumption: each of the methods already serialized by means of commutativity execute *atomically.* As such, the leaves describing the already serialized method computation can be pruned.

Summarizing, pruning is applied after a method was serialized to reduce the nesting of the general tree by simply considering the changes of the method at all objects involved in its computation and cutting the method's computation subtree from the general tree. By pruning, the nodes that were internal become now leaves and the same procedure can be applied to serialize them. Iteratively pruning and serializing the levels of the general tree the level immediately under the root is reached. The general computational tree can be then reconstructed by expanding each of the nodes with its underlying computational subtree, level by level.

Using the serializability correctness criterion, after the leafs of a *separated* method are serialized they can pruned from the computation tree leading to a *state-based conflict-preserving equivalent* computational tree. As opposed to the general commutativity used in the classical theory, the approach detailed above uses a state-based definition. As such, simply ignoring the information contained in objects' states and considering that two operations commute if they commute regardless of the state the objects at which they are invoked are in, we are back to the classical theory.

The algorithm for proving that histories generated by concurrency control protocols are serializable is very simple and can be described as a bottom-up reduction. State-based commutativity is used to reorder the leafs serially. If this operation is successful and all the leafs are serialized, the leaf operations are pruned from the general tree. Reordering and pruning are applied alternatively upwards as long as possible. If this strategy can be applied iteratively up to the roots than the general tree is proven to be serializable, so the schedules generated by the concurrency protocol analyzed are correct. A similar approach would be to reorder operations in a top-down direction using only state-based commutativity and no alterations (pruning) of the general computational tree.

3.6 A locking protocol for OODBMSs with objects exhibiting internal concurrency

3.7 The Basic Protocol

The protocol presented in this paper is an extension of the protocol introduced by Agrawal et all [3]:

1. A method t' can execute an atomic operation t on an object *o* if it can acquire a lock on *o*.

2. A method execution cannot commit until all its children have terminated. When a method terminates:

(a) If it is not the top-level, its locks are inherited by its parent.

(b) If it is not top-level and it aborts, its locks are discarded.

(c) If it is top-level, its locks are discarded.

3. A lock on an atomic operation o is granted to a method if and only if :

(a) The current state of the object permits the execution of the requesting method.

(b) If there exist non-ancestors methods holding inherited locks on o, there are some ancestors of these methods and the requesting method that commute.

(c) Granting locks and scheduling for execution in the same time all other concurrently-runnable methods preserves the partial order devised by the central transaction manager.

The novelty of this protocol consists of forcing parent methods to inherit the locks of terminated children methods, allowing conflicting operations to share locks if they have commuting ancestors and permitting objects to execute more than one method call at a time, according to the specifications designed by the users and the system designer.

3.8 Proof of Correctness

Semantic locking introduces more complexity and implementation effort, but there the increase in complexity can be balanced by the gain in concurrency reflected in a decreased average response time. The scheduler keeps track of the hierarchy of method executions and devise a general strategy for scheduling, dispatching the methods to the objects they are invoked on. The object-based schedulers use the conflict and lock tables to check whether the operations can be executed and what group of methods from the pending queue can be allowed to execute concurrently without altering the order devised by the central scheduler.

The correctness of the protocol can be proved using the bottom-up paradigm. The first step considers only the bottom level of the general tree, containing the last operations of the transactions in the system. Using state information from the previous level the leaf operations are reordered until they come to a serial order and the internal nodes immediately above are separated.

The next step carries on the rearrangement of methods to the next level, considering the state-based commutativity and concurrently-runnable properties. At each iteration of the process the node that has the leftmost leaf operation in the tree.

It has to be proven that in trees representing executions generated by this protocol it is always possible to separate the nodes on a level by level basis. Assume that at a certain level in the general tree, there is a node m that cannot be separated. This situation can occur only if there is another method n interleaved with child methods of m. Let m_i and m_j two sub-methods of m_i such that conflicts and precedes n, and n conflicts and precedes m_j. m_i conflicting

and preceding y means that there exist descendents m_{i_k} and n_l of m_i and n, respectively, such that m_{i_k} conflicts and precedes y_l. Because methods x_{i_k} and y_l conflict but were allowed to run concurrently it can be inferred that the locks for these conflicting methods were held at an upper level by two proper commuting ancestors of m_i, m_j and y. These two commuting ancestors must be methods invoked on the same object as state-based commutativity is defined on a per-object basis. As such, it must be that m_i is invoked at an object ot a higher level in the objects hierarchy than the object at which n is invoked. Similarly it can be shown that n is invoked at an object at a higher level than the object m is invoked at, which contradicts the fact that m is the parent of m_i. The argument presented above can be extended to the general case in which a set of methods are interleaved with methods of a node that cannot be serialized.

4 Conclusion and Future Work

4.1 Applications

This paper presented a model that can be used to describe transactions in object-oriented databases and a set of basic techniques that can be used for proving the correctness of concurrency control protocols considering serializability as the correctness criterion. The immediate applications are the databse systems designed using concurrent object-oriented languages or actor-oriented languages.

4.2 Limitations

Although the model can be used successfully for proving the correctness of single-level systems it is not clear yet what extensions need to be made in order to accommodate multi-level database systems. Because the main concern of this paper was concurrency control, recovery matters were not covered and more research is needed for devising efficient recovery strategies for this model.

4.3 Future Work

Concurrency control and recovery are closely related and a future paper could explore the subtle interdependence that arises between the two mechanisms. The object-oriented systems allow a more flexible handling of exceptions and failures by having a distributed form of concurrency control, but also have more complex coordination problems between the object-based schedulers and the central transaction manager, which need to be clearly defined and solved.

There is also need for more research towards extending the model with more powerful concepts and techniques that can be applied to multi-level database systems.

4.4 Conclusion

The main goal of this paper was to present techniques that can be used to model transactions and concurrency control in databases composed of objects exhibiting internal concurrency. We believe that the set of basic techniques and axioms that have been presented are general enough to be applied successfully to a large variety of database systems .

5   Bibliography

[1] Adya, A., Gruber, R., Liskov, B., Maheshwari, U., *Efficient Optimistic Concurrency control Using Loosely Synchronized Clocks*, Proceedings of SIG-MOD'95, San Jose, USA, 1995.

[2] Agrawal, D., El Abadi, A., and Singh, A. *Consistency and Orderability: Semantics-Based Correctness Criteria for Databases*, ACM Transactions on Database systems, Vol. 18, 3, 460-486, 1993.

[3] Agrawal, D., El Abadi, A., and Resende, R. *Semantic Locking in Object-Oriented Database Systems*, Proceedings of the 9th OOPSLA, 388-402, 1994.

[4] Badrinath, B. R and Ramamrithan, K *Semantic-Based Concurrency Control: Beyond Commutativity*, ACM Transactions on Database Systems, Vol. 17, No.1, 163-199, 1992.

[5] Beeri, C. , Bernstein, P., and Goodman, N. *A model for concurrency in nested transcation systems*, IEEE Transactions on Software Engineering, Vol. 5, No. 5, 203-216, 1979.

[6] Bernstein, A., and Lewis, P. *Concurrency in Programming and Database Systems*, Jones and Bartlett Publisher, 1993.

[7] Bernstein, P., Hadzilacos, V. and Goodman, N, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[8] Codd, E. *A Relational Model for Large Shared Databanks*, Communications of the ACM, Vol. 13, No. 6, 377-387, 1970.

[9] Delobel, C, Lecluse, C., and Richard, P. *Databases: From Relational to Object-Oriented Systems*, International Thompson Computer Press, 1995.

[10] Hadzilacos, T., and Hadzilacos, V. Transaction Synchronization in Object Bases, Proceedings of the 7th SIGACT-SIGMOD-SIGART, Austin, USA, 193-200, 1988.

[11] Herlihy, M. *Apologizing versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types*, ACM Transactions on Database Systems, Vol. 15, No. 1, 96-124, 1990.

[12] Janicki, R. and Koutny, M. *Fundamentals of Modelling Concurrency Using Discrete Relational Structures*, Acta Informatica, Vol. 34, 367-388, 1997

[13] Janicki, R. and Koutny, M. *Structure of Concurrency*, Theoretical

Computer Science, Vol 112, 5-52, 1993.

[14] Kempler A. *Object-Oriented Database Management*, Prentice Hall, 1994.

[15] Khoshafian, S. *Object-Oriented databases*, Wiley Sons, 1993.

[16] Krishnaswamy, V. and Bruno, J. *On the Complexity of Concurrency Control Using Semantic Information*, Acta Informatica 32, 271-284, 1995.

[17] Krishnaswamy, V., Agrawal, D. , Bruno, J.L. and El Abadi, A. *Relative Serializability: An Approach for Relaxing the Atomicity of Transactions*, Journal of Computer and System Science 55, 344-354, 1997.

[18] Kroha, P. *Objects and Databases*, McGraw-Hill Book Company, 1993.

[19] Kumar, V. *Performance of Concurrency control Mechanisms in Centralized database Systems*, Prentice Hall, 1996.

[20] Lamport, L. *Towards a Theory of Correctness for Multi-user Database Systems*, Massachusetts Computer Association, Tech Rep. 1976

[21] Matsuoka, S., and Yonezawa, A. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in G. Agha, P. Wegner, A. Yonezawa, editors, Reserach Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[22] Papadimitriou, C. *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

[23] Rastogi, R., Mehrotra, S., Breitbart, Y., Korth, H., Silberschatz, A. *On Correctness of Nonserializable Executions*, ACM Transactions on Database Systems, Vol. 17, No. 2, 96-124, 1992.

[24] Vidyasankar, K. *Generalized Theory of Serializability*, Acta Informatica, 25, 105-119, 1987.

[25] Won, K. *Introduction to Object-Oriented Databses*. The MIT Press. 1990.