Enhancing Intra-Transaction Concurrency in Object Bases[1]

Peter Graham and Ken Barker
Advanced Database Systems Laboratory
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2
{pgraham,barker}@cs.umanitoba.ca

Abstract

Advanced software systems utilize highly structured, complex data that can be effectively represented using nested objects. In this paper we argue that the efficient execution of transactions on such data can only be accomplished by exploiting the available concurrency *within* each object method.

We present a concurrency control strategy that *automatically* generates nested transactions on objects. Compile time analysis of the class definitions is used to determine conflict information which is used by the method scheduler to perform concurrency control.

Our approach offers three benefits; (1) the availability of sub-transactions which may be executed concurrently, (2) the use of a statically determined conflict condition which is less restrictive than conventional ones, and (3) the compile time determination of conflicts which decreases the overhead of run time concurrency control.

---

1   Introduction

One motivation for developing object systems is their suitability in supporting advanced applications including geographic information, multimedia, and computer aided design, systems. Such applications all utilize highly structured, complex data that can be effectively represented using nested objects. The efficient execution of transactions on complex data can only be accomplished by exploiting concurrency *within* each object's methods.

Nested objects provide a natural environment in which to support concurrency. The nesting of the objects induces a nesting of method invocations where method executions are the unit of concurrency. Unrestrained concurrency however, can lead to execution anomalies that may leave the object base in an inconsistent state. To avoid this we present an algorithm, based on low-cost compile time analysis, which provides run time concurrency control between method executions within a *single* transaction. The obvious benefits of our approach are those of nested transactions [12]; enhanced concurrency, and simplified recovery. Our application of nested transactions to object base systems is straightforward because sub–transaction creation and management is done *automatically*. The transaction programmer need not be aware of sub-transactions to reap their benefits.

Our algorithm employs compile-time derived semantic information to detect conflicts between methods. This provides increased concurrency since the conflict condition is less restrictive than object level locking and it also provides decreased overhead since unnecessary concurrency control operations are not performed.

The semantic information derived is more extensive, albeit often less exact, than information based on dynamically determined attribute reference patterns (E.g. locking schemes). This provides a two–fold benefit. The *á priori* information available permits potential conflicts, or lack thereof, to be detected prior to method execution. If no conflicts occur, no concurrency control overhead need be incurred. If conflicts may occur then their

early detection can permit decreased overhead (e.g. using "summary" locks as in [11]) and also decreases the likelihood of deadlock. The use of order-preserving concurrency control protocols enforces a serialization order on transactions whether or not that order is appropriate. Static semantic information may enhance concurrency by avoiding inappropriate orders.

Because the derivation of the semantic information is performed at class compile time, the overhead associated with its derivation is not a component of the concurrency control algorithm itself. This reduces the run time overhead and permits far more extensive analysis than would be possible at run time. This yields less restrictive concurrency control.

The rest of the paper is organized as follows. Related work is described in Section 2. Section 3 discusses our assumptions and Section 4 provides a definition of *intra-transaction serializability*. In Section 5 the compile-time analyses used to enable our run time concurrency algorithm are discussed. Section 6 presents the run time scheduler. Finally, Section 7 makes some concluding remarks and proposes directions for future research.

2   Related Work

A disadvantage of flat transactions is their monolithic structure which results in undesirable, long duration transactions for operations on complex data. This impacts both concurrency control and recovery. Being able to divide a transaction into parts which match the physical structure of the data being operated on is highly desirable. One approach to adding structure to transactions is the use of *nested transactions* [12, 18].

In nested transactions, a conventional transaction is divided into a root and one or more sub-transactions which may themselves be divided (In object bases, this may be based on a hierarchical division of the data being operated on by the transaction.). The root transaction

controls the execution of its sub-transactions by specifying which sub-transactions execute concurrently and the recovery procedure(s) to be applied in the event of sub-transaction failure. Two forms of transaction nesting exist; closed, and open the choice of which affects the unit of atomicity. Nested transactions offer clear benefits but the manual division of a transaction into sub-transactions and the management of their concurrent execution is a burden on the transaction programmer.

Object base concepts and design strategies have been discussed in several papers [1, 9, 3]. and many research and commercial systems have been constructed including; Exodus [4], ORION [10], and O_2 [5]. These systems typically provide only simple concurrency control mechanisms based on object-level locking.

The issue of concurrency control in object bases has also been discussed extensively with work concentrating on concurrency control for nested objects. Rakow, Gu, and Neuhold [15] define *object-oriented serializability* for open nested transactions in object bases. The concurrency control algorithm they describe exploits the semantics and nesting of operations to increase concurrency. Concurrency control is based on conflict analysis (using commutativity [17]). Commutativity tables for all methods must be generated manually.

Transaction synchronization in object bases has also been addressed by Hadzilacos and Hadzilacos [7] who define transaction management algorithms for closed nested transactions. The correctness of nested two-phase locking and nested timestamp ordering in object bases is shown and a new algorithm is described where a graph constructed according to a method-local partial order and inter-object ordering information is acyclic only if the corresponding history is equivalent to a serial history. Unfortunately, this algorithm corresponds to view serializability and hence is inherently inefficient.

Resende and El Abbadi [16] describe a graph testing concurrency control protocol using closed nesting and the model of Hadzilacos and Hadzilacos. Their *optimistic* protocol

specifies how to dynamically construct a graph for each method invocation which will become cyclic only after serializability has been violated.

Recently, attempts have been made to exploit *automatically derived* semantic information to increase concurrency in object bases. Graham, Zapp, and Barker [6] propose *method-level* scheduling. Dependence between methods is defined (and derived automatically at method compile time) based on attribute accesses. Some of the problems associated with inter-object method invocations are considered and algorithms are presented which use method dependence information.

Hakimzadeh and Perrizo [8] modify the ROLL concurrency control scheme of Perrizo [14] to produce OC-ROLL (Object Centered ROLL). Their approach provides object-local concurrency control on attribute access and uses statically derived bit-strings to summarize attribute access information in much the same way suggested in [6]. Inter-object serializability is provided by strictly ordering transaction method invocations by the time of their transaction's arrival. While *order preservation* simplifies the problem of inter-object serializability, it also decreases potential concurrency.

Malta and Martinez [11] describe an approach to fine granularity concurrency control in object bases which is also similar to [6] and [8]. They derive attribute access information statically and use it to define additional lock modes which thereby incorporate *semantic* information into concurrency control. Their algorithm fails to support nested method invocations and thereby precludes the benefits of nested objects.

3   Assumptions

An object base system consists of a set of uniquely identified objects containing structural and behavioural components. The structural component is a set of *attributes* whose values

define an object's state. The behavioural component is a set of *methods* that are the only means of accessing the structural components. In this paper, $O_i.A_j$ denotes an attribute A_j of object O_i. Similarly, a method M_j is denoted $O_i.M_j$.

Users execute transactions by invoking methods on objects. Thus, a transaction is a method execution. A method invocation made within a method execution is a sub-transaction. We denote the invocation of method M_k on object O_j by transaction T^i as $O_j.M_k^i$.

Objects are instantiated from a library of class definitions which specify the attributes and methods of the objects belonging to each class. It is the class definitions which are compiled and from which the static information is determined. A definition before use rule applies in class definitions. This requirement is reasonable and ensures that *complete* information is available to the class compiler.

This paper assumes that encapsulation is *strictly* adhered to. Inheritance is considered to be a static process in that when a class is defined, all inherited attributes are logically embedded *into* the class. We also assume certain restrictions on the structure of methods. Specifically, control structures and object arrays are not considered. Our approach does *not* require this, but limited space prevents the presentation of the relevant details. Due to the aliasing problems introduced by pointers, we assume no pointer references within methods. This too may not be strictly necessary.

Finally, we ignore inter-transaction concurrency control. It is assumed that some additional, compatible protocol is applied to provide inter-transaction concurrency control.

4 Intra-Transaction Serializability

Conflict serializability is our correctness criterion, the conventional definition of which involves a *collection* of transactions. In the case of intra-transaction concurrency, this is inappropriate. There is only *one* root transaction and the serializability of its sub-transactions must be equivalent to the *single* prescribed serial execution.

Following [7] we distinguish between *local steps* ($LS(O_i.M_j)$) and *message steps* ($MS(O_i.M_j)$) in a method. The local steps are those which operate on attributes and the message steps correspond to method invocations. The set of all steps in a method is denoted $STEPS(O_i.M_j) = LS(O_i.M_j) \cup MS(O_i.M_j)$.

We define two types of executions. A *serial execution* of $O_i.M_j$ where all steps are related by the total order $H_s = (STEPS(O_i.M_j), \prec_s)$, and a *serializable execution* where all steps are related by the *partial* order $H = (STEPS(O_i.M_j), \prec)$ where the relation $\prec$ is induced by the *data dependences* between the steps.

To ensure correctness any valid execution ordering of the method steps must contain $\prec$. Equivalently, any valid execution ordering must be *locally* serializable. By "locally" serializable we mean that it is serializable with respect to the operations on the attributes of the object. The use of dependences allows us to statically determine which execution order(s) will be locally serializable.

Local serializability alone is insufficient to guarantee inter-object serializability. If two method invocations are made against a sub-object, directly or indirectly, and if the corresponding executions conflict, then to be serializable they must be ordered consistently with H_s. Local serializability does not prevent remote serialization errors since the invocations of conflicting method executions on a sub-object may not themselves be locally dependent. Consider case '(2)' in Figure 1 where the invocations of $O_2.M_5$ and $O_3.M_1$ are independent
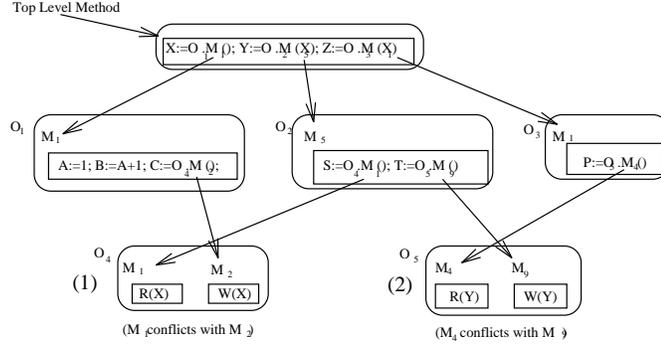
Figure 1: Intra-Transaction Concurrency Example

in the root method execution but there exists a conflict at object O_5. Contrast this with case '(1)' where the conflict at O_4 will never occur because the statements causing the conflicting executions are dependent in the root method and must therefore execute serially.

The correct order for conflicting method invocations on sub-objects is determined at compile time but must be enforced at run time. If method *invocations* are simply marked as conflicting then any concurrency between independent method executions in indirect invocations is lost. This is illustrated in Figure 1 where, in case '(2)', $O_2.M_5$ and $O_3.M_1$ are non-conflicting and can be executed concurrently.

Definition 1 A concurrent execution of the steps of a method represented by the partial order $H_c = (STEPS(O_i.M_j), \prec_c)$ is said to be *intra-transaction serializable* if $\prec_c \subseteq \prec$ (local serializability) and all conflicting method executions on sub-objects are ordered consistently with $\prec_s$. ∎

We now define method dependence based on references made by "access" and "mutate"

steps.

Definition 2 An *access* step is defined to be either a local step which reads an attribute value or a message step which uses an attribute value as an input parameter. ∎

Definition 3 A *mutate* step is defined to be either a local step which assigns to an attribute value or a message step which specifies an attribute as an output parameters. ∎

The following definitions of the three conventional forms of dependence [2, 13] apply to object methods.

Definition 4 *True dependence* $(S_x \delta S_y)$ occurs between steps if S_x is a mutate step which precedes (based on $\prec_s$) S_y, an access step, and both refer to the same attribute. ∎

Definition 5 *Anti dependence* $(S_x \overline{\delta} S_y)$ occurs between steps if S_x is an access step which precedes S_y, a mutate step, and both refer to the same attribute. ∎

Definition 6 *Output dependence* $(S_x \delta^\circ S_y)$ occurs between steps if S_x is a mutate step which precedes S_y, a mutate step, and both update the same attribute. ∎

An arbitrary dependence is denoted $\delta^?$. If steps S_x and S_y are related by $S_x \delta^? S_y$ then S_x and S_y are related by the partial order H.

5 Compile Time Processing

The first goal of our static analysis is to divide each method specification, at compile time, into a collection of concurrently executable components (*concurrent components*). Each component will contain one or more dependent message steps together with those local steps with which they depend. Once the set of concurrent components is identified, the code generated for the method is augmented with concurrency primitives to effect inter-component concurrency. This relieves the run time scheduler of the need to enforce method-local dependence relations.

To determine the concurrent components of a method (Algorithm 1) the inverse dependence graph of a method must be constructed. The determination of inter-statement dependence in the absence of control flow is straightforward and the construction of the dependence graph for a method is direct from the following definition:

Definition 7 The *dependence graph* of method $C_i.M_j$ is a directed graph $DG(C_i.M_j) = (V, E)$ containing vertices $v_x \in V$ for each step S_x in M_j and a directed edge $e \in E$ from v_s to v_t iff $S_s \delta^? S_t$. ∎

The construction of the inverse dependence graph from the dependence graph is captured in the following definition:

Definition 8 The *inverse dependence graph* of method $C_i.M_j$ is a directed graph $IDG(C_i.M_j) = (V, E)$ containing the same vertices as $DG(C_i.M_j)$ and edges $e \in E$ from v_s to v_t iff $DG(C_i.M_j)$ contains edges from v_t to v_s. ∎

Algorithm 1 *Partition into Concurrent Components and Emit Augmented Code*

```
Nd : node in IDG;
CurrTree, TreeCnt, CallNo : Integer;
RECURSIVE PROCEDURE DoTree(Root: node in IDG);
      BEGIN
      myCallNo, numSuccs, currSucc : INTEGER;
      SuccNode : node in IDG;

      myCallNo ← CallNo;  CallNo ← CallNo +1;  emit(Root);
      IF (Root has outdegree = 0) THEN RETURN;
      IF (Root has outdegree = 1) THEN
            BEGIN
            SuccNode ← successor of Root;  DoTree(SuccNode);
      } ELSE {
            numSuccs ← number of successors of Root;
            emit"e"myCallNo": sync "numSuccs;
            currSucc ← 1;
            FOREACH SuccNode a successor of Root DO
                  BEGIN
                  emit("meet e"myCallNo); DoTree(SuccNode); emit("p"myCallNo"."currSucc":");
                  currSucc ← currSucc +1;
                  END
            emit("par p"myCallNo."1, "p"myCallNo."2, ..., "p"myCallNo."TreeCnt);
            END
      END   CallNo ← 1;  emit("exit");
TreeCnt ← number of nodes in IDG with indegree = 0;
IF (TreeCnt > 1) THEN
      emit("END: sync "TreeCnt);
CurrTree ← 1;
FOREACH node Nd with indegree = 0 DO
      BEGIN
      emit("meet END");  DoTree(Nd);  emit("p"CurrTree":");
      CurrTree ← CurrTree +1;
      END
IF (TreeCnt > 1) THEN
      emit("par p"1, "p"2, ..., "p"TreeCnt);
emit("entry");
```

End of Algorithm

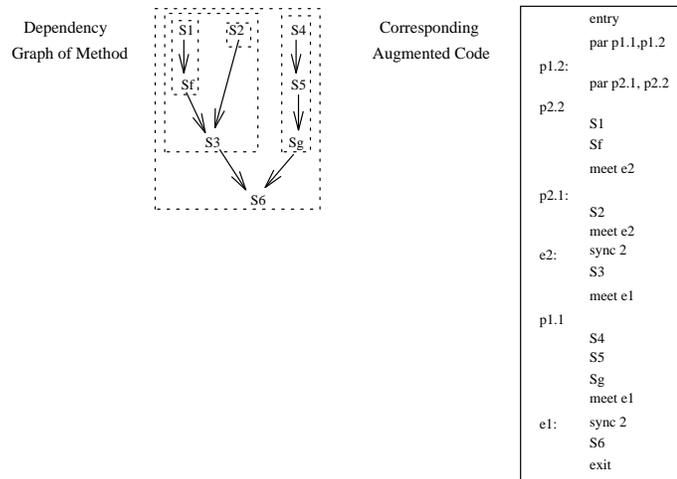| | entry |
	par p1.1,p1.2
p1.2:	
	par p2.1, p2.2
p2.2	
	S1
	Sf
	meet e2
p2.1:	
	S2
	meet e2
e2:	sync 2
	S3
	meet e1
p1.1	
	S4
	S5
	Sg
	meet e1
e1:	sync 2
	S6
	exit

Figure 2: Example Dependence Graph and Augmented Code

The augmentation of the method code with "sync", "meet", and "par" primitives enables the concurrent execution of locally independent method invocations in a method. The run time environment of the object base system must, of course, support the primitives.

A call to "par" creates a separate thread of control for each of its argument labels. The current thread terminates and each new thread begins execution at the corresponding label. To collect the results of concurrently executed sub-transactions, "meet" and "sync" are used. The final statement executed in each thread is a "meet" which specifies that the thread should terminate and synchronize with other threads at the sync point specified as its argument. At each sync point, a "sync" primitive waits for a specified number of threads to meet. The number of threads to synchronize is specified as the argument to "sync". Figure 2 presents an example dependence graph and the augmented code generated for it

by Algorithm 1.

Algorithm 1 does not distinguish between method invocations requiring run time synchronization and those which do not. By rewriting the code the algorithm captures only *method local* dependences. It is the method scheduler's job to provide non-local synchronization and it does this using statically provided method conflict information.

The second goal of our static analysis is, for each object, to determine (at compile or object instantiation time) a conflict relation between all its method pairs. Each method specification is analyzed and the set of attributes it reads and writes are enumerated as $RS(M_i)$ and $WS(M_i)$. These sets are conservative because in the presence of run time conditionals they may specify more attributes than will actually be referenced in a given method execution.

Local conflicts between method pairs are determined by testing $RS()$ and $WS()$.

Definition 9 The *Method Conflicts* of a class C_i is a partial order $(METHODS(C_i), \diamond)$ where $C_i.M_j \diamond C_i.M_k$ if $(RS(C_i.M_j) \cap WS(C_i.M_k) \neq \emptyset) \vee (WS(C_i.M_j) \cap RS(C_i.M_k) \neq \emptyset) \vee (WS(C_i.M_j) \cap WS(C_i.M_k) \neq \emptyset)$ ∎

Once computed, the method conflict relation is stored with the class definition.

Non-local conflicts may occur in nested objects. Capturing the nesting relationship requires the construction of class and object method reference sets. Like the attribute reference sets, the class and object method reference sets are conservative since they must assume that all control flow paths are visited.

The set of all class methods invoked by a given method may be constructed at compile time by taking the locally referenced class methods and then recursively adding the references made by those class methods. Since every class method referenced must already exist, the set of class methods it references must already have been constructed. This, gives

rise to the following definition which embodies the construction technique for the set:

Definition 10 The *class method reference set* of method M_j in class C_i is $CMRS(C_i.M_j) = \bigcup_{\forall C_x.M_y\,referenceable\,byC_i.M_j} CMRS(C_x.M_y).$ ∎

The OMRS for a method can only be constructed at object instantiation time but it need not be constructed for all objects. If no method in an object makes method invocations that can conflict with any other method in the object (based on the CMRS) then there is no need to construct the OMRS.

Definition 11 The *object method reference set* of method M_j in object O_i is $OMRS(O_i.M_j) = \bigcup_{\forall O_x.M_y\,referenceable\,byO_i.M_j} OMRS(O_x.M_y).$ ∎

Two methods in an object, M_i and M_j, may be tested at compile time for freedom from non-local conflicts by examining their CMRSs and the compile time defined subsets of their OMRSs. If M_i and M_j never reference a method from the same class, then they are certainly free of conflicts since they cannot access a sub-object in common. This may be determined by checking for an empty intersection of the methods' CMRSs. For sub-objects which are declared as a *physical* part of the object in the class definition we can test more precisely using the OMRSs restricted to the contained sub-objects. If methods arising from M_i and M_j execute at a common object in the OMRS subsets and those methods conflict, then M_i and M_j conflict, otherwise they may not.

Compile time checking produces one of three possible responses for each pair of class methods; "may conflict", "won't conflict", or "don't know". This information is stored with the class definition and is "inherited" by the objects instantiated from that class. Those method pairs for which compile time checking returns "don't know", must be checked

at object instantiation time. Once all sub-objects of an object have been instantiated, the OMRSs for methods within the object are complete and may be used to determine whether or not non-local conflicts may occur.

There is no need to check for conflicts between the method invocations in a concurrent component since they are executed sequentially. Conflict testing at run time is only done between concurrently active sub-transactions.

6   Run Time Scheduling

Most of the work involved in providing concurrency control is done at compile time and hence the job of the run time scheduler is simple – it checks for conflicts between concurrently executing methods. This is cheap because checking is based on pre-calculated conflict information built either at compile time or at object-instantiation time.

The "run time scheduler" is implemented as a collection of object schedulers, one per object. Each object scheduler ensures the serializability of method executions at that object. Method invocations on an object may be either user transactions or sub-transactions thereof. An object scheduler receiving a user transaction checks for conflicts by using the attribute and method information provided. Knowing when and where conflicts may occur, it sends messages to the object schedulers where sub-transaction conflicts may occur. These messages specify the execution orders required to ensure serializability of the conflicting method executions. The schedulers of sub-transactions simply execute method invocations subject to any orders specified by the root transaction scheduler.

7   Conclusions and Future Work

We have presented a technique for scheduling nested method invocations which offers improved concurrency over flat transactions. We statically analyze method specifications and derive as much conflict information as possible at compile time thereby decreasing run time overhead. The static information is used at run time to allow accurate conflict assessment thereby permitting concurrent, non-conflicting method executions at objects. Though not stated in the paper, our technique is also deadlock free.

This is early work and therefore leaves many areas open for further investigation including the relaxation of the many simplifying assumptions made in this paper. Support for conditionals and loops in methods is easily accomplished. Preliminary work on supporting access to object arrays within loops is nearly complete and is very promising. The aliasing problem for pointers remains to be addressed.

Future work also includes the application of our techniques to smaller program units. This offers additional potential concurrency due to the decrease in granularity. We have extended our method to apply it to control flow *paths* within methods. This idea is interesting since we can statically determine the conditions under which each control flow path will be executed. This information may be used to permit run time checking of the conditions prior to method execution to anable the use of more accurate attribute reference information.

References

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. of the First Intl. Conf. on Deductive and Object-Oriented Databases*, pages 40 – 57, 1989.

[2] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[3] E. Bertino and L. Martino. Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer*, 24(4):33 – 47, 1991.

[4] M. Carey *et al*. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*. Morgan-Kaufmann, 1990.

[5] O. Deux *et al*. The Story of O_2. *IEEE Trans. on Know. and Data Eng.*, 2(1):91 – 108, 1990.

[6] P. Graham, M. Zapp, and K. Barker. Applying Method Data Dependence to Transactions in Object Bases. Technical Report TR 92-7, University of Manitoba, Dept. Comp. Sci., 1992.

[7] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 43(1):2 – 24, 1991.

[8] H. Hakimzadeh and W. Perrizo. Instance Variable Access Locking for Object-Oriented Databases. *Intl. Journal for Micro and Mini Computer Applications*, 1993. *in print*.

[9] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[10] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Trans. on Knowledge and Data Eng.*, 2(1):109 – 124, 1990.

[11] C. Malta and J. Martinez. Automating Fine Concurrency Control in Object-Oriented Databases. In *Proceedings of the Intl. Conf. on Data Eng.*, pages 253 – 260, 1993.

[12] J. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing*. The MIT Press, 1985.

[13] D. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *CACM*, 29(12):1184 – 1201, December 1986.

[14] W. Perrizo. Request Order Linked List (ROLL): A Concurrency Control Object for Centralized and Distributed Database Systems. In *Proceedings of the Intl. Conf. on Data Eng.*, pages 278 – 285, 1991.

[15] T. Rakow, J. Gu, and E. Neuhold. Serializability in Object-Oriented Database Systems. In *Proc., Intl. Conf. on Data Eng.*, pages 112 – 120, 1990.

[16] R. Resende and A. El Abbadi. A Graph Testing Concurrency Control Protocol for Object Bases. In *Proc. of the 4th Intl. Conf. on Computing and Information*, pages 316 – 319, 1992.

[17] W. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Trans. on Computers*, 37(12):1488 – 1505, 1988.

[18] W. Weihl. Theory of Nested Transactions. In S. Mullender, editor, *Distributed Systems*, chapter 12, pages 237 – 262. ACM Press, 1989.