Synchronizing Transactions on Objects [1]

B. R. Badrinath
Krithi Ramamritham

COINS Technical Report 86-08

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

[1]This work was supported in part by the National Science Foundation under grants DCR-8403097 and DCR-85000332.

Abstract- This paper discusses a method for synchronizing operations on objects where the operations are invoked by transactions. The proposed technique, which is motivated by a desire to exploit possible concurrency in accessing objects, takes into consideration the granularity at which operations affect an object. We present a dynamic method for determining the *compatibility* of an invoked operation with respect to operations in progress. In making decisions, it utilizes the state of the object, the semantics of the uncommitted operations, the actual parameters of the invoked operation, and the effect of the operations on the objects. One of the attractive features of this technique is that a single framework can be used to deal with the problem of synchronizing access to simple objects as well as compound objects, i.e., objects in which some components are themselves objects.

Contents

1 Introduction

The utility of transactions as a tool for structuring computations in distributed systems is gaining wide acceptance. Work in this area includes the Argus project [15, 27], the Clouds project [1], the Archons project [22], the TABS prototype [25] and ISIS project [6]. A transaction is a unit of computation with the property that either it sucessfully completes execution or fails without having any effect; this property is called *recoverability or failure atomicity.* Further, a transaction executes as if it is the only activity in the system; this property is called *indivisibility or concurrency transparency.* These properties of transactions make them good building blocks for reliable distributed systems [24].

Objects are instances of abstract data types which implies that they can be manipulated only through well-defined procedures called operations. In this paper we focus on the problem of synchronizing operations invoked by atomic actions. *Serializability* [11] has been used as the main correctness criterion for synchronizing operations invoked by concurrent transactions. When actions involve just reads and writes, serializability can be guaranteed through the use of two phase locking [11], timestamping [4, 20], or hybrid schemes [9, 10]. When the same schemes are used for transactions that invoke operations on arbitrary objects, available concurrency is not exploited. For example, two operations that modify two different components of an object cannot proceed in parallel since both operations will be considered as writes on that object. However, concurrency can be enhanced if the semantics of the operations on objects can be precisely specified and the synchronization mechanism takes this semantics into account. For example, if the specifications for the modify operations above indicates which components of the object are modified then the operations modifying different parts of the object can be allowed to execute in parallel.

Use of complete semantic information might help us attain higher concurrency, but will also increase the complexity of the concurrency control mechanism. The additional complexity arises due to the complex domain of interpretation for various operations[19]. Hence, several researchers have included specific types of information such as transaction classes[3, 12] and the structure of the data[16, 23]. The specific type of information taken into consideration by our proposed synchronization technique is the granularity of the data items accessed by operations. This allows us to tap sources of potential concurrency without overly complicating the synchronization technique.

In our model each object is represented as a *granularity graph.* For each operation request we construct an *affected-set*, a set of edges and vertices in the graph, affected by the operation. Commutativity is used as the mechanism for determining compatibility of an invoked operation with other uncommitted operations on the object. One advantage of

our approach is that the compatibility of operations is determined dynamically, i.e.,when the operation is requested to be executed on an object. Improved concurrency is obtained as the compatibility is determined based on the current state of the object, uncommitted operations on it, the actual parameters of the operation and the semantics of the operation. There is no explicit request for a lock; instead, each operation request is granted if it can be executed in parallel with uncommitted operations. Another advantage of our approach is that we are able to handle synchronization of simple and compound objects within the same framework.

Serializability [11] has been used as the correctness criterion for the execution of concurrent transactions. Consider, for example a *mailbox* (two of the operations on it are send and receive) [14]. Mail can be *sent* by a user by presenting the mail system with the receiver's *user-id*; the *message* will be deposited in the receiver's mailbox. Mail can be received by a user by presenting the mail system with his *user-id*; the contents of one of the messages in the user's mailbox are returned to the caller. Suppose the required semantics of send and receive are such that every message sent will eventually be in the mailbox and that every message in the mailbox will be received (provided a user performs sufficient number of receives after the mail is deposited). If sending and receiving mail(s) is considered as *write* and *read* respectively on the mailbox object, then serializability [11] will cause all sends to be serialized. This is because write operations lock the whole object. However, if a send(receive) operation locks only the mail being sent(received) then concurrency is enhanced. The resulting changes are not serializable at the granularity of the mailbox; they are serializable with respect to the mail messages. Another way of stating this that changes made by concurrent actions may not be serializable if physical states form the basis for state equivalence; they will be if we use logical state equivalence as the basis. In the case of the mailbox, two mailboxes are *logically equivalent* if they contain the same messages.

Our approach to determining compatibility of two given operation types involves two steps. In the first step the semantics of the operations are analyzed to see if the operations of the two types are 1) always compatible, 2) always incompatible, or 3) compatible under certain conditions. (These conditions are typically dependent on parameters of the operations and the current state of the object).

The second step which is performed dynamically (i.e., at execution time), is performed only for those operations which are compatible under certain conditions (case 3). Compatibility of such an invoked operation with the operations in progress, (i.e., operations yet to be committed) is determined by the object's manager in an efficient manner. This process is efficient as it involves a check of the corresponding entry in the compatibility

matrix and if the entry indicates that the operation is compatible under certain condidtions then it is just a matter of determining intersection of affected-sets. It is this aspect that differentiates our technique from other schemes which use statically-specified locks [1] which are inherently pessimistic. It is the use of automated analysis in combination with dynamic checking that makes our approach attractive.

The rest of the paper is organized as follows. Section 2 briefly describes previous work in this area while section 3 provides details of our approach. Section 4 presents an example to illustrate our approach. Section 5 concerns the design of object managers given our scheme for synchronizing concurrent access. This design also deals with synchronizing access to compound objects, i.e., objects that are themselves made up of objects. In section 6 we make some concluding remarks and discuss future work.

2 Previous work

Various concurrency protocols based on locking and timestamping for databases are described in [5]. In Argus [15], synchronization is achieved using locking. For built-in atomic data types the operations are still considered in terms of *reads* and *writes*. Argus also provides user-defined atomic types which allow more concurrency by using the semantics of the operations but requires complex implementations [28].

In [1] different levels of locking using progressively more semantics is provided to the user. The choice of the level of locking is left to the programmer. Further, the specifications concerning the compatibility of the operations have to be given by the user. We feel that the designer of an object type need only specify the semantics of operations; their compatibilities ought to be determined from these specifications.

In [21], synchronization of shared abstract data types is done by type-specific locking. The compatibility of operations is determined by considering all the type-specific dependencies. In [18], there is a manager for each object and the compatibility of operations is directly specified. In [18] and [21] compatibility is determined statically and issues concerning the synchronization of compound objects are not dealt with. By contrast, the method proposed here permits handling of compound objects.

3 Our Approach

We begin by giving a brief overview of our approach and describe the model we use. In section 3.1 we give some definitions. In section 3.2, we introduce the concept of a granularity graph and the construction of affected-sets. Section 3.3 relates affected-sets to

commutativity of operations.

In our model, we assume that there is a manager for each object that handles requests from transactions, on that object. The manager of an object synchronizes operations on that object by concurrent actions, based on the compatibility of these operations.

We have adopted *commutativity* as the basis for determining whether a particular operation can be allowed to execute concurrently with those in progress. (Two operations *commute* if the order in which they execute does not affect the results of the operations i.e., results returned by the operation as well as the resulting state of the objects accessed). Because of this, the results of executing an operation hold independently of whether other operations commit or abort. In other words commutativity of operations will not only ensure serializability but also avoid *cascading aborts.*

In our method, each object is logically represented in terms of a *granularity graph.* The vertices are the elements at a certain granularity and the edges represent the *composed-of* relation. Each operation affects the vertices or edges or both. The manager of each object determines the affected vertices and edges. The set of vertices and edges affected by each operation is known as its *affected-set.* It is shown later that two operations from different transactions commute if the *affected-sets* of the respective operations are non-intersecting. The specification of an operation will identify its *affected-set.* Based on the semantic specification of an operation the object manager constructs its *affected-set* and determines whether this operation is compatible with (i.e., commutes with) uncommitted operations.

3.1 Definitions

We have extended here some of the definitions given in [13] to include transactions on objects.

Definition 1: A transaction t is a linearly ordered sequence of steps $(z_1, \ldots, z_n)$ where each z_i is of the form Y_{pq}, indicating that it is an operation p on object q. If $O = \{o_1, o_2, \ldots, o_m\}$ is a set of objects, each in a consistent state, then $t(O) = z_n(z_{n-1}(, \ldots, z_1(O), \ldots))$ is also in a consistent state. Here each z_i corresponds to a defined operation on a particular object which belongs to O.

Definition 2: Let $T = \{t_1, \ldots, t_n\}$ be a set of transactions. The steps of each t_i in T are denoted by triples $\{i, j, Y_{mn}\}$, read as step j of transaction i is operation m on object n.

Definition 3: A *schedule* S for T is a linear ordering of all steps of all members of T such that the linear ordering of steps within each member is preserved.

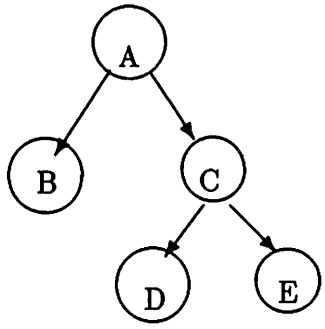

Figure 1: A Granularity Graph

Definition 4: A schedule S for T = $\{t_1, t_2, \ldots, t_n\}$ is said to *interleave* the steps of distinct transactions t_i and t_j on object m if for some steps (i, a, Y_{pm}), (i, b, Y_{qm}), and (j, c, Y_{rm}), step (i, a, Y_{pm}) precedes (j, c, Y_{rm}) in S and step (j, c, Y_{rm}) precedes (i, b, Y_{qm}) in S. A schedule with no *interleaving* steps is a serial schedule.
Note: The operations p, q, r on object m can be the same.

Definition 5: A schedule S for T= $\{t_1, t_2, \ldots, t_n\}$ is *consistent* if it is logically equivalent to a serial schedule S' for T. We call such a schedule *serializable.* That is, a schedule S for T is compatible if there is some total ordering of the members of T such that $S(O) = t_a(t_b(\ldots(t_c(O)\ldots)))$ for the set of objects O.

3.2 Granularity Graph and Affected Sets

Definition 6: A *granularity graph* is a directed acyclic graph whose *vertices* are interpreted as elements of certain granularity and whose *edges* typically represent the *composed-of* relation. A *vertex* at any level can also be an object if operations on objects at that granularity level are specified. (This leads to compound objects.) If there is an *edge* from a_i to a_j, then we say a_i is *composed-of* a_j.

In figure 1, object A is *composed-of* elements B and C and C is *composed-of* D and E. If, for example, C itself is an object, then A is a *compound object.* AC is the *parent-edge* of C and CE is a *child-edge* of C. E is the *child-vertex* of CE and C is the *parent-vertex* of edge CE. Also, as in standard terminology, C is the *child* of A, A is the *parent* of C, A is the *ancestor* of E and E is the *descendant* of A.

Each operation changes the state of the object by modifying the components of the object. These modifications manifest themselves as modifications to edges and vertices.

Specifically, an operation on an object can manifest itself as one or more of the following operations on the granularity graph representing the object.

1. Deletion of vertices and their parent edges.
2. Insertion of vertices and their parent edges.
3. Modifications to the values of vertices.
4. Examination of the values of vertices.
5. Examination of edges.

Based on the effect of an operation on the state of the object, an *affected-set* is constructed as explained below.

Definition 7: The *edge-set* is the set of pairs (e, a) where e is an edge affected by the operation and a indicates the affect on the edge and hence is one of *insert*, *delete*, or *examine*.

Definition 8: The *vertex-set* is the set of pairs (v, a) where v is a vertex and a indicates the affect on the vertex and hence is one of *insert*, *delete*, *examine*, or *modify*. An operation may also return a result that depends on the state of the object. The set of vertices examined by such an operation is also included in the *vertex-set* of the operation.

Definition 9: The *edge-set* together with the *vertex-set* of a particular operation is known as the *affected-set* of that operation.

An operation may affect a set of edges as well as vertices satisfying a certain condition (predicate). For example, a *delete* operation on elements which satisfy a certain condition can be represented by a set with a characteristic function instead of including all the edges and vertices explicitly. However, membership in a set with an arbitrary characteristic function is undecidable; hence, we will have to construct simple predicates as in [11].

Definition 10: The *intersection* of the *affected-sets* of any two operations OP_1 and OP_2, denoted by $\bigcap_G$, contains:

1. Edges e where $(e, a_1) \in$ the *edge-set* of OP_1 and $(e, a_2) \in$ the *edge-set* of OP_2 and both a_1 and a_2 are not *examine*.
2. Vertices v where $(v, a_1) \in$ the *vertex-set* of OP_1 and $(v, a_2) \in$ the *vertex-set* of OP_2 and both a_1 and a_2 are not *examine*.

3.3 Operations and Commutativity.

Each operation request by a transaction is allowed to be executed only if it is compatible with the uncommitted operations already performed; i.e., its execution will not affect and will not be affected by other uncommitted operations. The compatibility of the requested operation with ongoing operations on the object is determined by the object manager. Our formalism uses commutativity of operations to determine their compatibility.

Definition 11: Let Y_1 and Y_2 be two operations on object K in its current state. We say that these operations *commute* if the effect (which includes both the results returned and any modification to the state) is the same whether Y_1 is performed first and then Y_2 or vice versa [13]. Thus, a pair of operations Y_i and Y_j defined on an object K is said to commute if the final state of object K and the results returned by the individual operations are invariant (denoted by $\simeq$) to the order in which the two operations are executed.

Theorem 1: Two operations Y_i and Y_j commute if *affected-set*$(Y_i) \bigcap_G$ *affected-set*$(Y_j) = \emptyset$.

Proof: Consider the effect of operation Y_i executing after Y_j, that is the sequence $Y_i \circ Y_j$. Since the *affected-sets* are non intersecting, the set of edges and vertices affected by Y_j is different from Y_i. The sub graph of the granularity graph modified or selected by Y_i is different from Y_j. Hence the state of the components of the object as seen by Y_i or the values returned by Y_i is the same as if it had operated on the original object K. The same holds for the subgraph of the granularity graph modified by Y_j after the execution of Y_i on K. Hence the overall modification to the state and the values returned is the same irrespective of the order. Hence $Y_i \circ Y_j(K) \simeq Y_j \circ Y_i(K)$, thus Y_i and Y_j are commutative.

Note that, it is possible for two operations to commute even when the affected-sets are intersecting. For example, two *increments* commute as executing them in either order yields the same result and the operations involve modification of the same vertex. We are being pessimistic here for reasons of efficiency in determining commuting operations. A less pessimistic conflict predicate would where applicable, involve the semantics of the changes done to each vertex in the vertex set.

Corollary: Two operations commute if they are on different objects.

Theorem 2, discussed next, states that if two operations commute then they can be executed concurrently and yet preserve serializability. In other words, an operation can be executed if it is commutative with other uncommitted operations.

Typically, two phase locking [11] has been used to guarantee serializability. However, if the locks are released early, then there is the problem of cascading aborts [29]. To preclude the possibility of cascading aborts, operations on shared objects must not be able to see the information that might change if an uncommitted transaction were to abort. However, Theorem 2 states that if operations are allowed only if they are commutative then serializability is ensured and cascading aborts are prevented.

Definition 12: In any arbitrary schedule S, consider a step (i, a, Y_{pK}) of a transaction t_i, where Y_{pK} is an operation on an object K. (i, a, Y_{pK}) is said to *immediately precede* a step (j, c, Y_{qK}) of transaction t_j, where Y_{qK} is an operation on the same object K, if there is no other interleaved step (k, b, Y_{rK}) between (i, a, Y_{pK}) and (j, c, Y_{qK}), and Y_{rK} is an operation on the same object K for any transaction.

Theorem 2: Let O be the set of objects on which operations are defined. Let T be a set of transactions which operate on the objects. Suppose every operation is allowed to be executed only if it commutes with other uncommitted operations is serializable. Then the resulting schedule S for T is free of cascading aborts.

Proof: Let t_i, t_j be any two transactions. Let Y_{lK}, Y_{mK} be 2 commutative operations on an object K. Let the sequence of steps in t_i be $(i, 1, Y_{aK})$, $(i, 2, Y_{bL})$, ... , (i, m, Y_{cN}) and the sequence of steps in t_j be $(j, 1, Y_{dM})$, $(j, 2, Y_{eP})$, ..., (j, n, Y_{fQ}). For any arbitrary interleaved schedule S resulting from the use of commutativity as the compatibility criterion, let Y_{lK} of t_j *immediately precede* Y_{mK} of t_i . That is, (j, e, Y_{lK}) *immediately precedes* (i, g, Y_{mK}) for some e and g. If Y_{lK} and Y_{mK} commute, it is equivalent to (i, g, Y_{mK}) followed by (j, e, Y_{lK}). If Y_{mK} is the only operation performed by t_i and Y_{lK} the only operation by t_j then since Y_{mK} and Y_{lK} commute, they can be executed in any order producing results equivalent to both $t_i \circ t_j$ and $t_j \circ t_i$. If t_i and t_j invoke multiple operations including Y_{mK} and Y_{lK} then if all operations in t_i commute with all operations in t_j then the results will be equivalent to both $t_i \circ t_j$ and $t_j \circ t_i$. On the other hand if operation (i, g, Y_{mK}) *immediately precedes* (j, e, Y_{lk}) and does not commute then (j, e, Y_{lK}) can be executed only after action t_i commits in which case t_i will appear before t_j in the serialization order. Thus, in all cases, the interleaved schedule is serializable.

Recall that according to our model, since Y_{lK} and Y_{mK} commute, the *affected-set*$(l) \bigcap_G$ *affected-set*$(m) = \emptyset$. The components of the granularity graph as observed by

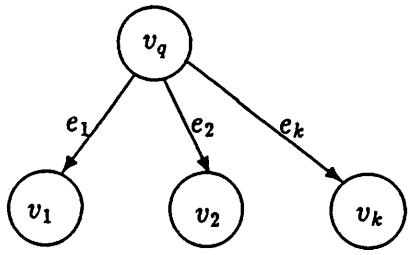

Figure 2: Granularity Graph for a Semi-Queue

any operation of t_j are not dependent on whether t_i runs to completion or not. Hence operation Y_{mK} by t_j can commit independently of t_i. Hence there can be no cascade aborts.

4. An Example

As mentioned earlier, the manager of an object determines conflicting requests using commutativity as the compatibility criterion and synchronizes concurrent requests for operations.

To illustrate the construction of the *affected-set* and show how the granularity information enhances concurrency, we shall examine a specific data type and determine the compatibility of the operations defined on objects of that type.

Consider the *Semi-Queue* data type whose granularity graph is as shown in figure 2. Semi-Queues are similiar to queues except that elements are not required to be removed in strict FIFO order. Two operations on Semi-Queues are: **enq**, which adds an element to a Semi-Queue, and **deq**, which removes and returns an element *nondeterministically.* This nondeterminism offers potential concurrency. Two **enq** operations can run concurrently as can an **enq** and **deq** operation or two **deq** operations as long as operations involve different elements. This is the required condition attached to the CYES (Conditional YES) in the compatibility matrix given at the end of this section. Whether or not the condition holds for a pair of operations can be dynamically determined by finding the *intersection* of the respective *affected-sets.*

The specifications for the operations in terms of the granularity graph are as follows.

1. **Create(Q)**:{ insert a vertex v_q = Q (corresponds to the semi-queue Q) }

	CREATE	ENQ	DEQ	DISPLAY
CREATE	NO	NO	NO	NO
ENQ	NO	YES	CYES	NO
DEQ	NO	CYES	CYES	NO
DISPLAY	NO	NO	NO	YES

Table 1: Compatibilty Table for Semi-Queue

2. **Enq(a,Q)**: { examine vertex v_q =Q; insert a *child-edge* e_k (k is unique) and a vertex $v_k = a$ such that e_k is the *parent-edge* of v_k(indicates that a is a component of the Semi-Queue Q)}

3. **Deq(Q)**: { examine vertex $v_q = Q$; choose any child-vertex v_i; delete the child-edge e_i connecting v_q and v_i; delete the vertex v_i. }

4. **Display(Q)**: { examine all the children of vertex $v_q = Q$ }.

Note: The subscripts i, k, q for edges and vertices are used to identify edges and vertices. If an operation is executable, we assume that a manager executes the specified modifications in an atomic fashion.

The specifications of the operations is specified in terms of the edges and vertices affected in the granularity graph. These specifications are analyzed statically to determine the compatibility of operations and thereby to construct a compatibility matrix as follows: If the intersection of the sets is empty under all conditions the operations are always compatible, if it is not empty under all conditions then the operations are not compatible. However, when the intersection of the two *affected-sets* depends upon some input parameter of the operations or on the state of the object, then the operations are said to be compatible under certain conditions.

When two operations are always compatible, the corresponding entry in the compatibility matrix will be YES. An entry YES indicates that the intersection of the *affected-sets* need not be determined as the operations are always compatible. Similarly, when two operations can never be compatible, the entry will be NO. However, when two operations are compatible under certain conditions, the entry in the compatibility matrix will be CYES, which indicates that the object manager must determine whether the two *affected-sets* intersect.

The compatibility matrix for operations defined on a Semi-Queue data type is as shown in table 1.

Recall that, in our model, for each object, there is a manger which is responsible for controlling the operations. This manager uses information in the compatibility matrix at execution time. The affected-sets of operations on an object are constructed and maintained by the object's manager. As commutativity is the basis for determining compatibility the object manager can take decisions locally without interacting with the managers of other independent objects.

The effects of a transaction must be committed or aborted only when the transaction completes or fails. It is the task of managers to ensure this. An object could be a simple object i.e., none of the components of the object are themselves instances of shared abstract data types, or it is a compound object, i.e., one or more components of the objects are themselves objects.

If an object is a simple object then the manager of the simple object maintains the granularity graph of the object as well as the *affected-sets* for uncommitted operations. On the other hand, if the object is compound then the managers of the components which are themselves objects cooperate in the maintenance of the affected set of the compound object.

To illustrate the process of synchronization we will consider the example of an airline reservation system, shown in figure 3. Each day, the airline operates a number of flights. Each flight has an arrival/departure record and a record for each class such as business, economy and first class. Information on each class consists of the number of reservations made, maximum number of seats in that class as well as information on each reservation made. In figure 3, components of the compound object which are themselves objects are marked by concentric circles. Operations such as *reserve-seat, cancel-seat, cancel-flight, add-flight, etc.* are defined on the compound object.

Consider a request to reserve a seat on flight TWA 16 on a certain date in economy class. Assume that the operation request is of the form *reserve (airline = TWA, date = 26, fltno = TWA16, class = economy, seats = 1, ...)*. With reference to figure 3, reserving a seat means examining the vertices T (airline), v_1 (date), v_2 (flight number), v_3 (class) and an insert operation on the object (v_3) corresponding to seat to be reserved in economy class. The *affected-sets* constructed by various object managers will be as follows.

- The *affected-set* at T : $(T, e), (v_1, e)$ (examine airline and date)
- The *affected-set* at v_1 : (v_2, e) (examine flight number)
- The *affected-set* at v_2 : (v_3, e) (examine economy class)
- The *affected-set* at v_3 : $(v_5, i), (e_5, i)$ (insert seat)

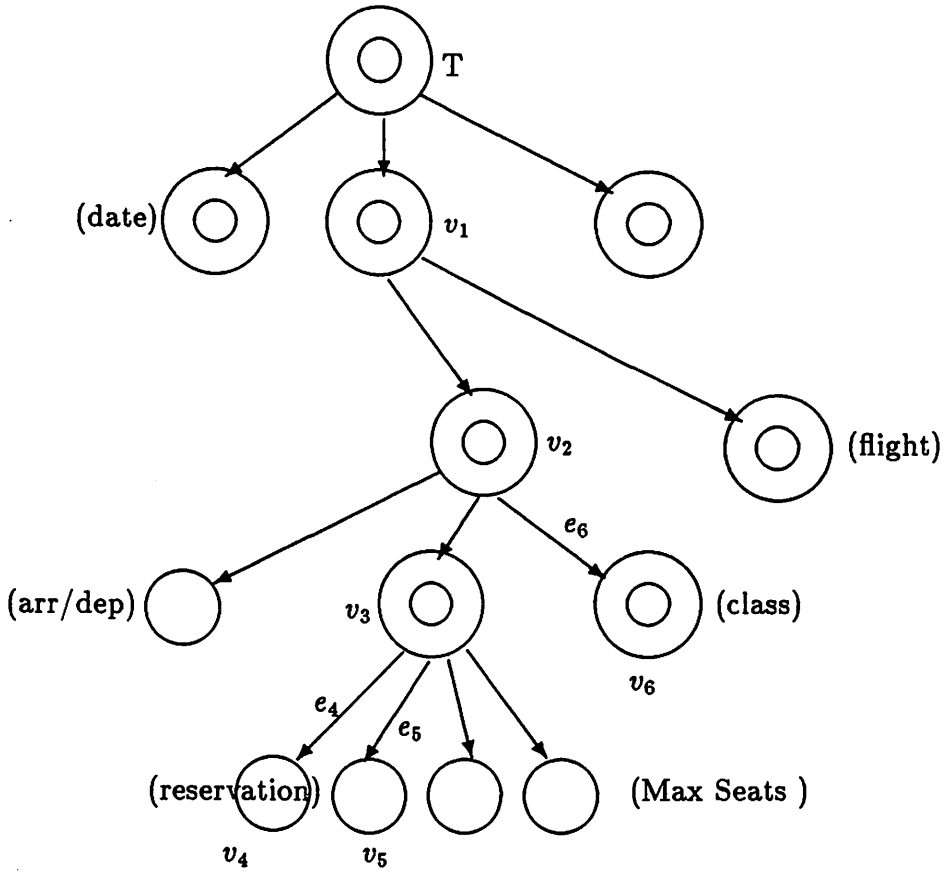

Figure 3: Airline Reservation System

If another request by the airline company to cancel the flight TWA 16 on the same date comes then executing the operation will involve examining the vertex T and deleting the vertex v_1 and the ones below it. If the operation request comes before the transaction that reserved a seat commits, the operation will be found incompatible by the object manager at T because the *vertex-sets* of the two transactions intersect.

Since a compound object has components which are objects, a number of managers are involved in the synchronization of access to a compound object. We now discuss the interactions between these managers. In general, let an operation on a particular object involve vertices along the path v_a v_b, and v_m in the object's granularity graph. Assume v_a, v_b and v_m are objects. The *affected-set* involving edges and vertices from v_a to v_b will be maintained by the manager of the object corresponding to v_a, and similarly the *affected-set* involving vertices and edges from v_b to v_m will be maintained by the manager of the object corresponding to v_b.

The compatibility of an operation involving the vertices and edges along the path v_a to v_b will be first determined by the manager corresponding to the object v_a. If it is found compatible then the manager will invoke the manager of object v_b, which will check compatibility of vertices and edges along the path $v_b, \ldots, v_m$ and so on. Otherwise, the requested operation will be deemed to be incompatible. This distributed maintenance of the granularity graph of a compound object makes it possible to implement our scheme even when the components of the compound object are themselves in different nodes of a distributed system.

Let us consider another example. The airline wants to insert a new *class* on the flight TWA 16 on a certain date from New York to Boston. This will involve examining the vertex corresponding to the requested date, examining the vertex corresponding to the flight TWA 16 and inserting the vertex corresponding to the new class. Since *reservation* type is itself an object, it means creation of a new instance of the object. Hence this will result in the creation of the following *vertex-sets* (v_6 is the vertex being inserted).

- The *vertex-set* at T :$\{(T, e), (v_1, e)\}$ (examine airline and date)
- The *vertex-set* at v_1 :$\{(v_2, e)\}$ (examine flight number)
- The *vertex-set* at v_2 :$\{(v_6, i), (e_6, i)\}$ (insert a new class)

Whereas the above operation will be involved at the top level, in general, a user process can invoke the manager at any level in the object hierarchy. In such a case each object manager maintains the *affected-set* for operations executed on the corresponding object and checks for compatibility using the same procedure as used in the case of a request

coming to a top-level manager. However, the results are returned to the manager to which the initial request was sent.

4 Conclusions and Future work.

We have provided a method for synchronizing generalized transactions. The model was based on representing each object as a granularity graph. The *affected-set* of each operation was constructed from its semantic specification. Compatibility of operations was based on their commutativity, which in turn was determined from their *affected-sets.*

As mentioned earlier, the more semantics we use in synchronization to improve concurrency, the more complex the synchronization mechanism becomes. Here we utilized the structure of the objects, represented by an abstract granularity graph and expressed the semantics of operations in terms of operations on the granularity graph. Our method achieves higher concurrency than other schemes which set *read* or *write* lock on the entire object.

The synchronization scheme we have adopted is similar to locking. Hence, as in any locking scheme, can result in deadlock. Deadlock can be handled using any of the known deadlock detection algorithms [7, 17]. We could also use avoidance schemes by imposing a structure on the objects and adapting techniques similar to those developed for structured databases [8, 26].

In [2], we have proposed a new semantics-based conflict predicate which allows even non-commutative operations to be executed while still avoiding cascading aborts. Another extension we are currently working on involves the development of schemes for object recovery from action aborts. Hereagain, use of operation semantics specified through the granularity graph scheme of object representation will greatly reduce the overheads inherent in the maintenance of information needed to perform recovery.

References

[1] Allchin, J. E., and McKendry, M. S., "Synchronization and recovery of actions," *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, (August 1983), pp. 31–44.

[2] Badrinath, B. R., and Ramamritham, K., "Semantics-based concurrency control: Beyond commutativity," Submitted to *ACM Transactions on Database Systems.*

[3] Bernstein, P. A., Shipman, D. W., and Rothnie, J. B., "Concurrency control in a

system for distributed databases(SDD-1)," *ACM Transactions on Database Systems*, Vol. 8, No. 2 (June 1983), pp. 186–213.

[4] Bernstein, P. A., and Goodman, N., "Concurrency control in distributed database systems," *Computing Surveys*, Vol. 13, No.2 (June 1981), pp. 185–221.

[5] Bernstein, P. A., Goodman, N., and Hadzilacos, V., "Concurrency control and recovery in database systems," Addison Wesley, 1987.

[6] Birman, K. P., et al., "Implementing fault-tolerant distributed objects," *IEEE Transactions on Software Engineering*, Vol. 11, Vol.6 (June 1985), pp. 520–530.

[7] Bracha, G., and Toueg, S., " Distributed algorithm for generalized deadlock detection," *Third ACM Symposium on Principles of Distributed Computing*, (August 1984).

[8] Buckley, G. N., and Silberschatz, A., "Beyond two phase locking", *Journal of the ACM*, Vol. 31, No. 2 (April 1985), pp. 314–326.

[9] Chan, A., et al., "The implementation of an integrated concurrency control and recovery scheme," *In SIGMOD Conference on Management of Data* , (June 1982) pp. 184–191.

[10] DuBourdieu, D. J., "Implementation of distributed transactions," *In Proceedings of the Sixth Berkely Workshop on Distributed Database Management and Computer Networks*, 1982, pp. 81–94.

[11] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The notion of consistency and predicate locks in a database system," *Communications of the ACM*, Vol. 19, No. 11 (November 1976), pp. 624–633.

[12] Garcia-Molina, H., "Using semantic knowledge for transaction processing in a distributed database," *ACM Transactions on Database Systems*, Vol. 8, No. 2 (June 1983) pp. 186-213.

[13] Korth, H. F., "Locking primitives in database systems," *Journal of the ACM*, Vol. 30, No. 1 (January 1983), pp. 55–79.

[14] Liskov, B., and Scheifler, R., "Guardians and actions : Linguistic support for robust distributed programs," *ACM Transactions on Programming Language and Systems*, Vol. 5, No.3 (July 1983), pp. 381–404.

[15] Liskov, B., "Overview of the ARGUS language and system," *Programming Methodology Group Memo 40*, M.I.T, Laboratory for Computer Science, Cambridge, MA (February 1984).

[16] Mohan, C., Donald Fussell, et al., "Lock conversion in non two phase locking protocols," *IEEE Transactions on Software Engineeing*, Vol. 11, No.1 (January 1985), pp. 15–22.

[17] Mukul, S., and Natarajan, N., " A priority based distributed deadlock detection algorithm," *IEEE Transactions on Software Engineering* , Vol. 11, No.1 (January 1985), pp. 67–80.

[18] Natarajan N., "Communication and synchronization primitives for distributed systems," *IEEE Transations on Software Engineering*, Vol. 11, No. 4 (April 1985) pp. 396–416.

[19] Papadmitriou, C. H., "The serializability of concurrent database updates," *Journal of the ACM*, Vol. 26, No. 4 (October 1979), pp. 631–653.

[20] Reed, D., P., "Naming and synchronizing in a decentralized computer system," PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA (1978).

[21] Schwarz, M. P., and Spector, A. Z., "Synchronizing shared abstract data types," *ACM Transactions on Computer systems*, Vol. 2, No. 3 (August 1984), pp. 223–250.

[22] Sha Liu, E., Jensen, E. D., Rashid, R., and Northcutt, J.D., " Distributed co-operating processes and transactions," *Proceedings of SIGCOMM Symposium* , (August 1983), pp. 188–196.

[23] Silberschatz, A. and Kedem, Z., "Consistency in hierarchical database systems," *Journal of the ACM*, Vol. 27, No. 1 (January 1980), pp. 72–80.

[24] Spector, A. Z., and Schwarz, M. P., "Transactions: A construct for reliable distributed computing," CMU Technical Report (April 1983).

[25] Spector, A. Z., "Support for distributed transactions in TABS prototype," *IEEE Transactions on Software Engineering*, Vol. 11, No. 6 (June 85), pp. 520–530.

[26] Wei, Z., and Ramamritham, K., "Use of transaction structure for improving concurrency", Technical Report, University of Massachusetts, Amherst (April 1985).

[27] Weihl, W., "Specification and implementation of atomic data types," PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA. (March 1984).

[28] Weihl, W., and Liskov, B., "Implementation of resilient, atomic data types," *ACM Transactions Programming Languages and Systems*, Vol. 7, No. 1 (April 1985), pp. 244–269.

[29] Wood, W. G., "Recovery control of communicating processes in a distributed system," Technical Report 158, University of Newcastle upon Tyne (1980).