CONCURRENCY CONTROL FOR OBJECT ORIENTED PROGRAMMING ENVIRONMENTS

Henry Tirri[1]

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-86-14 June 1986

---

[1]On leave from Department of Computer Science, University of Helsinki, Tukholmankatu 2, SF-00250 Finland.

Abstract

In this paper we address the problem of developing a concurrency control mechanism for a distributed object oriented programming environment which allows several users on workstations to share objects in a transparent manner. We develope an intuitive transaction model for the object oriented environment which incorporates operation (method) inheritance with arbitrarily nested transactions and describe a simple implementation of the concurrency control mechanism based on method specific locking. Our approach can be applied to existing object oriented systems such as Smalltalk.

1 Introduction

An object oriented programming environment such as Smalltalk [Gold 84] provides a single user personal workstation to support the programming process. With the modern day technology these workstations can be connected with a high performance local area network (LAN) which allows users to transfer data between several workstations. Our purpose is to introduce a concurrency control mechanism that allows several users to share objects in this environment in a consistent way. This problem is related to the widely studied field of concurrency control in distributed databases [BerG 81], however in an object oriented environment the operations and the notion of transaction are radically different from the ones in the traditional database framework as described in [Gray 80].

First of all, an execution of an operation on an object defined by a method is triggered by a message sent to the object itself as opposed to the traditional database approach, where operation requests are sent to an agent (e.g. data manager) that performs the operation. Hence in our case data, i.e. objects, is active in the sense that it can perform the operation by itself.

Secondly, since a method may invoke other operations on different objects by sending messages the transactions have a nested structure [BaKK 85], [Lync 83], [Moss 81] with all the problems introduced by subtransactions executing concurrently with their parents.

Finally the number of operation types allowed is much larger than in a conventional database environment, which typically has only four operations: retrieve, update, insert and delete a data item. The ability to distinguish between different operations introduces additional semantic information that can be used by the concurrency control mechanism [KunP 83].

In this paper we address the problem of developing a concurrency control mechanism for a distributed object oriented programming environment where objects can be shared and manipulated concurrently by several users. Our study was inspired by the work of Schwartz and Spector [SchS 84] on synchronization issues that arise when the notion of transaction is extended for shared abstract data types. However, their general study left open the problem of transaction nesting and method inheritance, both of which are essential properties of an object oriented programming environment.

Our work is divided as follows. In Section 2 we present an intuitive

model of the object oriented environment. With the notions introduced in this model in Section 3 we describe a simple mechanism based on locking [EGLT 76] to implement concurrency control for this environment. Our approach can be directly applied to existing object oriented programming environments such as Smalltalk. A simple prototype implementation of the ideas presented has been implemented in Smalltalk itself on a Macintosh microcomputer.

2 Transactions in an object oriented system

Our distributed transaction model differs from the traditional model in several ways. Our model incorporates the concept of an *object* and *methods* specific to that object instead of data items and general operations. Also the details of how objects are represented and methods are implemented are known only to the object's implementor. Hence to develope a transaction model we need to describe briefly the architecture of the environment assumed in the model. This architecture is intended to capture some of the essential features of a distributed environment based on the object oriented paradigm.

2.1 The environment

We assume a graphical, interative programming environment consisting of personal workstations supporting object oriented approach to programming. An excellent example would be a workstation running a single user Smalltalk system. A workstation has a private mass storage device (hard disk) that is used for storing objects.[1]

Secondly these workstations are assumed to be connected to a single high performance local area network that allows data to be exchanged between workstations. To be able to share objects it is assumed that all the objects in the distributed system have a unique id and a message can be sent to any object from any workstation (a performance degradiation is naturally possible if "remote" objects are used).

Macintosh is a trademark licensed to Apple Computer, Inc.

[1] Alternatively workstations can be connected to a file server which handles all the object storing.

Since in this study we are interested only in the concurrency control aspects we assume that the underlying message passing system is reliable. In reality the environment necessarily has a recovery mechanism resembling the ones used in distributed databases. Recovering from failures is a research issue in itself and not a proper topic of this paper.

2.2 The transaction model

As usual, to be able to define transactions we first have to define entities accessed in transactions which in our case are objects. In addition we have to define what the object states are and what operations are possible.

Definition 1 *A class C is a pair (M,V) where M is a set of methods and V is a set of instance variables. A class $C' = (M',V')$ is a subclass of class C if $M \subset M'$ and $V \subset V'$.*

A class describes the implementation of a set of objects that are all represented in the same way. The set of methods (M) describes the operations that can be performed on an object belonging to this class. The set of instance variables (V) describe the type of the value components of the objects state. Subclasses inherit all the methods and instance variables of the superclass (but have additional methods and/or variables).

Definition 2 *An object o described by class C is an instance of the class. We denote this relation by membership, i.e. $o \epsilon C$.*

Before defining operations we have to introduce the notions of object and environment states.

Definition 3 *Let $Dom(v)$ be the value domain of an instance variable $v \epsilon V$. The state s_o of an object o is a mapping $f : V \rightarrow \cup Dom(v)$, $f(v) \epsilon Dom(v)$, $v \epsilon V$.*

Hence state of an object (for our purposes) is simply an assignment of values to the instance variables. The set of all objects o is called *environment.* Ther notion of state is extended for the environments in the obvious way:

Definition 4 *A state S of an environment E is the union $\cup s_o$ for all $o \epsilon E$.*

Definition 5 *An operation $p \epsilon M_o$ is a mapping $p : s_o \rightarrow s_o$ i.e. operation can change the state of one object only.*

Definition 6 *A transaction t is a 3-tuple $(T, P, \prec)$ where T is a set of transactions, P is a set of operations, $\prec$ is a partial order on $T \cup P$. Every transaction is of finite depth.*

In an object oriented programming environment a transaction is a message to an object. This message carries the type of the operation to be performed. The semantics of the operation is described by a method which may involve sending other messages that initiate subtransactions, which initiate new subtransactions etc. At the bottom level, however, the transactions consist only of primitive operations. The definition above defines transactions as arbitrarily nested hierarchical structures. To describe a transaction in terms of atomic operations we use the notion of *transaction closure* introduced in [BaKK 85].

Definition 7 *The closure t^* of a transaction $t = (T, P, \prec)$ is a pair $(P^*, \prec^*)$ where $P^* = P \cup \{p^* \mid (t, p, \prec) \epsilon T\}$ and $\prec^*$ is defined as follows: assume that $r, s \epsilon P^*$. We have $r \prec^* s$ if one of the following conditions holds:*

- *$r, s \epsilon P$ and $r \prec s$,*
- *there exists a transaction $t_i = (T_i, P_i, \prec_i) \epsilon T$ such that $r, s \epsilon P_i^*$ and $r \prec^* s$,*
- *there exists a transaction pair $t_i = (T_i, P_i, \prec_i)$, $t_j = (T_j, P_j, \prec_j) \epsilon T$ such that $r \epsilon P_i^*, s \epsilon P_j^*$ and $t_i \prec t_j$ $(i \neq j)$.*

Transaction closure flattens the transaction hierarchy, but preserves the ordering constraints of the operations. In general the ordering $\prec_*$ remains partial. The nested form of a transaction is more intuitive than its closure. However, the closure has to be introduced so that we are able to define interleaving of the operations of nested transactions.

When a transaction is executed system forces additional ordering among the operations. In a distributed workstation environment several operations can be executed in parallel, but the operations using a same executing agent are serialized. Hence for the definition of a *execution* of a set of transactions we attach an executing *agent* to each operation.

Definition 8 *Let $\mathcal{A}$ be a set of agents and $A : P^* \rightarrow \mathcal{A}$ a mapping that relates an operation p and the executing agent $A(p)$. Then an execution e of a finite set of transactions $T = \{t_i\}$ $(i \geq 1)$ is a pair $(\cup P_i^*, \prec')$ where $\prec'$ is a partial order defined as follows: assume that $r, s \in \cup P_i^*$. Then the following conditions hold:*

- *if $r, s \in P_i^*$ then $r \prec^* s \Rightarrow r \prec' s$,*
- *for each pair of operations (r, s) such that $A(r) = A(s)$ we have either $r \prec' s$ or $s \prec' r$.*

Definition 9 *The effect of an execution e is a finite collection $F = \{f_i\}$ $(i \geq 1)$ where $f_i : S \rightarrow f_i(S)$ is defined as follows: let $p_1, p_2, ..., p_n$ be the ordered list of operations that have the same agent $A(p)$. Then f_i is the composition $p_1 \circ p_2 \circ ... \circ p_n$. The resulting state from an execution e is $S' = \cup f_i(S)$.*

Until this point, we have not imposed any constraints on the transactions or restrictions to the set of possible executions. By definition transactions are usually understood as an abstraction mechanism that allows users to group a collection of operations to form a unit of consistency. Hence transactions are assumed to be consistency preserving in the sense that given a consistent state of the environment the execution of the transaction also leaves the environment to a consistent state. This allows the definition of a very general notion of execution correctness called *serializability* [EGLT 76]. Since transactions are units of consistency any serial (one-by-one) execution trivially preserves the consistency of the environment. Hence any execution that has the same effect on the enviroment (users included) than a serial execution of the transactions involved is correct.

Definition 10 *An execution e is serializable if the effect of the execution F_e is the same as the effect of some serial execution e_s, i.e. $F_e(S) = F_{e_s}(S)$ for some e_s.*

This notion of serializability is used as the correctness criteria against which the mechanisms for concurrency control can be tested. Unfortunately testing for serializability of an execution is an inherently difficult task [Papa 79], hence the efficient mechanisms can only approximate this criteria. However, we require that the mechanism is not allowed to produce non-serializable executions. A mechanism satisfying this condition is *sound.*

3 Method specific locking

The formal framework defined above allows us to describe a wide variety of concurrency control mechanisms. In this Section our purpose is to describe a simple sound concurrency control mechanism that can be efficiently implemented. While we do not describe the mechanism with any specific programming language syntax, we show how it preserves consistency by restricting the interleaving of operations. Filling up the details for a specific environment (e.g. Smalltalk) is a straightforward task and not considered here.

The proposed mechanism is based on *locking*, a method which is widely used in database systems. In a locking mechanism each operation has to lock the object accessed before the operation can be carried out, i.e. a locking provides a mutual exclusion mechanism to restrict the concurrent access to objects. However, locking by itself does not guarantee consistency preservation, additional restrictions have to be imposed on the order how a transaction locks and releases objects. A classic result by Eswaran et al [EGLT 76] shows that under the general assumptions we have adopted consistency preservation requires locks to be acquired in *two-phase manner*: no locks can be released in a transaction before it has all the locks it will ever need.

The basic two-phase locking policy described above, although sound, is a very conservative method that restricts the possible concurrency of the operations drastically. Fortunately this method can be improved by introducing more information for the locking mechanism. Our notion of correctness required only that the effect of the execution should be similar to that of a serial one. Hence there exists operations that can be allowed to interleave with the operations performed by transaction since they do not violate the correctness of the result. A canonical example is a read operation that does not change the state of an object and thus can be interleaved with read operations from other transactions. In locking this additional information is captured by *lock modes* [Kort 83].

The mechanism introduced here, *method specific locking (MSL)* is based on these two simple observations. Hence it suffices to show

- 1) How lock modes can be implemented in an object oriented environment.

- 2) How the two-phase behavior of the transactions can be guaranteed.

After this showing the soundness of the mechanism is easy.

3.1 Implementing method lock modes

By definition each object is associated with a set of methods that describe how the possible operations are performed. To implement method locks the method invoking mechanism has to be augmented with a *compatibility vector*, a *lock vector* and a *lock queue*. These data structures describe the state of an object from the concurrency control point of view and are attached to objects (data structures) representing methods and the object itself.

- A *compatibility vector (cv)* is a bit vector that has a component for each possible object method. Every method m has a compatibility vector of its own which specifies those methods that are incompatible with m by setting the corresponding component value to one.
- A *lock vector (lv)* for each object is a vector of pairs (b, l) that has a component for each possible object method. The b_i components of the pairs describe the method types for which the object is currently locked ($b_i = 1$ if the object is locked in mode method i). An l_i component is a transaction id list denoting transactions that hold the corresponding lock b_i.
- A *lock queue* is a message queue that in addition to the queueing messages also has information which method lock types a message is queueing for.

These data structures can be used to implement method lock modes as follows. When a message is received the compatibility vector of the corresponding method cv_m is compared against the b_i components of the lock vector lv_o of the receiver o bitwise with a logical and operation. If any of the comparisons is true, the corresponding transaction id lists l_i are checked if the lock is held only by the transaction itself in which case the lock conflict is ignored and the lock is granted. Otherwise the object is locked by an incompatible method by some other transaction and the message have to wait in the lock queue releasing of the corresponding lock component(s). If no incompatible lock exists the lock vector is updated to reflect the new

lock status of the object, i.e. the corresponding b component value is set to one, id list l is augmented with the transaction id and the operation is performed. The lock vector is also updated when the object receives a specific *release message*. Setting a lock vector component b to zero initiates search for queueing messages that could be performed. In case of several alternatives to be invoked messages are served in First-Come-First-Served basis.

Compatibility vectors provide a simple tool to implement locking schemes that vary from mutual exclusion to total sharing of objects. It should be observed that this mechanism allows the implementation of concurrency control also with only partial information of the possible methods since unnecessary incompatibilities only decrease concurrency level but do not affect soundness. This is important since objects inherit methods from the superclasses and it is sometimes impractical to require that the introducer of a new method should be aware of semantics of all the inherited methods.

3.2 Implementing two-phase behavior

We have already mentioned that soundness of our locking mechanism depends on the two-phase behavior of transactions. Since our transactions are allowed to be arbitrary nested hierarchical structures a subtransaction can not independently start releasing locks, since this doesn't guarantee global two-phase behavior. In fact requiring global two-phase behavior in our distributed system is related to the general problem of detecting so-called stable properties of distributed systems [ChaL 85], especially termination. The general algorithm presented in [ChaL 85] is viable also in our case where the lock point[2] of a transaction need to be found. However, since our transactions are finite and tree-structured we do not have to be as general as Chandy and Lamport in their study and hence can outline a simpler algorithm to determine the lock point in our special case.

Let us call a receiver of an original transaction message (for transaction t) a *root object* and objects that have not sent subtransaction messages *leaf objects*. The protocol that transactions have to obey when acquiring and releasing locks (*Tree Locking Protocol (TLP)*) can be described in terms of objects responding to messages in the following way. Let m be a message to object o with method p.

[2]Lockpoint is the moment of computation when a transaction has acquired all the locks it requires.

- p = *ready(t)*: if o is not a root object send message m' with method *ready(t)* to the parent object in transaction t. If o is a root object and there are no more messages to invoke subtransactions, send a message m' with method *release* to all son objects (in transaction t).
- p = *release(t)*: if o is not a leaf object send message m' with method *release* to all son objects (in transaction t).
- $p \notin \{ready(t), release(t)\}$: acquire lock in mode p on object o for transaction t.
- A lock on an object can be released if the operation is performed and a *release(t)* message has been received.

Theorem 1 *Tree Locking Protocol guarantees that transaction locks are acquired in a two-phase manner.*[3]

Intuitively *TLP* forces two-phase behavior by requiring the son objects to inform the parent object when they have reached their local lock point. This information is accumulated to the original root object which then informs subtransaction objects when the global lock point has been reached. It should be observed that the objects do not have to wait the end of the execution of the operation to send the *ready(t)* message, they only have to know that no more messages are sent.

The method described above forces serializability of the executions by restricting the possible interleavings by delaying some of the messages. It allows compatible operations to be interleaved freely as the order of these operations does not affect the resulting state of the environment.

Theorem 2 *If the execution of the original transaction terminates with all the messages performed, MSL is sound.*

3.3 Coping with deadlocks

The theorem above states that no inconsistencies are created if *MSL* is used and the computation does not terminate prematurely. However, as with any locking method a possible *deadlock* may cause this early termination.

[3]Proofs are presented in the full version of this paper.

Hence to be practical the implementation of *MSL* has to be able to cope with deadlocks. Due to the distributed nature of our object environment detecting deadlocks is an inherently difficult problem. There exist several distributed deadlock detection algorithms in the literature, many of them are incorrect [GliS 80] and all the proposals involve considerable message overhead which makes them impractical in a high performance programming environment.

There exists at least two ways to avoid using these algorithms: introduction of a centralized object that creates a global data structure to describe the waiting relation between transactions, and approximation of deadlock detection with *time-outs*. The former approach is very inefficient due to the bottleneck created and against the overall distributed approach adopted in the architecture we described in Section 2.1. Hence we choose the latter one and attach a time-out mechanism to each non-leaf receiver object.

If an object does not receive *ready(t)* message from a son object within a prespecified time interval it sends a *clear(t)* message to the corresponding son object and waits for an acknowledgement (a *cleared(t)*) from the object. After this the object waits a randomly chosen time period before resending the original message.

If an object receives a *clear(t)* message it checks the lock vector to find if t holds a lock on the object. If it doesn't the corresponding message is removed from the lock queue and the parent transaction is acknowledged with a *cleared(t)* message. If t holds a lock and the object is a non-leaf object a *clear(t)* message is sent to all son objects. After receiving *cleared(t)* messages from all the son objects the lock vector is updated and the parent object is acknowledged. In the latter case a leaf object only updates the lock vector.

The obvious disadvantage of this simple scheme (and any other time-out scheme) is that it can restart part of the computation unnecessarily even if a deadlock does not exist. The tree structure of our transactions suggests that the time-out value should vary depending on the hierarchy level of the object, i.e. the accepted delay should increase depending on how close the object is to the root object in the transaction tree. This can be implemented for example by keeping track of the level of nesting with an index field in each subtransaction message.

4 Conclusions

In this study we have addressed the problem of developing a concurrency control mechanism for object oriented programming environments. The simple formal transaction model is intended to capture the nature of transaction processing in a workstation based distributed environment; it incorporates hierarchical nested transaction structures and the notion of correctness in this environment.

The framework of the model was used to develope a concurrency control method based on locking called *Method Specific Locking (MSL)*, a variant of a well known two-phase locking policy used in databases. *MSL* profits from the additional information of compatible operations by using several lock modes (method lock modes) and forces global two-phase behavior in a distributed manner with a special protocol. Deadlocks are resolved with a simple time-out mechanism with different levels of delays.

Our study has deliberately ignored reliability issues. A practical transaction mechanism has to be complemented with a distributed object oriented recovery mechanism, an interesting topic for further research.

5 References

[**BaKK 85** / Bancilhon, F., W.Kim and H.Korth, A model of CAD transactions. Proceedings of the 11th International Conference on Very Large Databases, 1985, 25-33.

[**BerG 85** / Bernstein,P. and N.Goodman, Concurrency control in distributed database systems. ACM Computing Surveys, 13:2 (1981), 185-221.

[**ChaL 85** / Chandy,M. and L.Lamport, Distributed snapshots: determining global states of distributed systems. ACM Transactions of Computer Systems, 3:1 (1985), 63-75.

[**EGLT 76** / Eswaran,K., J.Gray, R.Lorie and T.Traiger, The notion of consistency and predicate locks in a database system. Communications of the ACM, 19:11 (1976), 624-633.

[**GliS 80** / Gligor,V. and S.Shattuck, Deadlock detection in distributed systems. IEEE Trans. Softw. Eng. SE-6:5 (1980), 435-440.

[**Gold 84** / Goldberg,A., Smalltalk-80: The interactive programming environment. Addison-Wesley, 1984.

[**Gray 80** / Gray,J. A transaction model. IBM Res. Report RJ2895, IBM Research Lab., San Jose, 1975.

[**Kort 83** / Korth,H., Locking primitives in a database system. Journal of the ACM, 30:1 (1983), 55-79.

[**KunP 83** / Kung,H. and C.Papadimitriou, An optimality theory of concurrency control for databases. Acta Informatica 19:1 (1983), 1-11.

[**Lync 83** / Lynch,N., Multilevel atomicity - a new correctness criterion for database concurrency control. ACM Transactions on Database Systems, 8:4 (1984), 484-502.

[**Moss 81** /Moss,J., Nested transactions: an approach to reliable distributed computing. Ph.D. dissertation, MIT, 1981.

[**Papa 79** / Papadimitriou,C., The serializability of concurrent database updates. Journal of the ACM, 26:4 (1979), 631-653.

[**SchS 84** /Schwartz,P. and A.Spector, Synchronizing shared abstract types. ACM Transactions on Computer Systems, 2:3 (1984), 223-250.