TRANSACTION SYNCHRONISATION IN OBJECT BASES†

Thanasis Hadzilacos
Computer Technology Institute and
Department of Computer Engineering
University of Patras, Greece

Vassos Hadzilacos
Department of Computer Science and
Computer Systems Research Institute
University of Toronto, Canada

ABSTRACT

We propose a formal model of concurrency control in object bases. An object base is like a database except that information is represented in terms of "objects" that encapsulate both data and the procedures through which the data can be manipulated. The model generalises the classical model of database concurrency control: it allows for nested transactions (as opposed to flat transactions) which may issue arbitrary operations (as opposed to just read and write operations). We establish an analogue to the classical serialisability theorem and use it to derive simple proofs of correctness of two concurrency control algorithms for object bases, namely Nested Two-Phase Locking (Moss' algorithm) and Nested Timestamp Ordering (Reed's algorithm).

Concurrency control in object bases can be viewed as a combination of intra-object and inter-object synchronisation. The former ensures that each object's own methods are executed in serialisable fashion; the latter ensures the compatibility of transaction serialisation orders in different objects. This separation allows for the possibility that each object chooses the most suitable concurrency control algorithm for synchronising its own procedures, independently of the algorithms used by other objects, thus enhancing concurrency.

Keywords: Database systems, Transactions, Object-oriented systems, Object bases, Concurrency control, Serialisability, Nested transactions.

January 13, 1998

† This is a revision of a paper that appeared in the *Proc. of the 7th SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pp. 193–200, March 1988, Austin, TX.

1. INTRODUCTION

In this paper we investigate the problem of synchronising transactions in an object base. We begin with an informal account of what an object base is and how it differs from a traditional database. The notions introduced here are formalised later in the paper.

An object base is a collection of objects, much the way a database is a collection of data items. An *object*, for our purposes, consists of a collection of *variables* and a set of procedures, called *methods*, that are the only means of accessing (sensing or modifying) the object's variables. The values of an object's variables at any point in time constitute the object's *state* at that time. The states of all objects comprise the state of the object base. Reflecting the aspect of the world that the object base models, certain states are deemed consistent.

A method operates by executing *local operations* and by sending *messages* to other objects. A local operation affects directly the object's state. A message results in the invocation of a method of the recipient object; when the invoked method completes its execution, a value is returned to the method that sent the message. The invoked method may itself send messages, thereby causing other methods to be invoked. In this way, the execution of a method may exhibit arbitrary nesting.[1] A method execution e_1 is the *parent* of method execution e_2 if e_2 was invoked by a message of e_1. More generally, e_1 is an *ancestor* of e_2 if e_2 was invoked, directly or indirectly (via a sequence of messages), by e_1.

In traditional databases, users manipulate the database by running *transactions*. A transaction is a programme that accesses the database by reading and writing its data items. It is assumed that a transaction, executed in isolation and to completion, maintains the consistency of the database. Because arbitrary interleaving of different transactions' operations may lead to inconsistencies, concurrent executions of transactions are controlled to ensure that they are *serialisable*. An execution of transactions is serialisable if it is equivalent to a serial execution of the same transactions — i.e., an execution in which each transaction runs to completion before the next one begins. Since individual transactions maintain the consistency of the database, by induction, so do serial executions of transactions. Consequently, so do serialisable executions, being equivalent to serial executions.

Similarly, users manipulate an object base through transactions. In this case, a transaction is a programme that accesses the object base by invoking methods of various objects. Recall that methods, too, are programmes that invoke other methods. For uniformity, we assume a fictitious object, called the *environment*, whose methods are the users' transactions. This permits us to use the terms "transaction" and "method" interchangeably. The terms "user transaction" or "top-level transaction" refer to a method of object environment; i.e., a method that has no parent.

---

[1] In contrast to certain models in which objects correspond to "levels of abstraction" (cf. Weikum [1986]), our model is completely general in this respect. For example, it is permissible for a method of object A to call a method of object B which, in turn, may call some other method of object A again.

As in databases, we assume that a user transaction, executed in isolation and to completion, maintains the consistency of the object base. Also as before, we use serialisability as the mechanism for ensuring correctness in the presence of concurrency. Even though the overall setting looks very similar to that of traditional databases, there are some significant differences: (a) transactions are nested; (b) they issue operations that are not necessarily executed in mutual exclusion; and (c) they exhibit internal parallelism.

(a) A user transaction invokes methods (transactions) which may invoke other methods (transactions) and so on. Unlike the traditional database setting, where transactions issue a sequence of simple read and write operations, in object bases we are confronted with *nested transactions* which issue subtransactions of arbitrary complexity (cf. Moss [1981], Traiger [1983]).

(b) The read and write operations issued by database transactions are short enough that they can be processed serially without loss of efficiency. In contrast, object base transactions invoke methods which can themselves be quite long programmes. We can therefore no longer assume, without severely curtailing parallelism, that the operations issued by transactions are executed in mutual exclusion. In particular, we would like to allow multiple invocations of methods of the *same* object to be active simultaneously. These method executions must be synchronised since they may access common variables, namely the object's variables. The principle of serialisability is used to synchronise these methods.

(c) We also want to allow methods to exhibit internal concurrency, again in order to maximise parallelism. In particular, a method should be allowed to send messages, invoking other methods, simultaneously. Method invocations that have a common ancestor must be synchronised so as to present to that common ancestor a serial view of their execution.

In spite of these differences, it is possible to turn the problem of transaction synchronisation in object bases into the problem of transaction synchronisation in databases in the following way. First, we shall view each object as a data item. We shall treat a method invocation as a group of read or write operations on those data items (i.e., objects) that were accessed, directly or indirectly, as a result of that method invocation. Furthermore, we shall require that only one method execution can be active at each object at any one time. With these restrictions, any conventional database concurrency control method (two-phase locking, timestamp ordering, certification etc. — cf. Bernstein *et al.* [1987]) can be employed to synchronise concurrent transactions in the object base. This approach has the virtue of simplicity and may be well-suited to certain environments. It is, for example, the approach taken in the GemStone project and product (cf. Maier and Stein [1987], Purdy *et al.* [1987]). In this paper, however, we are interested in exploring approaches to concurrency control in object bases which take into account their special features and differences from databases as outlined previously. The hope is that this will lead to more efficient techniques.

The remainder of this paper is organised as follows. In Section 2 we propose a conceptual scheme for concurrency control which separates the task into *intra-object* and *inter-object* synchronisation. In Section 3 we present a formal model for representing executions in object bases; we also define what serialisable (i.e., correct) executions are in this context. In Section 4 we present an extension of the Serialisability Theorem of "classical" concurrency control which relates the serialisability of an execution to the acyclicity of a graph (cf. Eswaran *et al.* [1976]). In Section 5 we exhibit the utility of this theorem by using it to derive simple proofs of the correctness of Nested Two-Phase Locking (an algorithm proposed by Moss [1981]) and Nested Timestamp Ordering (an algorithm proposed by Reed [1978]). We also present a corollary to this theorem that we feel justifies our hopes for modular synchronisation mechanisms. We conclude with a description of further work in Section 6.

This work draws on three main sources: "classical" serialisability theory (e.g., Bernstein *et al.* [1987], Papadimitriou [1986]); the theory of nested transactions (e.g., Beeri *et al.* [1986], Lynch and Merritt [1986]); and object-oriented systems (e.g., Stefik and Bobrow [1986]).

2. INTRA-OBJECT AND INTER-OBJECT SYNCHRONISATION

As discussed previously, multiple invocations of an object's methods may be active simultaneously. These methods may operate on common data (the object's variables), and must therefore be synchronised. That is, if we view the object's variables as a database, and the simultaneous method invocations as concurrent transactions, we have to solve the serialisability problem *within* a single object. We call this *intra-object* synchronisation.

It is not difficult to see that simply ensuring serialisability within each object is not, in itself, enough to guarantee serialisability of the overall computation. For instance, there may be two transactions T_1 and T_2, each accessing objects A and B, so that in object A the concurrent computation of the two transactions' method executions serialises T_1 before T_2, while the reverse holds in object B. The effect of such an execution is not the same as running the two transactions serially in either order and the overall computation is therefore not serialisable, even though the computation at each object is. Thus, in addition to intra-object synchronisation, it is also necessary to exercise some *inter-object* synchronisation, whose goal will be to ensure the compatibility of the independent decisions made at each object.

The potential advantage of separating intra- from inter-object synchronisation is that we may be able to allow each object to use, for intra-object synchronisation, the most suitable algorithm depending on its semantics, the implementation of its methods and so on. For example, an object representing a dictionary data type (with methods Lookup, Insert and Delete) might be implemented as a B-tree. Thus, one of the many special B-tree algorithms could be used for intra-object synchronisation by this object (cf. Bayer and Schkolnick [1977], Ellis [1980], Kung and Lehman [1980], Kwong and Wood [1982], Lehman and Yao [1981], Manber and Ladner [1982], Samadi [1976]). That object would enjoy the efficiency of the special algorithm, even though that algorithm is not applicable to other types of objects. Of course, the viability of such a scheme depends on the existence of efficient inter-object synchronisation schemes that can be used with disparate intra-object synchronisation algorithms. Even though we have no definitive answer for this question, our work so far leaves us hopeful that this may indeed be possible.

Weihl [1984] has proposed *local atomicity* as a principle for synchronising transactions in object bases. We briefly discuss the relationship between this principle and our conceptual scheme that views concurrency control in object bases as a two-part task: intra-object and inter-object synchronisation.

Consider a property which pertains to (i.e., is true or false of) concurrent method executions. Such a property is a *local atomicity property* provided that if the concurrent executions of methods in each object separately satisfy the property then the execution of methods over the entire object base is serialisable. For instance, the property "the execution is serialisable" is *not* a local atomicity property, as illustrated by the example given above. On the other hand, the property "conflicting operations are processed in timestamp order" is a local atomicity property since, as it turns out, it guarantees that the execution over the entire object base can be serialised in timestamp order. Weihl has defined various local atomicity properties: dynamic atomicity (a generalisation of two-phase locking), static atomicity (a generalisation of timestamp ordering) and hybrid atomicity. It is important to appreciate the fact that in this scheme all objects must synchronise transactions using the *same* local atomicity property: e.g., they must all use dynamic atomicity; or they must all use static atomicity.

The principle of local atomicity may therefore be viewed as a special instance of our scheme, where all objects are required to use the same intra-object synchronisation mechanism and that mechanism should implement some local atomicity property. This restriction simplifies the task of *inter*-object synchronisation which is then reduced merely to enforcing some straightforward constraints (such as that a method may not issue any operations following its termination operation or, in the case of static atomicity, that operations carry the timestamp of the issuing method). However, the scheme of inter-object and intra-object synchronisation allows for greater generality than is possible with local atomicity: it places less stringent conditions on intra-object synchronisation by letting each object synchronise each methods in any correct — i.e., serialisable — way. The price of greater freedom for each object is the need for more complex and stringent inter-object synchronisation techniques. This is a trade-off worthy of further study.

3. THE MODEL

Definition 1. An *object base* is a set of *objects*. An *object* is a pair (V, M), where V is a set of *variables* and M is a set of *methods*. A method is a set of *method executions* (cf. Def. 4). A mapping associating values to the variables of an object is called a *state* of the object. There is a distinguished object, called the *environment*.

Definition 2. A *local operation* a of an object is a pair (ρ_a, σ_a) where $\rho_a : S \mapsto R$ is a function from the states of the object to *return values* and $\sigma_a : S \mapsto S$ is a function from states to states. A *local step* of object o is a pair (a, v) where a is a local operation of o and v is a value in the range of ρ_a. A *message step* is a pair (m, v) where m specifies an object and one of its methods, and v is a return value.

Comment. *Local operations are atomic operations on the object's variables — such as read, write, increment a variable etc.* $\sigma_a(s)$ *is the function that specifies how the state of the object changes when operation* a *is executed on state* s. $\rho_a(s)$ *is the function that* *specifies the value that operation a returns when executed on state s. The execution of a local operation results in a "local step" which identifies the operation executed and the value returned by the operation. A "message step" is the invocation of a method and the ensuing return value. As we shall see in Def. 4, the execution of a method consists of local steps and message steps.*

If $t = (a, v)$ is a local step, we shall sometimes abuse notation and write ρ_t and σ_t, instead of ρ_a and σ_a; also, $rv(t)$ will denote the return value of step t, in this case v.

Let $\vec{t} = t_1, t_2, \ldots, t_n$ be a sequence of local steps of object o and s be a state of o. We say that $\vec{t}$ is *legal on* s if, for all $1 \leq i \leq n$, $rv(t_i) = \rho_{t_i}(\sigma_{t_{i-1}} \circ \ldots \circ \sigma_{t_1}(s))$. Informally, this means that the sequence of steps $\vec{t}$ represents a possible computation in object o starting at state s.

Definition 3. Let t_1, t_2 be local steps of object o. t_1 *commutes* with t_2 iff for any state s of o such that t_1, t_2 is legal on s then (a) t_2, t_1 is legal on s; and (b) $\sigma_{t_2} \circ \sigma_{t_1}(s) = \sigma_{t_1} \circ \sigma_{t_2}(s)$. t_1 *conflicts* with t_2 if t_1 does not commute with t_2.

Comment. *t_1 conflicts with t_2 if there is a state in which the execution order of the two steps matters in that their transposition either does not represent a possible computation or results in a different final state for the object. Note that according to our definition commutativity and, therefore, conflict are not necessarily symmetric relations.*[2]

Definition 4. A *method execution* or *transaction* of object o is a partial order $(T, \lhd)$ where:

a. T is a set of (local and message) steps. All local steps in T are steps of object o.

b. If $t, t' \in T$ and t conflicts with t' then either $t \lhd t'$ or $t' \lhd t$.

Definition 5. A *history* is a quadruple $h = (E, <, B, S)$ where

1. E is a set of method executions.
2. $<$ is a partial order on the steps of h (i.e., the steps of the method executions in E).
3. B is a function from the message steps of h to method executions in E.
4. S is a set of states, one for each object in the object base.

Comment. *Histories capture the essentials of concurrent transaction executions in object bases, much like schedules or histories do in "classical" concurrency control theory. E is the set of all method executions which participate in the execution. $<$ denotes the temporal order between steps: $t < t'$ signifies that step t is completed before t' is initiated. B indicates the calling pattern of the execution: if t is a message step, $B(t)$ is the method execution to which t results. S specifies the initial states of the objects.*

Not every quadruple of the above type makes sense as a representation of a computation in an object base. Some restrictions must be placed on the components of a history towards that end. Before we give these it is convenient to define some terminology. Consider a history $h = (E, <, B, S)$. $B(t)$ is the *child* of the method execution that contains t. Let $e, e' \in E$. e' is a *descendent* of e if $e' = e$ or e' is a child of a descendent of e. e' is

[2] Herlihy [1987] has defined a notion of conflict based on so-called *serial dependence relations*, which is more general than the notion of conflict defined here.

a *proper descendent* of e if it is a descendent other than e itself. e is a *(proper) ancestor* of e' if e' is a (proper) descendent of e. e is a *top-level* method execution if it has no proper ancestors. e and e' are *incomparable* if neither is a descendent of the other. The *least common ancestor* of a set of method executions $E' = \{e_1, e_2, \ldots, e_n\} \subseteq E$, denoted $lca(e_1, e_2, \ldots, e_n)$, is the method execution $e \in E$ so that e is an ancestor of every $e_i \in E'$ but no proper descendent of e has this property. Such a method execution may not exist for some E', but if one exists it is unique. The use of genealogical terminology in this manner will be justified shortly. We also extend these relations (child, descendent etc.) to steps (rather than executions) in the obvious way: E.g., a step t' is a child of t if t' belongs to $B(t)$, etc.

Definition 6. A history $h = (E, <, B, S)$ is *legal* iff the following conditions hold:

1. B is a 1-1 function and no method execution is a proper ancestor of itself. Further, every top-level method execution belongs to the object environment (cf. Def. 1).
2. The $<$ relation must satisfy:
   a. For every $e = (T, \triangleleft) \in E$, $< \supseteq \triangleleft$.
   b. If t, t' are local steps of h and t conflicts with t' then either $t < t'$ or $t' < t$.
   c. If $t < t'$ and u, u' are descendents of t, t' respectively, then $u < u'$.
3. For every object o there is a topological sort $\vec{t} = t_1, \ldots, t_n$ of the local steps of o in h, consistent with $<$, such that $\vec{t}$ is legal on the initial state s of o in h. The state $\sigma_{t_n} \circ \ldots \circ \sigma_{t_1}(s)$ is the *final state of o after h*.

In connection with Condition 3 we shall show in Theorem 1 below that it does not matter which topological sort of the local steps (consistent with $<$) of each object is chosen. Thus a legal history is well-defined, and the final state of each object in h is unique. Henceforth we restrict our attention to legal histories and shall refer to them simply as histories.

Comment. *Legal histories represent possible concurrent computations, albeit not necessarily correct ones. We explain the motivation for and some consequences of the conditions given above.*

The first condition states that each message results in exactly one method execution and that it is impossible for a method execution to be caused (directly or via a chain of intermediate messages) by one of its own messages. The facts that B is 1-1 and that proper descendent is an acyclic relation imply that the computation represented by a history has a forest-like structure (and thus justify our use of the terms "child", "descendent" etc. to describe that structure). In particular, they imply the existence of top-level executions in any (legal) history. These are the transactions submitted to the object base by users and are therefore method invocations of object "environment".

Relation $\triangleleft$ in a method execution $e = (T, \triangleleft)$ is derived from the algorithmic structure of the method's implementation. Thus, the orderings stipulated by that relation must be respected in any computation involving e. This is stated formally in Condition 2a. Condition 2b requires a history to specify the order of any conflicting operations. This ensures that the computational effect of a history is determinate (cf. Theorem 1). The meaning of Condition 2c becomes clear if we take the point of view that a message is really a surrogate for everything that happens "under" it in the tree structure imposed by B.

The third condition states that the return value mentioned in each step is as it should be, i.e., as the return value function of the operation executed in that step specifies for the state on which the step is applied.

To prove that a legal history is well-defined and the final state of an object in a history unique we need some lemmata.

Lemma 1. *Let $t_1, t_2, \ldots t_n, \tau$ be local steps of object o such that every t_i commutes with τ. For every state s of o, if $t_1, t_2, \ldots, t_n, \tau$ is legal on s then*

(i) $\tau, t_1, t_2, \ldots, t_n$ is legal on s; and

(ii) $\sigma_\tau \circ \sigma_{t_n} \circ \ldots \circ \sigma_{t_1}(s) = \sigma_{t_n} \circ \ldots \circ \sigma_{t_1} \circ \sigma_\tau(s)$.

PROOF: Straightforward induction on n. ■

Let $\vec{t}$ be a sequence of local steps; a permutation $\vec{u}$ of $\vec{t}$ is called *conflict-consistent (with $\vec{t}$)* if whenever t conflicts with and appears before t' in $\vec{t}$, then t appears before t' in $\vec{u}$.

Lemma 2. *Let $\vec{t} = t_1, t_2, \ldots, t_n$ be a sequence of local steps of object o and $\vec{u} = u_1, u_2, \ldots, u_n$ be a conflict-consistent permutation of $\vec{t}$. For every state s of o, if $\vec{t}$ is legal on s then*

(i) $\vec{u}$ is legal on s; and

(ii) $\sigma_{t_n} \circ \ldots \circ \sigma_{t_1}(s) = \sigma_{u_n} \circ \ldots \circ \sigma_{u_1}(s)$.

PROOF: Induction on the number of steps, n. The basis, $n = 1$, is trivial. For the induction step, suppose the Lemma holds for sequences of $n-1$ steps, for some $n > 1$. Let k be such that

$$t_n = u_k. \tag{1}$$

Since $\vec{u}$ is conflict-consistent with $\vec{t}$,

$$u_j \text{ commutes with } u_k, \text{ for all } k < j \leq n. \tag{2}$$

Let $\vec{w} = w_1, w_2, \ldots, w_n$ be the sequence obtained from $\vec{u}$ by moving u_k to the end; more precisely,

$$w_i = \begin{cases} u_i, & \text{if } 1 \leq i < k \\ u_{i+1}, & \text{if } k \leq i < n \\ u_k, & \text{if } i = n. \end{cases} \tag{3}$$

Now consider the sequences $\bar{t} = t_1, \ldots, t_{n-1}$ and $\bar{w} = w_1, \ldots, w_{n-1}$. Clearly, $\bar{w}$ is conflict-consistent with $\bar{t}$ and the two sequences have $n-1$ steps each. By induction hypothesis then,

$$\bar{w} \text{ is legal on } s, \quad \text{and} \tag{4}$$

$$\sigma_{w_{n-1}} \circ \ldots \circ \sigma_{w_1}(s) = \sigma_{t_{n-1}} \circ \ldots \circ \sigma_{t_1}(s). \tag{5}$$

Now for the induction step we must show,

(i) $\vec{u}$ is legal on s, and

(ii) $\sigma_{u_n} \circ \ldots \circ \sigma_{u_1}(s) = \sigma_{t_n} \circ \ldots \circ \sigma_{t_1}(s)$.

First we claim that $\vec{w}$ is legal on s. By (4), it is enough to show that $\rho_{w_n}\big(\sigma_{w_{n-1}} \circ \ldots \circ \sigma_{w_1}(s)\big) = rv(w_n)$. Indeed,

$$\begin{aligned}\rho_{w_n}\big(\sigma_{w_{n-1}} \circ \ldots \circ \sigma_{w_1}(s)\big) &= \rho_{t_n}\big(\sigma_{t_{n-1}} \circ \ldots \circ \sigma_{t_1}(s)\big) && \text{[by (1), (3), (5)]}\\ &= rv(t_n) && \text{[by assumption that } \vec{t} \text{ is legal on } s\text{]}\\ &= rv(w_n). && \text{[by (1) and (3)]}\end{aligned}$$

Now let

$$s' = \sigma_{w_{k-1}} \circ \ldots \circ \sigma_{w_1}(s). \tag{6}$$

Since $\vec{w}$ is legal on s, it follows that

$$w_k, w_{k+1}, \ldots, w_{n-1}, w_n \text{ is legal on } s'. \tag{7}$$

By (7) and (3), $u_{k+1}, u_{k+2}, \ldots, u_n, u_k$ is legal on s'. By (2) and Lemma 1(i), $u_k, u_{k+1}, \ldots, u_n$ is legal on s' and, by (6) and (3), $u_1, \ldots, u_n$ is legal on s, as wanted for (i).

By (7), (2) and Lemma 1(ii), we have

$$\begin{aligned}\sigma_{w_n} \circ \sigma_{w_{n-1}} \circ \ldots \circ \sigma_{w_k}(s') &= \sigma_{w_{n-1}} \circ \sigma_{w_{n-2}} \circ \ldots \circ \sigma_{w_k} \circ \sigma_{w_n}(s')\\ &= \sigma_{u_n} \circ \sigma_{u_{n-1}} \circ \ldots \circ \sigma_{u_{k+1}} \circ \sigma_{u_k}(s'). && \text{[by (3)]}\end{aligned}$$

From this, together with (6) and (3) we get

$$\sigma_{w_n} \circ \ldots \circ \sigma_{w_1}(s) = \sigma_{u_n} \circ \ldots \circ \sigma_{u_1}(s). \tag{8}$$

By (5), $\sigma_{w_{n-1}} \circ \ldots \circ \sigma_{w_1}(s) = \sigma_{t_{n-1}} \circ \ldots \circ \sigma_{t_1}(s)$. By (1) and (3) $w_n = t_n$; therefore, $\sigma_{w_n} \circ \ldots \circ \sigma_{w_1}(s) = \sigma_{t_n} \circ \ldots \circ \sigma_{t_1}(s)$. By (8),

$$\sigma_{t_n} \circ \ldots \circ \sigma_{t_1}(s) = \sigma_{u_n} \circ \ldots \circ \sigma_{u_1}(s),$$

as wanted for (ii). ■

Theorem 1. *A legal history is well-defined and the final state of an object in a history is unique. I.e., in Condition 3 of legal history (cf. Def. 6) it does not matter which topological sort of the local steps of each object is chosen.*

PROOF: Consider any legal history $h = (E, <, B, S)$. Fix an object o and let s be the initial state of o in S. Let $\vec{t} = t_1, t_2, \ldots, t_n$ be a topological sort of the local steps of o in h, consistent with $<$, that is legal on s (such a $\vec{t}$ exists by legality of h). We must show that for any other topological sort $\vec{u} = u_1, u_2, \ldots, u_n$ of the same steps that is also consistent with $<$,

(i) $\vec{u}$ is legal on s; and

(ii) $\sigma_{t_n} \circ \ldots \circ \sigma_{t_1}(s) = \sigma_{u_n} \circ \ldots \circ \sigma_{u_1}(s)$.

Since $<$ orders any pair of conflicting operations and both $\vec{t}$ and $\vec{u}$ are topological orders of $<$, $\vec{u}$ is a conflict-consistent permutation of $\vec{t}$. (i) and (ii) follow, respectively, by Lemma 2(i) and Lemma 2(ii). □

Definition 7. Two histories $h_i = (E_i, <_i, B_i, S_i)$ for $i = 1, 2$ are *equivalent* iff

1. $E_1 = E_2, B_1 = B_2$ and $S_1 = S_2$; and
2. Every object has the same final state in h_1 as in h_2.

Comment. *This definition of history equivalence is stronger than is sometimes necessary, but is sufficiently general for our results. As remarked in Beeri et al. [1986], histories that represent executions which are intuitively equivalent, may have very different structures below the top level. Accordingly, a more liberal definition of equivalence would only require that equivalent histories have "similar" top-level executions. Furthermore, it need not be the case that the final states of each object under the two histories be identical; it is sufficient that they be "equivalent". These two ways of relaxing the definition of equivalence are related to the issues of "operation abstraction" and "state abstraction", respectively (cf. Beeri et al. [1986]), and the use of such a weaker definition of equivalence is important in the study of certain types of algorithms (cf. Shasha [1984]). Since our results do not make use of these two types of abstraction, we adopt the simpler definition of equivalence given above. Pang [1989] has developed a model that incorporates operation and state abstraction.*

Definition 8. A history $h = (E, <, B, S)$ is *serial* iff for any two incomparable $e, e' \in E$, either all steps of e's descendents precede (in $<$) all steps of e''s descendents, or vice versa. A history is *serialisable* iff it is equivalent to a serial history.

Transaction Failures

Because we allow local operations of arbitrary nature, our model is general enough to describe method executions that "abort" due to an exception, a failure or — as we'll see in Section 5 — nuances of synchronisation algorithms. The abortion of a method execution is simply a termination condition, albeit an "abnormal" one, which is reported to the method execution's parent (if one exists) just like a "normal" termination condition would. In other words, we can posit a local operation, called Abort, that a method can invoke as its last operation. The fact that a method execution, invoked by message m, was aborted will be reflected in the return value of m; in this way the parent of an aborted method execution will be informed of the abortion. Informally, the semantics usually associated with the Abort operation are:

(a) An aborted method execution of object o has no effect on the state of o; and

(b) If a method execution aborts then so do all its descendents.

Regarding (b) it is important to emphasise that the abortion of a method execution implies the abortion of its *descendents*, not its ancestors. This is a useful feature because it allows for the following type of interaction: A method M can invoke another method M' to accomplish a certain task. If M' fails and aborts, M is *not* also doomed to failure: it may still try an alternative way of accomplishing the same task.

These semantics can be formally expressed in our model as follows. For any legal history $h = (E, <, B, S)$ we require:

(a) For any object o, let $\vec{t} = t_1, t_2, \ldots, t_n$ be a topological sort of the local steps of o consistent with $<$, $\vec{u} = u_1, u_2, \ldots, u_m$ be the subsequence of $\vec{t}$ consisting of exactly the local steps of o that belong to non-aborted method executions, and s be the initial state of o in S. Then

   i. $\vec{u}$ is legal on s; and

   ii. $\sigma_{t_n} \circ \ldots \circ \sigma_{t_1}(s) = \sigma_{u_m} \circ \ldots \circ \sigma_{u_1}(s)$

(b) If $e \in E$ contains an Abort step then for all message steps m of e, $B(m)$ also contains an Abort step.

4. THE SERIALISABILITY THEOREM

In this section we prove a theorem which establishes a sufficient condition for serialisability. This condition turns out to be very useful in proving the correctness of concurrency control algorithms. Applications of the theorem are presented in the next section. The theorem, which is a generalisation of the well-known result by Eswaran *et al.* [1976], relates the serialisability of a history to the acyclicity of a graph.

Definition 9. The *serialisation graph* of history h, $SG(h)$, is a directed graph whose nodes correspond to the method executions in h and which has an edge $e \to e'$ iff e, e' are incomparable and either

(a) there exist descendents f, f' of e, e' respectively, so that some step of f precedes and conflicts with a step of f'; or

(b) $lca(e, e') = f$ and $t \lhd t'$ where t, t' are the ancestors of (the steps of) e, e' respectively in f.

Comment. *Roughly speaking, the edges of $SG(h)$ capture orderings of method executions that must be obeyed by an equivalent serial history. Edges of type (a) reflect that the equivalent serial history must run conflicting method executions in the order in which the conflicts were manifested in h. Edges of type (b) reflect that the equivalent serial history must respect any order imposed by the algorithmic structure of the method executions.*

Observation. If there is an edge $e \to e'$ in $SG(h)$ there is also an edge $f \to f'$ where f, f' are any distinct ancestors of e, e' respectively.

In the sequel we shall speak of nodes (of $SG(h)$) and method executions interchangeably.

Theorem 2. *If the serialisation graph of h is acyclic then h is serialisable.*

PROOF: We begin with an outline, as the proof is a bit long. First we define a binary relation $\Rightarrow$ between nodes of $SG(h)$, which extends the edges of the graph and defines an order between any two incomparable executions. We then show that this relation is acyclic (Claim 1). Using $\Rightarrow$ we define a history h_s which has the same executions as h except that the steps may be ordered differently. We show that h_s is legal (Claim 4), serial (Claim 5) and equivalent to h (Claim 6). Thus, h is serialisable.

Let the *level* of a node in $SG(h)$ be the number of its proper ancestors in h (thus top-level executions are nodes at level 0). The $\Rightarrow$ relation is defined inductively: We begin with the ordering implied by the edges and proceed, level-by-level, extending $\Rightarrow$, first to a total ordering among all nodes of that level and then inheriting orderings between nodes of that level to their descendents. More precisely, $\Rightarrow$ is the relation at the end of the execution of the following procedure.

> $\Rightarrow := \{(e, e') :$ there is an edge $e \to e'$ in $SG(h)\}$
> **for** $l := 0$ **to** max. level of any node in $SG(h)$ **do**
>   **if** $\Rightarrow$ is acyclic **then**
>   extend $\Rightarrow$ to totally order all level l nodes (in any way)
>   further extend $\Rightarrow$ by setting $f \Rightarrow f'$ for any f, f'
>   that are descendents of level l nodes e, e' with $e \Rightarrow e'\}$
>   **end if**
> **end for**

Claim 1. *$\Rightarrow$ is acyclic.*

Proof of Claim 1: By induction on the iteration number l. Supposing that $\Rightarrow$ is acyclic before iteration l (of the **for** loop in the above procedure) we shall show that $\Rightarrow$ remains acyclic after it. Since $\Rightarrow$ is initially the same as $SG(h)$ which, by assumption, is acyclic, it follows that at the end of the procedure $\Rightarrow$ is acyclic, as wanted. (Thus the **if** statement in the procedure is really a red herring: $\Rightarrow$ will always be acyclic.)

Suppose, by way of contradiction, that before iteration l $\Rightarrow$ is acyclic but afterwards it has a cycle, say $e_0 \Rightarrow e_1 \Rightarrow \ldots \Rightarrow e_{k-1} \Rightarrow e_0$. Let

$$l^* = \begin{cases} level\big(lca(e_0, \ldots, e_{k-1})\big) + 1, & \text{if } lca(e_0, \ldots, e_{k-1}) \text{ exists} \\ 0, & \text{otherwise.} \end{cases}$$

Define a *block* of the cycle to be a maximal set of consecutive nodes along the cycle which have a common level l^* ancestor. Let $B_0, \ldots, B_{m-1}$ be the blocks of the cycle and $b_0, \ldots, b_{m-1}$ be the corresponding level l^* ancestors (i.e., b_i is the common level l^* ancestor of all nodes in B_i). Further, let a_i, z_i be the first and last nodes, respectively, of block B_i (it is possible that $a_i = z_i$). Thus, at the end of iteration l we have $z_i \Rightarrow a_{i\oplus 1}$ for all $0 \leq i < m$.[3]

We shall prove in a moment that this implies that by the end of iteration l^* we also had $b_i \Rightarrow b_{i\oplus 1}$ for all $0 \leq i < m$ — i.e., we had a cycle among level l^* nodes. By the choice of l^*, $m \geq 2$, i.e. the cycle among the b_i's involves at least two nodes. Granting, for now, that

[3] $\oplus$ is addition modulo m.

this is so, we derive the wanted contradiction: First, it must be that $l = l^*$ (because, by choice, l was the *first* iteration in which a cycle was created in $\Rightarrow$). But before iteration l^*, there was no cycle and in particular no cycle among level l^* nodes. The only $\Rightarrow$ relations added between level l^* nodes during that iteration were such as to extend $\Rightarrow$ to a total order among level l^* nodes, contradicting that a cycle involving only level l^* nodes was created.

We return now to the missing step, viz., proving that, for all $0 \leq i < m$, if at the end of iteration l we have $z_i \Rightarrow a_{i\oplus 1}$ then at the end of iteration l^* we have $b_i \Rightarrow b_{i\oplus 1}$. Consider any $0 \leq i < m$.

Case 1: $z_i \Rightarrow a_{i\oplus 1}$ was true before iteration 0. Then $z_i \Rightarrow a_{i\oplus 1}$ is an edge in $SG(h)$ and, by Observation following Definition 9, we have $b_i \Rightarrow b_{i\oplus 1}$ is an edge in $SG(h)$ and therefore $b_i \Rightarrow b_{i\oplus 1}$ was true before iteration 0 and therefore after iteration l^*.

Case 2: Let $l' \leq l$ be the earliest iteration in which $z_i \Rightarrow a_{i\oplus 1}$ was added to $\Rightarrow$. This can only happen because in iteration l' $c_i \Rightarrow c_{i\oplus 1}$ was added, where $c_i, c_{i\oplus 1}$ are the level l' ancestors of $z_i, a_{i\oplus 1}$ respectively. If $l' \leq l^*$ then $b_i \Rightarrow b_{i\oplus 1}$ was also added to $\Rightarrow$ in iteration l', and we are done. If $l' > l^*$ then $c_i, c_{i\oplus 1}$ are proper descendents of $b_i, b_{i\oplus 1}$ respectively. At the end of iteration l^* either $b_i \Rightarrow b_{i\oplus 1}$ or $b_{i\oplus 1} \Rightarrow b_i$ must hold. In the latter case, $c_{i\oplus 1} \Rightarrow c_i$ will also hold after iteration l^*. But then $c_i \Rightarrow c_{i\oplus 1}$ would not have been added to $\Rightarrow$ in iteration l', a contradiction. Hence, $b_i \Rightarrow b_{i\oplus 1}$ must hold after iteration l^*, as wanted. $\blacksquare_{\text{Claim 1}}$

Let the given history be $h = (E, <, B, S)$. Using relation $\Rightarrow$ we construct a history $h_s = (E, <_s^+, B, S)$ where $<_s$ is defined as follows ($<_s^+$ is the irreflexive, transitive closure of $<_s$):

$<_s$ .1 If t, t' are steps of comparable method executions e, e' respectively then $t <_s t'$ iff

(a) one of t, t' conflicts with the other and $t < t'$; or

(b) e is an ancestor of e' and $t \triangleleft t''$, where t'' is the ancestor of step t' in e; or

(c) e' is an ancestor of e and $t'' \triangleleft t'$, where t'' is the ancestor of step t in e'.

$<_s$ .2 If t, t' are steps of incomparable executions e, e' respectively then $t <_s t'$ iff $e \Rightarrow e'$.

Claim 2. *For any incomparable executions $e, e' \in E$, either $e \Rightarrow e'$ or $e' \Rightarrow e$.*

Proof of Claim 2: Let

$$l = \begin{cases} level\big(lca(e, e')\big) + 1, & \text{if } lca(e, e') \text{ exists} \\ 0, & \text{otherwise.} \end{cases}$$

Since e, e' are incomparable, there exist distinct executions f, f' of level l that are ancestors of e, e' respectively. By the end of iteration l at the latest, we shall have $f \Rightarrow f'$ or $f' \Rightarrow f$, and by the definition of $\Rightarrow$, the same relation will hold between any pair of descendents of f, f' — in particular, between e and e'. $\blacksquare_{\text{Claim 2}}$

Claim 3. *If t conflicts with t' and $t < t'$ then $t <_s t'$.*

Proof of Claim 3: Suppose t conflicts with t' and $t < t'$. If t, t' belong to comparable executions then $t <_s t'$ by $<_s$ .1(a). If t, t' belong to incomparable executions, say e, e' respectively, then $e \to e' \in SG(h)$. Therefore, $e \Rightarrow e'$ and, by $<_s$ .2, $t <_s t'$. $\blacksquare_{\text{Claim 3}}$

Claim 4. *h_s is a (legal) history.*

Proof of Claim 4: First observe that $<_s$ is acyclic, for a cycle in $<_s$ would imply a cycle in either $<$ (which is acyclic since h is a legal history) or $\Rightarrow$ (which is acyclic by Claim 1).[4]

Now we have to show that the three legality conditions (cf. Definition 6) are satisfied by h_s. Condition 1 is inherited directly from h (which satisfies it being, by assumption, a legal history). That Condition 2(a) is satisfied follows from $<_s$.1(b) (recall that a method execution is an ancestor of itself). Condition 2(b) is satisfied by $<_s$.1(a) and the assumption that h is legal, if t, t' are steps of comparable executions; and it is satisfied by Claim 2 and $<_s$.2, if they are steps of incomparable executions. For Condition 2(c) we have:

Case 1: t, t' belong to comparable executions e, e' and $t <_s t'$.

Subcase 1a: One of t, t' conflicts with the other. Then t, t' are local steps and have no proper descendents, so we are done.

Subcase 1b: e is an ancestor of e' and $t \lhd t''$ where t'' is the ancestor of t' in e. Then for any descendent u' of t', the same is true and by $<_s$.1(b) we have $t <_s u'$ (1). Now consider any *proper* descendents u, u' of t, t' respectively. The executions f, f' to which u, u' respectively belong, are incomparable. By definition of $SG(h)$ however, there is an edge $f \to f'$ (because $t \lhd t''$) and thus $f \Rightarrow f'$. By $<_s$.2 then, $u <_s u'$ (2). Finally, consider any proper descendent u of t and t' itself. If e' is a proper descendent of e then the execution f to which u belongs and e' are incomparable and by arguing as above we have $u <_s t'$; if $e = e'$ then by $<_s$.1(c) we have $u <_s t'$ (3). (1), (2) and (3) show that for any descendents u, u' of t, t' we have $u <_s u'$, as wanted.

Subcase 1c: e' is an ancestor of e and $t'' \lhd t'$ where t'' is the ancestor of t in e'. This is similar to Case 1b.

Case 2: t, t' belong to incomparable executions e, e' and $t <_s t'$. Then $e \Rightarrow e'$ and for any descendents f, f' of e, e' respectively, we also have $f \Rightarrow f'$. But then for any u, u' descendents of t, t' respectively, we shall have $u <_s u'$, by $<_s$.2.

Finally we come to Condition 3 of legal histories. By assumption h is legal. Thus, for any object o there is a topological sort $\vec{t}$ of the local steps of o in h which is consistent with $<$ and is legal on the initial state of o in h, say s. Let $\vec{u}$ be any topological sort of the local steps of o in h_s that is consistent with $<_s$. By Claim 3, $\vec{u}$ is a conflict-consistent permutation of $\vec{t}$. Therefore, by Lemma 2(i), $\vec{u}$ is legal on s, as wanted. $\blacksquare_{\text{Claim 4}}$

Claim 5. *h_s is serial.*

[4] To see this, break an assumed cycle in $<_s$ into "blocks", grouping together into the same block consecutive steps along the cycle which belong to comparable executions. We couldn't have a single block, or else $<$ would have a cycle. Now, between executions corresponding to successive blocks we must have a $\Rightarrow$ relation holding, thereby yielding a cycle in $\Rightarrow$.

Proof of Claim 5: Let e, e' be incomparable executions. By Claim 2, we can assume, without loss of generality, that $e \Rightarrow e'$. Consequently for any descendents f, f' of e, e', we have $f \Rightarrow f'$. By $<_s$.2 then, $u <_s u'$ for any steps u, u' of f, f' respectively. That is, all steps of all descendents of e precede all steps of all descendents of e'. $\blacksquare_{\text{Claim 5}}$

Claim 6. *h_s is equivalent to h.*

Proof of Claim 6: By definition of h_s, the two histories have the same E, B and S components. It remains to show that they have the same final states for each object. Consider any object o, and let $\vec{t} = t_1, t_2, \ldots, t_n$ be a topological sort of o's steps in h consistent with $<$ and $\vec{u} = u_1, u_2, \ldots, u_n$ be a topological sort of the same steps consistent with $<_s$. By Claim 3, $\vec{u}$ is a conflict-consistent permutation of $\vec{t}$. By Lemma 2(ii), $\sigma_{t_n} \circ \ldots \circ \sigma_{t_1}(s) = \sigma_{u_n} \circ \ldots \circ \sigma_{u_1}(s)$, where s is o's initial state in h_s (and h). $\blacksquare_{\text{Claim 6}}$

This concludes the proof of Theorem 2. $\blacksquare$

5. APPLICATIONS OF SERIALISABILITY THEOREM

Using Theorem 2 we can derive simple proofs of correctness of two popular concurrency control methods for object bases, Nested Two-Phase Locking (N2PL) and Nested Timestamp Ordering (NTO).

5.1 Nested Two-Phase Locking

N2PL, as outlined below, is a generalisation of the synchronisation algorithm employed in the Argus system (cf. Moss [1981], Liskov and Scheifler [1983], Liskov [1988]), embodying some ideas of Weihl [1984].

Associated with each local step t is a lock denoted $L(t)$. *$L(t)$ conflicts with $L(t')$* if t conflicts with t'. A method execution can *acquire* a lock and later *releases* it. Between the times it acquires and releases a lock, a method execution is said to *own* the lock. A method execution e must observe the following rules:

1. e can issue step t only while it owns $L(t)$.
2. e can acquire a lock L only if every method execution which owns a lock that conflicts with L is an ancestor of e.
3. e cannot acquire any lock after releasing one.
4. e cannot release a lock until its children have released all of theirs.
5. When e releases a lock, the lock is immediately acquired by e's parent, if one exists.[5,6]

[5] The lock is forced on the parent; it will be acquired even if the parent has not requested it. Rule 4 ensures that this forced acquisition will not violate rule 3.

[6] The Argus locking protocol requires that a method execution release all its locks at the end; this is slightly stronger than Rules 2 and 3. There are reasons, having to do with failure recovery, that make the more restrictive rule of Argus a good choice.

Theorem 3. *N2PL allows only serialisable executions.*

PROOF: Let h be a history representing an execution that follows the rules of N2PL. We shall prove that $SG(h)$ is acyclic. The result then follows by Theorem 2.

Claim: If $e \to e'$ is an edge in $SG(h)$, then e releases some lock before e' releases any lock.

Proof of Claim: Let $e \to e'$ be an edge in $SG(h)$. By definition of $SG(h)$, e, e' are incomparable and either (a) they have descendents f, f' which contain, respectively, steps t, t' such that t conflicts with t' and $t < t'$; or (b) e, e' are created, respectively, by messages t, t' of the least common ancestor of e, e' and $t \lhd t'$. Consider each case separately.

Case (a): We claim that

$$e \text{ must release } L(t) \text{ before } f' \text{ acquires } L(t'). \tag{$*$}$$

To see this first note that, by rule 1, f must acquire $L(t)$ and f' must acquire $L(t')$. If $lca(f, f')$ does not exist, rules 2 and 5 of N2PL imply that all ancestors of f — e among them — will acquire and release $L(t)$ before f' acquires $L(t')$. If $lca(f, f')$ exists, it cannot be a descendent of e. For, if it was, f' would be a descendent of e, as well as of e', contradicting that e and e' are incomparable. Therefore $lca(f, f')$ is a proper ancestor of e. But then, rules 2 and 5 of N2PL imply that f' can't acquire $L(t')$ until all nodes which are ancestors of f and proper descendents of $lca(f, f')$ — e among them — have obtained and released $L(t)$, proving (*).

By rule 3, f' must acquire $L(t')$ before releasing any lock. If $e' = f'$ then the Claim follows by (*). Otherwise, f' is a proper descendent of e' and thus, by rule 4 and induction, f' must release all locks before e' can release any locks. Therefore, f' must acquire $L(t')$ before e' can release any lock. Combining this with (*) we get that e must release some lock (namely $L(t)$) before e' releases any lock, as wanted.

Case (b): In this case, e finishes before e' starts. Without loss of generality we can assume that e and e' each obtain some lock (if not, we can imagine, for the sake of this argument, that each obtains some lock which it releases immediately — it is not necessary that they obtain the same lock). Since e finished before e' starts, we have, in this case too, that e releases some lock (indeed all of them!) before e' releases any lock. $\blacksquare_{\text{Claim}}$

By induction, the Claim implies that if there is a path from e to e' in $SG(h)$, e releases some lock before e' releases any lock. Therefore, if a cycle existed in $SG(h)$ it would follow that each execution along the cycle releases some lock before it releases any lock, an absurdity. So $SG(h)$ cannot have cycles and h must be serialisable. $\square$

Implementation Considerations

Rule 1 of N2PL appears to be circular: To issue step t, a method execution must own $L(t)$. Recall that a step is a pair (a, v) where a is an operation and v is a return value. Before the actual processing of t, the method execution "knows" the operation a that it wants to perform but does not, in general, know the value v that the operation will return until *after* the step has been processed. How, then, can the method execution "know" which lock to obtain *before* the step is issued?

There are two possible solutions to this apparent circularity. The more common is to associate locks with *operations*, rather than steps. An operation a conflicts with a' if there exist steps $t = (a, v)$ and $t' = (a', v')$ of a and a', respectively, such that t conflicts with t'. A lock $L(a)$ of operation a conflicts with lock $L(a')$ of operation a' if a conflicts with a'. To enforce rule 1, a method execution that wishes to issue operation a must first acquire $L(a)$. As we noted in the preceding paragraph, a method execution "knows" each operation it wants to issue before the corresponding step (i.e., the execution of the operation) is actually carried out; therefore, there is no problem with the method execution "knowing" which locks to acquire. The acquisition and release of operation locks follows rules 2–5 above. In effect, this solution amounts to using a more conservative notion of conflict than necessary. The original source of N2PL (cf. Moss [1981][7]) uses this approach.

The second solution is for a method execution to provisionally issue an operation, observe the resulting return value and, having establish the actual step, acquire the necessary lock. If the lock cannot be acquired without violating rules 2–5, the actual processing of the operation must be delayed until a later provisional execution of the operation results in a step for which a lock can be acquired. Of course, for this to work, the provisional execution of an operation and the attendant acquisition of a lock must be performed atomically with respect to other local operation executions of the same object. This solution has the advantage of using a weaker notion of lock conflict and therefore allowing more parallelism. This is especially so when objects have operations whose executions occasionally conflict but usually do not. For example, in many reasonable representations of queues, an Enqueue conflicts with a Dequeue only if the latter returns the item placed into the queue by the former. Thus, if we locked operations with no regard to their return values, an Enqueue operation would delay *any* Dequeue operation of an incomparable method execution. If, on the other hand, we locked steps, an Enqueue step would only delay those Dequeue steps attempting to remove the item inserted by the Enqueue. The observation that the return values of operations can be exploited to enhance concurrency is due to Weihl [1984]. An alternative proof of the correctness of N2PL, based on the "I/O automata" formalism of Lynch and Merritt [1986], is given by Fekete *et al.* [1987].

[7] Moss' algorithm assumes that the local operations of all objects are Read and Write of variables. The idea readily generalises to arbitrary operations.

5.2 Nested Timestamp Ordering

NTO, sketched below, is a technique proposed by Reed [1978]. Each method execution e has a *hierarchical timestamp*, $hts(e)$, that uniquely identifies it. $hts(e)$ is of the form $\langle a_1, a_2, \ldots, a_k \rangle$, where $\langle a_1, a_2, \ldots, a_{k-1} \rangle$ is the hierarchical timestamp of e's parent (if e is a top-level method execution, $hts(e)$ has a single component). Hierarchical timestamps are totally ordered lexicographically; let $\prec$ denote this order. The NTO algorithm enforces the following rules:

1. If e, e' are incomparable method executions and t, t' are local steps of e, e' respectively, such that t conflicts with t' and $t < t'$ then $hts(e) \prec hts(e')$.
2. If m, m' are message steps of the same method execution and $m \lhd m'$ then $hts(B(m)) \prec hts(B(m'))$.

Theorem 4. *NTO allows only serialisable executions.*

Proof: Let h be a history representing an execution using NTO. Let $e \rightarrow e'$ be an edge in $SG(h)$. Then rule 1 or 2 of NTO (according as the edge is of type (a) or (b) — cf. Def. 9) implies that $hts(e) \prec hts(e')$. Thus, if $SG(h)$ had a cycle $e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n \rightarrow e_1$, we would have $hts(e_1) \prec hts(e_2) \prec \ldots \prec hts(e_n) \prec hts(e_1)$, a contradiction. Thus, $SG(h)$ is acyclic and, by Theorem 2, h is serialisable. □

Implementation Considerations

We sketch how rules 1–2 of NTO can be implemented. First consider rule 2. A counter $\mathbf{ctr}_e$ is maintained for each method execution e. The counter is initialised to zero and is manipulated by operation Increment which, atomically, increments the counter by one and returns its new value. Before e sends the message m that results in method execution $e' = B(m)$, Increment($\mathbf{ctr}_e$) is performed and the hierarchical timestamp assigned to e' is $\langle hts(e), v \rangle$, where v is the value returned by Increment($\mathbf{ctr}_e$). The atomicity of Increment will assure that method executions resulting from messages sent in parallel are assigned unique timestamps the order of which, however, is not predetermined. By contrast, method executions e and e' resulting from messages m, m' issued serially, with m preceding m', will result in the assignment of timestamps such that $hts(e) \prec hts(e')$, as required by rule 2.

Now consider rule 1. When a method execution e wants to issue step t, it must be possible to ascertain, for *every* method execution e' which is incomparable to e and has issued a step t' preceding and conflicting with t, whether $hts(e') \prec hts(e)$. If so, t may be processed; otherwise it must be rejected and e aborted. We are confronted again with the same apparent circularity that we observed in our discussion of N2PL: To determine if an operation may be processed, we need to know the resulting step (so that we can tell with which earlier steps it conflicts and therefore whether the corresponding timestamps are properly ordered); but the step — in particular, its return value — can't be known until *after* the operation has been processed. The remedies are similar to those described for N2PL.

First, we can adopt the conservative approach of testing for conflict between operations, rather than steps. This can be accomplished as follows. For each local operation

a of each object, we keep the maximum timestamp of any method execution that has issued operation a; this is denoted as $hts(a)$. When method execution e wishes to issue an operation a we check whether $hts(a') \prec hts(e)$ for *every* operation a' that conflicts with a. If so, $hts(a)$ is set to the maximum of its present value and $hts(e)$ and e is allowed to execute a; otherwise, e is aborted.

Alternatively, we can use the idea of provisionally executing an operation to determine the resulting step and hence whether the step can, in fact, be allowed to stand without violating rule 1. However, we are now faced with another difficulty: To determine of the step does *not* violate rule 1, we must somehow "remember" every step executed so far and the timestamp of the method execution to which it belongs.[8] To make this approach feasible, a mechanism is needed which will render some of this information obsolete and will allow us to "forget" it. There is a very simple mechanism that can accomplish this: First, we shall require that the object environment assigns timestamps to (top-level) method executions so that if e terminates before e' begins then $hts(e) \prec hts(e')$. A counter, maintained by object environment and incremented each time a method execution is initiated, will suffice for this. Call a method execution *active* if its top-level ancestor has not yet terminated. Then information about the steps of an inactive method execution e can be discarded as soon as it is the case that for *all* active method executions e', $hts(e) \prec hts(e')$.

5.3 Separating Inter- and Intra-object Synchronisation

$SG(h)$ captures the serialisation order of method executions in the overall computation. We now define, for each object, two graphs that capture the serialisation order of that object's method executions: one of them, SG_{local} reflects orderings implied by conflicts between local steps accessing the object's variables; the other, SG_{mesg} reflects orderings implied by conflicts between messages (manifested by conflicts between local steps of descendents of the messages).

Definition 10. Let h be a history and o an object. $SG_{local}(h, o)$ is a directed graph whose nodes correspond to the method executions of object o in h and which has an edge $e \rightarrow e'$ iff the corresponding method executions are incomparable and some step of e precedes and conflicts with some step of e'. $SG_{mesg}(h, o)$ is a directed graph with the same nodes as $SG_{local}(h, o)$ and an edge from e to e' iff e, e' are incomparable and there are proper descendents f, f' of e, e' so that (f, f') is an edge in $SG_{local}(h, o')$ for some object o'.

Theorem 5. *$h = (E, <, B, S)$ is serialisable provided:*

a. *for every object o, $SG_{local}(h, o) \cup SG_{mesg}(h, o)$ is acyclic; and*
b. *for any $e \in E$, the relation $\rightarrow_e$ is acyclic, where $u \rightarrow_e u'$ iff u, u' are distinct messages of e and either $u \lhd u'$ or there are conflicting descendents t, t' of u, u' so that $t < t'$.*[9]

[8] Note that the previous, conservative, approach did not face this problem: the timestamps of all earlier steps were "summarised" into a fixed, and relatively small, number of timestamps — one for each type of operation that can be applied to an object.

[9] Note that the relation $\rightarrow_e$ is between messages of *the same* method execution.

PROOF: We shall prove that if $SG(h)$ contains a cycle then either (a) or (b) is violated. The result then follows by Theorem 2.

Let $e_0 \to e_1 \to \ldots \to e_{k-1} \to e_0$ be a cycle in $SG(h)$. Let o be any object so that all of $e_0, \ldots, e_{k-1}$ have, respectively, ancestors $f_0, f_1, \ldots, f_{k-1}$ which are method executions of o. (At least one such object, the environment, exists.) By Observation following Def. 9 it follows that, for all $0 \leq i < m$, if $f_i \neq f_{i\oplus 1}$ then $f_i \to f_{i\oplus 1}$ is an edge in $SG(h)$. Since $f_i, f_{i\oplus 1}$ are method executions of the same object o, $f_i \to f_{i\oplus 1}$ is an edge in either $SG_{local}(h, o)$ or $SG_{mesg}(h, o)$. Thus, unless all of $f_0, f_1, \ldots, f_{k-1}$ are identical, there is a cycle in $SG_{local}(h, o) \cup SG_{mesg}(h, o)$, violating (a).

If all of $f_0, f_1, \ldots, f_{k-1}$ are identical then $e_0, e_1, \ldots, e_{k-1}$ have a common ancestor. Let $e = lca(e_0, \ldots, e_{k-1})$. Let a *block*[10] in the cycle $e_0 \to e_1 \to \ldots \to e_{k-1} \to e_0$ be a maximal sequence of consecutive nodes whose steps are descendents of the same message step of e, and let $B_0, B_1, \ldots, B_{l-1}$ be the blocks of the cycle and $m_0, m_1, \ldots, m_{l-1}$ be the corresponding messages of e (i.e., m_i is the common ancestor of (the steps of) all method executions in block B_i). Further, let a_i, z_i be the first and last nodes of block B_i, respectively (it is possible that $a_i = z_i$). Because e is the *least* common ancestor of $e_0, \ldots, e_{k-1}$, there must be at least two blocks; i.e., $l \geq 2$.

Now, for all $0 \leq i < l$, $z_i \to a_{i\oplus 1}$ is an edge of $SG(h)$. By definition of $\to_e$, $m_i \to_e m_{i\oplus 1}$. Therefore, there is a cycle in $\to_e$, violating (b). $\square$

Ensuring the acyclicity of SG_{local} is the job of intra-object synchronisation. Ensuring the acyclicity of SG_{mesg} and $\to_e$ is the job of inter-object synchronisation. Condition (a) expresses the requirement that intra- and inter-object synchronisation be compatible. Condition (b) is needed because our model allows concurrency within a method execution: Unless it holds, two concurrent messages may result in two pairs of conflicting steps, each pair requiring the serialisation of the concurrent messages in the opposite order than the other; this would make it impossible to order the two messages in an equivalent serial execution.

6. FURTHER WORK

In this paper we reported some initial results of on-going research on transaction processing in object bases.

From a practical standpoint, the most interesting problem is to design general and efficient mechanisms for inter-object synchronisation. Theorem 5 states precisely what is required of such mechanisms. It is not hard to see that it is possible to enforce the conditions of Theorem 5. However, we are interested in mechanisms which (i) require as little communication between objects as possible and (ii) do not overly restrict intra-object synchronisation. Our work so far suggests that these goals are conflicting and that there is a question of trade-offs. For example, there are techniques that resemble certifiers (or "optimistic" schedulers) in conventional database concurrency control which favour (ii) at the expense of (i) — and the increased danger of scheduling errors requiring abortions. At

[10] The notion of "block" used here is similar to, but not quite the same as, that used in the proof of Theorem 2.

the other extreme, methods like N2PL or NTO require little communication but restrict intra-object synchronisation by imposing a uniform policy over all objects.

To the extent this approach proves feasible, the next question to consider is how to make it usable in a real object base management system. The main issue here is to design a facility that will allow the programmer of an object to define its (intra-object) synchronisation algorithm, including the possibility of selecting such algorithms out of a "standard library". The system itself will provide the inter-object mechanisms.

Finally, another issue that deserves further study is whether certain organisational principles that are common in object bases, such as inheritance and aggregation, can be exploited to make the task of transaction synchronisation in object bases more efficient.

ACKNOWLEDGEMENTS

The research reported herein was supported by grants from the General Secretariat of Research and Technology of Greece and the Natural Sciences and Engineering Research Council of Canada. The first author gratefully acknowledges the hospitality of the Department of Computer Science, University of Toronto, during a visit to which most of this work was carried out. We thank an anonymous referee whose comments provided the impetus for strengthening the results of a previous draft.

REFERENCES

- Bayer, R. and Schkolnick, M. "Concurrency of operations on B-trees". *Acta Inf.*, 9:1–21, 1977.
- Beeri, C., Bernstein, P.A. and Goodman, N. "A model for concurrency in nested transaction systems". *JACM* (to appear).
- Bernstein, P.A., Hadzilacos, V. and Goodman, N. *Concurrency control and recovery in database systems.* Addison-Wesley, Reading, MA, 1987.
- Ellis, C.S. "Concurrent search and inserts in 2-3 trees". *Acta Inf.*, 14(1):63–86, 1980.
- Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. "The notions of consistency and predicate locking in a relational database system". *CACM*, 19(11):624-633, Nov. 1976.
- Fekete, A., Lynch, N.A., Merritt, M. and Weihl, W.E. "Nested transactions and read/write locking". *Proc. of the 6th ACM Symp. on Princ. of Database Systems*, pp. 97–111, March 1987.
- Herlihy, M. "Extending multiversion time-stamping protocols to exploit type information". *IEEE Trans. on Computers*, C-36(4):443–448, Apr. 1987.
- Kung, H.T. and Lehman, P.L. "Concurrent manipulation of binary search trees". *TODS*, 5(3):339–353, 1980.
- Kwong, Y.S. and Wood, D. "Method for concurrency in B-trees". *IEEE TSE*, SE-8(3):211–223, 1982.

- Lehman, P.L. and Yao, S.B. "Efficient locking for concurrent operations on B-trees". *TODS*, 6(4):650–670, Dec. 1981.
- Liskov, B. "Distributed programming in Argus". *CACM*, 31(3):300-312, March 1988.
- Liskov, B. and Scheifler, R. "Guardians and actions: Linguistic support for robust, distributed programs". *TOPLAS*, 5(3):381–404, July 1983.
- Lynch, N.A. and Merritt, M. "Introduction to the theory of nested transactions". TR-367, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1986.
- Maier, D. and Stein, J. "Development and implementation of an object-oriented DBMS". Manuscript, 1987.
- Manber, U. and Ladner, R.E. "Concurrency control in a dynamic search structure". *Proc. of the 1st ACM Symp. on Princ. of Database Systems*, pp. 268–282, March 1982.
- Moss, J.E.B. *Nested transactions: An approach to reliable distributed computing.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1981.
- Pang, R.S.F. *Using abstraction to synchronise transactions in object bases.* M.Sc. thesis, Dept. of Computer Science, University of Toronto, 1989.
- Papadimitriou, C.H. *The theory of database concurrency control.* Computer Science Press, Rockville, MD, 1986.
- Purdy, A., Schuchurdt, B. and Maier, D. "Integrating an object-server with other worlds". *TOOIS*, 5(1):27–47, Jan. 1987.
- Reed, D.P. *Naming and synchronization in a decentralized computer system.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1978.
- Samadi, B. "B-trees in a system with multiple users". *IPL*, 5(4):107–112, 1976.
- Shasha, D.E. *Concurrent algorithms for search structures.* Ph.D. thesis, Harvard University, Cambridge, MA, 1984.
- Stefik, M. and Bobrow, D.G. "Object-oriented programming: Themes and variations". *AI Magazine*, 6(4):40–62, 1986.
- Traiger, I.L. "Trends in system aspects of database management". *Proc. of the 2nd Int'l Conf. on Databases*, Aug.–Sep. 1983.
- Weihl, W.E. *Specification and Implementation of Atomic Data Types.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1984.
- Weikum, G. "A theoretical foundation of multi-level concurrency control". *Proc. of the 5th ACM Symp. on Princ. of Database Systems*, pp. 31–42, March 1986.