Data-dependent Concurrency Control and Recovery (Extended Abstract)

Computation Structures Group Memo 228
June 1983

William E. Weihl

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract no. N00014-75-C-0661, and in part by the National Science Foundation under grant MCS79-23769.

Data-dependent

Concurrency Control and Recovery

(*Extended Abstract*)

William E. Weihl

Massachusetts Institute of Technology

Abstract

Maintaining the consistency of long-lived, on-line data is a difficult task, particularly in a distributed system. A variety of researchers have suggested *atomicity* as a fundamental organizational concept for such systems. In this paper we present a formal treatment of atomicity. Our treatment is novel in three respects: First, we treat serializability and recoverability together, facilitating the precise analysis of online implementations. Second, we explore how to analyze user-specified semantic information to achieve greater concurrency. Third, we focus on local properties of components of a system, thus supporting modular design. We present three local properties, verify that they ensure atomicity, and show that they are optimal. Previously published protocols are suboptimal. We show that these differences are the result of fundamental limitations in the model used to analyze those protocols; these limitations are not shared by our model.

1. Introduction

There are many applications in which the manipulation and preservation of long-lived, on-line data is of primary importance. Examples of such applications are banking systems, airline reservation systems, office automation systems, database systems, and various components of operating systems. A major issue in such systems is preserving the consistency of on-line data in the presence of concurrency and hardware failures. In this paper we are concerned with how to define data objects that help provide needed consistency.

To support consistency it is useful to make the activities that use and manipulate the data *atomic* (see [Liskov & Scheifler 82] and [Reed 78], among others, for discussion and motivation of the use of atomicity). Atomic activities are referred to as *actions* or *transactions*; they were first identified in work on databases [Davies 73, Davies 78, Eswaren 76]. Atomic actions are characterized informally by two properties: serializability and recoverability. *Serializability* means that the actions appear to execute in some sequential order, even though they actually execute concurrently. *Recoverability* means that each action appears to be all-or-nothing: either it executes to completion and commits, or it aborts and appears to have no effect on data shared with other actions. Recoverability allows the user on whose behalf an action runs to decide to discard the effects of that action. In addition, it allows an action to work properly even if there is a hardware failure during its execution and allows the system to abort an action if necessary (e.g., to resolve a deadlock). Serializability in databases has received a great deal of attention in the theoretical literature (see, for example, [Papadimitriou 79]); recoverability has received less.

In this paper we present a formal treatment of atomicity. Our treatment is novel in three respects:

- It is integrated: we treat serializability and recoverability together, rather than analyzing only serializability or only recoverability.

- It is data-dependent: our definition of atomicity is based on an explicit specification of the semantics of the data.

- It supports modular design: we focus on local properties of individual data objects that ensure global atomicity of the activities using the objects.

Others (see, for example, [Bernstein 81], [Korth 81], and, more recently, [Schwarz & Spector 82]) have also recognized the utility of analyzing specifications of the semantics of data objects to achieve greater concurrency. Existing approaches, however, suffer from two limitations that are not shared by our approach. First, they have a fixed model of recovery, and formally treat only serializability. As we will illustrate in Section 5, this biases the specifications of objects toward certain recovery techniques. This bias limits concurrency unnecessarily, and makes it difficult to analyze novel recovery techniques such as those proposed by Reed [Reed 78]. Second, their specifications require operations to be functions, precluding the description of non-deterministic operations. Non-deterministic operations are useful for avoiding over-specification during the design of a system; in addition, as we illustrated in [Weihl & Liskov 83], non-determinism may be needed to achieve a reasonable level of concurrency among actions.

Our focus on local properties is useful when applying a design methodology based on the use of abstract data objects (cf. [Liskov & Zilles 74]). Since the synchronization and recovery needed for an object may depend heavily on the semantics of the object, it is important that the code that performs the synchronization and recovery not be scattered throughout each activity; rather, it should be encapsulated within the implementation of each data object. The local properties that we present in this paper support this kind of encapsulation.

The rest of this paper is organized as follows: In Section 2, we discuss our system model. In Section 3, we use the model to define atomicity. Then, in Section 4, we explore properties of individual objects that ensure atomicity of activities using the objects. We consider three such local properties: a generalization of common locking protocols, which we call *dynamic atomicity*; a generalization of the timestamp-based multi-version protocol proposed by Reed [Reed 78], which we call *static atomicity*; and a novel hybrid approach that incorporates features of both locking and timestamp protocols. Next, in Section 5, we discuss related work and illustrate the problems with existing approaches. Finally, in Section 6, we summarize our results and discuss their significance.

2. System Model

We view a system as composed of activities and objects. *Activities* correspond roughly to processes or threads of control: they are the active entities in the system, and perform tasks for users. *Objects* contain the state of the system: they provide operations by which activities can examine and modify the system state, and constitute the sole path by which activities can pass information among themselves.

We model a computation as a finite sequence of events. (More precisely, our sequences may be considered as *observations* of a computation; for our purposes it is sufficient to consider all possible observations of a system rather than actual computations, which may be more properly viewed as partial orders.) For the remainder of this section, Section 3, and the first part of Section 4, we assume that an event is either the invocation of an operation on an object by an activity, the termination of an invocation, the commit (successful completion) of an activity at an object, or the abort (unsuccessful completion) of an activity at an object. Each event identifies the activity and the object that participated in it. In Sections 4.2 and 4.3 we will augment our model with additional events that introduce information about timestamps for activities.

For example, suppose *x* is an object that is intended to behave like a set of integers, with operations to insert an integer in *x*, to delete an integer, and to check for membership. If *a* is an activity, example events include the following:

- *a* invokes insert on *x* with argument 3 (written <insert(3),x,a>)

- an invocation of insert by *a* on *x* terminates (written <ok,x,a>)

- *a* invokes member on *x* with argument 7 (written <member(7),x,a>)

- an invocation of member by *a* on *x* terminates with result "true" (written <true,x,a>)

- *a* commits at *x* (written <commit,x,a>)

If *a* and *b* are activities, the following event sequence might be a computation of a system containing a set object *x*:

<insert(3),x,a>
<ok,x,a>
<member(3),x,b>
<commit,x,a>
<true,x,b>
<commit,x,b>

If *h* is an event sequence and *x* is an object, we define *h*|*x* to be the subsequence of *h* consisting of all events in which *x* participates. We define *h*|*a* similarly for an activity *a*.

Not all event sequences make sense as computations: activities are intended to act like sequential processes. Thus, we restrict our attention to event sequences *h* satisfying the following conditions:

- An activity must wait until one invocation terminates before invoking another operation.

- No activity both commits and aborts in *h* (at the same or different objects).

- An activity cannot commit if it is waiting for an invocation to terminate.

- An activity cannot invoke any operations after it commits.

Such event sequences will be called *well-formed*; in the remainder of this paper we will only be concerned with well-formed sequences.

The objects in our model are abstractions; the interpretation of the results of their operations is determined by their specifications. We assume that the specification of an object *x* describes the permissible sequences of events involving *x*; thus a specification of an object *x* is a set of well-formed event sequences *h* such that every event in *h* involves *x*. For instance, if *x* is an integer set object, the example sequence above might be in the specification of *x*. On the other hand, the following sequence would probably not be in the specification of *x*:

<insert(3),x,a>
<ok,x,a>
<member(3),x,b>
<commit,x,a>
<false,x,b>
<commit,x,b>

Similarly, each activity has a specification describing the sequences of events in which it can participate. In this paper we will assume that specifications are given for each object and activity in a system; details of how to write such specifications will be published in [Weihl 83].

Given specifications for each object and activity in a system, the possible computations of the system are defined to be all well-formed sequences *h* such that, for every object *x* and activity *a*, $h|x$ is permitted by the specification of *x* and $h|a$ is permitted by the specification of *a*.

3. Global Atomicity

In Section 1 we gave an informal definition of atomicity, characterizing it by two properties, serializability and recoverability. Now that we have described precisely our model of computation, specifications of objects and activities, and how the specifications of the components of a system determine the possible computations of the system, we can make the notions of serializability and recoverability more precise.

We begin by defining serializability. We say that two sequences *h* and *k* are *equivalent* if every activity has the same view in *h* as in *k*; i.e., if $h|a = k|a$ for every activity *a*. We also say that a sequence is *acceptable* in a system if it is permitted by the specifications of the objects and activities in the system. We then say that a sequence is *serializable* if it is equivalent to an acceptable serial sequence (one in which events for different activities are not interleaved). In addition, if *T* is a total ordering of activities, we say that a sequence is *serializable in the order T* if it is equivalent to an acceptable serial sequence in which the activities appear in the order *T*. If no information about the semantics of the operations on objects is available, the serial sequences may be characterized by assuming a free interpretation for the operations as in [Papadimitriou 79]. A similar approach is commonly used when operations are classified as read and write operations. We want to analyze the concurrency that is permissible when we have complete information about the semantics of the operations on objects; thus, we assume we have an explicit description of the acceptable sequences for each object.

For example, an integer set object *x*, with its initial state being the empty set, might allow the following serial sequence:

<insert(3),x,b>
<ok,x,b>
<commit,x,b>
<member(3),x,a>
<true,x,a>
<commit,x,a>

On the other hand, *x* would probably not allow the following serial sequence:

<insert(3),x,b>
<ok,x,b>
<commit,x,b>
<member(3),x,a>
<false,x,a>
<commit,x,a>

This defines serializability; we now define atomicity. If *h* is a well-formed event sequence, let *perm*(*h*) be the subsequence of *h* consisting of all events involving activities that commit in *h*, and no others. We say that *h* is *atomic* if *perm*(*h*) is serializable. This formalizes recoverability by throwing away events for non-committed activities, and requiring that the committed activities be serializable. This is similar to the definition of serializability in [Papadimitriou 79], where it is assumed that some underlying recovery mechanism handles aborts of activities, and the formal analysis considers only events for committed activities. It is different in that we include events for aborted and active activities in our formal model; this facilitates the precise description of online support for recoverability.

For example, if *x* is an integer set and *h* is the sequence

<member(3),x,a>
<insert(3),x,b>
<ok,x,b>
<true,x,a>
<commit,x,b>
<delete(3),x,c>
<ok,x,c>
<commit,x,a>
<abort,x,c>

then *perm*(*h*) is the sequence

<member(3),x,a>
<insert(3),x,b>
<ok,x,b>
<true,x,a>
<commit,x,b>
<commit,x,a>

which is equivalent to the following serial sequence:

<insert(3),x,b>
<ok,x,b>
<commit,x,b>
<member(3),x,a>
<true,x,a>
<commit,x,a>

This serial sequence is permitted by *x*, so *h* is atomic. On the other hand, the sequence

<member(2),x,a>
<true,x,a>
<commit,x,a>

is not atomic: since *x* is initially empty, the member operation cannot return "true" in a serial sequence unless the queried element was inserted by a previous operation.

4. Local Atomicity Properties

We are interested in ways of ensuring that all possible computations of a system are atomic. As discussed in Section 2, the possible computations of a system are determined by the specifications of the components of the system. In this section we investigate several properties of individual objects that ensure atomicity of activities using the objects. We call such properties local atomicity properties. More precisely, a *local atomicity property* is a property ***P*** of specifications of objects such that the following is true: If the specification of every object in a system satisfies ***P***, then every computation of the system is atomic. The remainder of this section is divided into three subsections, each of which presents a different local atomicity property.

4.1. Dynamic Atomicity

Our first local atomicity property, which we call dynamic atomicity, is a generalization of the locking protocols of [Bernstein 81], [Korth 81], and [Schwarz & Spector 82]. Locking protocols are based on the notion of *dependencies* between operations: informally, we say that operation P depends on operation Q if P is executed after Q, and P and Q cannot be reordered without affecting the results of P or Q. We say that activity A depends on activity B if an operation executed by A depends on an operation executed by B. Locking protocols work by preventing one activity from depending on another unless the latter has committed. For example, locking protocols generally prevent one activity from reading data written by another activity until the latter has committed. Similarly, an activity cannot write data read by another activity until the latter has committed.

We can describe dynamic atomicity precisely as follows. If *h* is an event sequence, define *precedes*(*h*) to be the following relation on activities: <*a*,*b*>∈*precedes*(*h*) if and only if there exists an operation invoked by *b* that terminates after *a* commits. For example, if *h* is the sequence

<insert(2),x,a>
<ok,x,a>
<member(3),x,b>
<false,x,b>
<commit,x,b>
<commit,x,a>

then *precedes*(*h*) is the empty relation, while if *h* is the sequence

<insert(2),x,a>
<ok,x,a>
<member(3),x,b>
<commit,x,a>
<false,x,b>
<commit,x,b>

then *precedes*(*h*) contains the pair <*a*,*b*>. Note that if *h* is well-formed then *precedes*(*h*) is a partial order.

Intuitively, dynamic atomicity allows an activity *b* to depend on an activity *a* in a computation *h* only if <*a*,*b*>∈*precedes*(*h*). Thus, if *a* and *b* are not related by the *precedes* relation, there must be an equivalent serial sequence containing *a* followed by *b*, and another equivalent serial sequence containing *b* followed by *a*. More precisely, we say that a sequence *h* is *dynamic atomic* if *perm*(*h*) is serializable in every total order consistent with *precedes*(*h*). In other words, every serial sequence equivalent to *perm*(*h*), with the activities in an order consistent with *precedes*(*h*), must be allowed by the specification of the permissible serial sequences.

For example, the following sequence *h* is atomic:

<member(3),x,a>
<insert(3),x,b>
<ok,x,b>
<false,x,a>
<member(3),x,c>
<commit,x,b>
<true,x,c>
<commit,x,a>
<commit,x,c>

However, it is not dynamic atomic, for the following reason. *Perm*(*h*), which is the same as *h*, is equivalent to the following acceptable serial sequence:

<member(3),x,a>
<false,x,a>
<commit,x,a>
<insert(3),x,b>
<ok,x,b>
<commit,x,b>
<member(3),x,c>
<true,x,c>
<commit,x,c>

and thus is serializable in the order *a* followed by *b* followed by *c* (written *a-b-c*). However, since *precedes*(*h*) contains only the single pair <*b*,*c*>, *perm*(*h*) must also be serializable in the orders *b-a-c* and *b-c-a*. This is not the case; for example, the serial sequence

<insert(3),x,b>
<ok,x,b>
<commit,x,b>
<member(3),x,a>
<false,x,a>
<commit,x,a>
<member(3),x,c>
<true,x,c>
<commit,x,c>

is not acceptable. Informally, we might say that *b* depends on *a*, since *a* must be serialized before *b*.

As another example, the sequence

<member(2),x,a>
<insert(3),x,b>
<ok,x,b>
<false,x,a>
<member(3),x,c>
<commit,x,b>
<true,x,c>
<commit,x,a>
<commit,x,c>

is dynamic atomic. *Precedes*(*h*) contains the single pair <*b*,*c*>, and *perm*(*h*) is serializable in the orders *a-b-c*, *b-a-c*, and *b-c-a*. Informally, we might say that *a* does not depend on *b* or *c*, and that *c* depends on *b*. Since <*b*,*c*>∈*precedes*(*h*), *c* can depend on *b*.

We say that an object is *dynamic atomic* if every sequence permitted by the object's specification is dynamic atomic. The following theorem justifies our claim that dynamic atomicity is a local atomicity property:

Theorem 1: If every object in a system is dynamic atomic, then every computation of the system is atomic.

Proof: The theorem follows from the following two technical lemmas, which follow easily from the definitions:

Lemma 2: If *h* is a well-formed sequence and *x* is an object, then *precedes*(*h*|*x*) ⊆ *precedes*(*h*).

Lemma 3: If *h* is a well-formed sequence, *h* is serializable in the order *T* if and only if, for every object *x*, *h*|*x* is serializable in the order *T*.

Now, suppose every object in a system is dynamic atomic, and let *h* be a computation of the system. Since *h* is well-formed, *precedes*(*h*) is a partial order; let *T* be a total order of the activities in *h* that is consistent with *precedes*(*h*). By Lemma 2, *precedes*(*h*|*x*) ⊆ *precedes*(*h*), so *T* is also consistent with *precedes*(*h*|*x*) for every *x*. Since each object is dynamic atomic, *perm*(*h*|*x*) is serializable in every total order consistent with *precedes*(*h*); in particular, it is serializable in the order T. By Lemma 3, *perm*(*h*) is serializable in the order T. Thus, *h* is atomic.

This description of dynamic atomicity can be used to prove the correctness of common synchronization and recovery techniques, including a combination of the locking protocols of [Bernstein 81], [Korth 81], and [Schwarz & Spector 82] and the intentions lists of [Lampson & Sturgis ??]. Details can be found in [Weihl 83].

Dynamic atomicity is *optimal*: there is no other local atomicity property that allows strictly more concurrency. We sketch a proof of this result here; a complete proof can be found in [Weihl 83]. The proof proceeds by contradiction: Suppose ***P*** is a local atomicity property that allows strictly more concurrency than dynamic atomicity. We exhibit a system composed of objects satisfying ***P*** and a non-atomic computation of that system, contradicting the claim that ***P*** is a local atomicity property.

Since ***P*** is more permissive than dynamic atomicity, there must be a specification S_x of an object x such that S_x satisfies ***P*** but S_x is not dynamic atomic. In particular, there must be at least one event sequence h_x in S_x that is not dynamic atomic; that is, such that *perm*(h_x) is not serializable in at least one total order T consistent with *precedes*(h_x). We can construct an object y whose specification contains a sequence h_y involving the activities in h_x that is serializable only in the order T. Now, consider a system containing x, y, and all the activities in h_x. Because of the order in which activities appear in h_x and h_y, there is a computation h of this system such that $h|x = h_x$ and $h|y = h_y$. Since *perm*(h_y) is only serializable in the order T, and *perm*(h_x) is not serializable in that order, *perm*(h) is not serializable. Thus, h is not atomic.

The construction of y is as follows: Let y have a single operation called increment. y is intended to behave like a counter: its state is initially zero, and each invocation of the increment operation increments the state of y and returns the resulting value. The serial sequences permitted by y have the following form:

<increment,y,a1>
<1,y,a1>
<commit,y,a1>
<increment,y,a2>
<2,y,a2>
<commit,y,a2>
...
<increment,y,an>
<n,y,an>
<commit,y,an>
...

The only serial sequences permitted by y are similar to this, but may contain more than one invocation per activity. Let the specification of y be the largest set S_y that is dynamic atomic and contains all these serial sequences. Since ***P*** is more permissive than dynamic atomicity, S_y satisfies ***P***. Let *a1, a2, ..., an* be the committed activities in h_x in the order T, and let h_y be the serial sequence permitted by y in which each activity performs one invocation, and with committed activities *a1, a2, ..., an* in that order. Note that h_y is serializable only in the order T. This gives the desired contradiction.

The locking protocols of [Bernstein 81], [Korth 81], and [Schwarz & Spector 82] are suboptimal: while sufficient to ensure atomicity (given the assumptions about the underlying recovery mechanism), they permit strictly less concurrency than does dynamic atomicity. This is due in part to fundamental limitations of the underlying models of those protocols. We will illustrate this point with detailed examples in Section 5.

4.2. Static Atomicity

Our second local atomicity property, which we call static atomicity, is a generalization of the timestamp-based multi-version protocol proposed by Reed [Reed 78]. Static atomicity differs from dynamic atomicity in that the serialization order of activities is determined *a priori*: before an activity invokes any operations, it chooses a unique timestamp. Each object then ensures that activities are serializable in timestamp order.

In the implementation of static atomicity described in [Reed 78] for objects with read and write operations, serializability in timestamp order is achieved by maintaining multiple versions of each object; associated with each version is the timestamp of the activity that wrote it. When an activity with timestamp t invokes a read operation on an object x, it selects the version of x with the largest timestamp less than t. In this section we describe in general terms the correctness property for Reed's scheme, extending Reed's ideas to include objects with user-specified operations.

4.2.1. Additional Events

To define static atomicity, we need to introduce some new events that describe the timestamps chosen by activities. An activity chooses a timestamp when it starts. Thus, in addition to invocation, termination, commit, and abort events, we include initiation events. We write the event corresponding to the initiation of activity a at object x with timestamp t as <initiate(t),x,a>. We assume that timestamps are taken from some countable, well-ordered set; in this paper we will use natural numbers.

In addition to the well-formedness constraints on event sequences stated earlier, we have the following constraints:

- An activity must initiate at an object before invoking any operations at the object.
- Initiation events for distinct activities must have distinct timestamps.
- Any two initiation events for the same activity must have the same timestamp.

For example, the following sequence is well-formed:

<initiate(1),x,a>
<member(2),x,a>
<false,x,a>
<commit,x,a>

The sequence

<initiate(1),x,a>
<member(2),y,a>
<false,y,a>
<initiate(2),y,a>
<initiate(1),y,b>
<commit,x,a>

is not, however, for three reasons. First, *a* initiates with two different timestamps. Second, *b* initiates with a timestamp used by *a*. Third, *a* invokes operations at *y* before initiating at *y*.

4.2.2. Definition of Static Atomicity

Let *h* be a well-formed sequence containing initiation, invocation, termination, commit, and abort events. We say that *h* is *static atomic* if *perm*(*h*) is serializable in timestamp order.

For example, the following sequence *h* is atomic:

<initiate(2),x,a>
<member(3),x,a>
<false,x,a>
<commit,x,a>
<initiate(1),x,b>
<insert(3),x,b>
<ok,x,b>
<commit,x,b>

However, it is not static atomic, for the following reason. *Perm*(*h*) is an acceptable serial sequence, and thus is serializable in the order *a*-*b*. However, the timestamp order in *h* is *b*-*a*, and *perm*(*h*) is not serializable in this order.

As another example, the sequence

<initiate(2),x,a>
<insert(3),x,a>
<ok,x,a>
<commit,x,a>
<initiate(1),x,b>
<member(3),x,b>
<false,x,b>
<commit,x,b>

is static atomic. *Perm*(*h*) is serializable in timestamp order (*b*-*a*).

We say that an object is *static atomic* if every sequence permitted by the object's specification is static atomic. The following theorem verifies that static atomicity is a local atomicity property:

Theorem 4: If every object in a system is static atomic, then every computation of the system is atomic.

Proof: Suppose that every object in a system is static atomic, and let *h* be a computation of the system. Let *T* be the timestamp order on the activities in *h*. By the definition of static atomicity, *perm*(*h*|*x*) is serializable in the order *T*. By Lemma 3, *perm*(*h*) is also serializable in the order *T*, so *h* is atomic.

Static atomicity, like dynamic atomicity, is optimal. The proof of optimality is similar to that for dynamic atomicity; the details may be found in [Weihl 83].

4.2.3. Comparison of Dynamic and Static Atomicity

Dynamic atomicity and static atomicity are different: each permits operations to be interleaved in ways that the other does not. This implies that optimality is a relatively weak property. In particular, optimal does not mean "best," but rather that nothing else is strictly better.

Which of these two local atomicity properties is best for a given application will depend on the patterns of operations invoked by activities. For example, dynamic atomicity works poorly for long read-only activities such as audits. If dynamic atomicity is implemented using a locking protocol, a read-only activity, once it has a lock on an object, will cause other activities that need conflicting locks to wait. Because of the need to wait for locks, long read-only activities can be quite prone to deadlock. Static atomicity, however, works reasonably well for long read-only activities. In the implementation proposed by Reed [Reed 78], read-only activities are never forced to abort (the analog of deadlock in a locking system), and are rarely delayed by other activities. On the other hand, static atomicity works poorly for updating activities unless timestamps are generated using closely synchronized clocks. For example, in the implementation proposed by Reed, if an activity attempts to write an object after another activity with a later timestamp has already read the object, the former activity must be aborted. Using dynamic atomicity, the writer might be delayed until the reader committed, but would then be able to proceed.

4.3. Hybrid Atomicity

Our final local atomicity property, which we call *hybrid atomicity*, combines features of dynamic and static atomicity, so as to avoid some of the disadvantages of each. It is similar to the multiversion scheme proposed in [DuBourdieu 82] and formally analyzed in [Bernstein & Goodman 82], but is more general in that we permit (and take advantage of) user-specified operations.

Hybrid atomicity is based on a partition of activities into two classes: read-only activities, and update activities. Intuitively, a read-only activity is one that does not invoke any operations that change the state of an object; a formal definition of this property can be found in [Weihl 83], where we describe in detail how to specify objects and activities as well as what it means for an operation to change the state of an object. All activities that invoke operations that change the states of objects are considered to be update activities.

Hybrid atomicity processes updates using dynamic atomicity. Then, as update activities commit, they choose timestamps in such a way that the updates are serializable in timestamp order. Finally, read-only activities choose timestamps before invoking any operations. When a read-only activity with timestamp *t* invokes an operation, it computes the answer to its query by including the effects of all operations executed by committed updates with timestamps less than *t*.

4.3.1. Additional Events

To define hybrid atomicity precisely, we need to use a slightly different set of events to describe timestamps. In addition, we must partition the set of activities into two subsets: the updates (written *a*, *b*, and *c*), and the read-only activities (written *r*, *s*, and *t*). Timestamps for updates are chosen when they commit; we write the event corresponding to the commitment of an update *a* at object *x* with timestamp *t* as <commit(t),x,a>. Timestamps for read-only activities are chosen when they start, so we use initiation events for them, writing the events as <initiate(t),x,r>. We use the term *timestamp events* to denote the set of all commit events for updates and all initiation events for read-only activities.

In addition to the well-formedness constraints on event sequences stated in Section 2, we have the following constraints:

- A read-only activity must initiate at an object before invoking any operations at the object.
- Any two timestamp events for distinct activities have distinct timestamps.
- Any two timestamp events for the same activity have the same timestamp.

For example, the following sequence is well-formed:

<insert(3),x,a>
<ok,x,a>
<commit(2),x,a>
<initiate(1),x,r>
<member(3),x,r>
<false,x,r>
<commit,x,r>

The following sequence *h*, however, is not:

<insert(3),x,a>
<ok,x,a>
<commit(2),x,a>
<member(3),x,b>
<true,x,b>
<commit(1),x,b>
<initiate(2),x,r>

Precedes(*h*) contains the single pair <*a*,*b*>, yet the timestamp chosen by *b* is less than that chosen by *a*. Also, *r* and *a* use the same timestamp, violating the uniqueness property of timestamps.

4.3.2. Definition of Hybrid Atomicity

Let *h* be a well-formed sequence. Define *updates*(*h*) to be the subsequence of *h* consisting of all events involving update activities in *h*, and no others. Thus, *updates*(*h*) can be obtained from *h* by throwing away all events for read-only activities. We say that *h* is *hybrid atomic* if *perm*(*h*) is serializable in timestamp order.

For example, the following sequence *h* is atomic:

<insert(3),x,a>
<ok,x,a>
<insert(4),x,b>
<ok,x,b>
<commit(1),x,a>
<commit(3),x,b>
<initiate(2),x,r>
<member(3),x,r>
<true,x,r>
<member(4),x,r>
<true,x,r>
<commit,x,r>

since it is serializable in the order *a*-*b*-*r*. However, it is not hybrid atomic, for the following reason. *Perm*(*h*) in timestamp order is the sequence

<insert(3),x,a>
<ok,x,a>
<commit(1),x,a>
<initiate(2),x,r>
<member(3),x,r>
<true,x,r>
<member(4),x,r>
<true,x,r>
<commit,x,r>
<insert(4),x,b>
<ok,x,b>
<commit(3),x,b>

which is not an acceptable serial sequence.

As another example, the sequence

<insert(3),x,a>
<ok,x,a>
<insert(4),x,b>
<ok,x,b>
<commit(1),x,a>
<commit(3),x,b>
<initiate(2),x,r>
<member(3),x,r>
<true,x,r>
<member(4),x,r>
<false,x,r>
<commit,x,r>

is hybrid atomic.

We say that an object is *hybrid atomic* if every sequence permitted by the object's specification is hybrid atomic. The following theorem verifies that hybrid atomicity is a local atomicity property:

> **Theorem 5:** If every object in a system is hybrid atomic, then every computation of the system is atomic.

The proof is identical to that for static atomicity.

Hybrid atomicity is optimal; the proof again is by contradiction.

4.3.3. Discussion

At first glance hybrid atomicity might not seem very different from static atomicity: both work by establishing a global timestamp ordering on activities and ensuring that activities are serializable in that order. Hybrid atomicity, however, chooses timestamps for updates as they commit, not before they start executing. This difference is substantial: it raises a number of interesting implementation issues, and results in some useful properties.

The basic correctness condition for hybrid atomicity is that committed activities are serializable in timestamp order. The online implementation of hybrid atomicity discussed in [Weihl ??] combines several properties to achieve this result. First, it generates timestamps for updates so that the timestamp ordering on updates is consistent with *precedes* at each object. This can be achieved easily with some simple modifications to a two-phase commit protocol [Gray 78], or by using a Lamport clock [Lamport 78] as suggested in [Bernstein & Goodman 82]; the details can be found in [Weihl ??]. Second, it processes updates using dynamic atomicity; in combination with the first property, this ensures that committed updates are serializable in timestamp order. Third, it computes the results for operations invoked by read-only activities by allowing a read-only activity with timestamp *t* to see the effects of exactly those committed updates with timestamps less than *t*.

The implementation of hybrid atomicity discussed in [Weihl ??] processes read-only activities so that they do not interfere in any way with update activities. Thus, the problems with read-only activities under dynamic atomicity are avoided. In addition, updates are processed using dynamic atomicity, avoiding the problems with updates under static atomicity.

In a sense, hybrid atomicity is really better than dynamic atomicity: it allows more interleaving of operations, although the results seen by a read-only activity under hybrid atomicity may be different than those seen by the same activity under dynamic atomicity. Actually comparing the event sequences permitted by the two properties is problematic since the event sets differ. Assuming that the cost of implementing hybrid atomicity is not too large, however, it seems likely that hybrid atomicity will perform better than dynamic atomicity. Hybrid atomicity achieves this improvement over dynamic atomicity by using more information, namely that certain activities are read-only.

For instance, consider the example of a banking system presented by Lamport [Lamport 76]. The system contains transfer activities (that move money between two accounts) and audit activities (that print out the current balances of all accounts). Lamport noted the performance problems of locking implementations, and suggested that the solution to these problems is to allow non-atomic executions. He defined a correctness property, namely that the view of the database seen by an audit must be consistent, and described an implementation that guarantees this property while permitting more concurrency than a locking implementation of atomicity. His correctness property does not ensure, however, that the view seen by an audit bears any relation to the actual state of the database. In addition, audits under his implementation still interfere with some updates. Hybrid atomicity solves the problem addressed by Lamport, namely the performance problems with read-only activities under dynamic atomicity. In contrast to Lamport's solution, hybrid atomicity ensures atomicity; this means that the view seen by an audit can be related to the updates performed by transfers and to the views seen by other audits. Furthermore, audits under the implementation of hybrid atomicity in [Weihl ??] do not interfere with any updates.

5. Related Work

Our discussion of related work is organized into two parts. First, in Section 5.1, we illustrate the limitations of a model commonly used to analyze the concurrency control problem, and discuss why our approach avoids these limitations. Then, in Section 5.2, we make some general comments on the idea of using user-specified information about the semantics of a system to achieve greater concurrency.

5.1. Limitations of the Scheduler Model

Dynamic atomicity permits more concurrency than conventional locking protocols. In this subsection we show that this difference is the result of a fundamental limitation in the model used to analyze those protocols. We begin by comparing the degree of concurrency permitted by dynamic atomicity with that permitted by conventional locking protocols, showing that dynamic atomicity permits more concurrency. Then we discuss in some detail the limitations of the model used to study the locking protocols.

The following example illustrates that dynamic atomicity allows more concurrency than the locking protocols of [Bernstein 81], [Korth 81], and [Schwarz & Spector 82]. Let *y* be a bank account object, with initial balance 0, and with operations to deposit a sum of money, to withdraw a sum of money, and to examine the current balance. Assume that an invocation of the withdraw operation can terminate in one of two ways: either normally (with result *ok*), indicating that the requested sum has been withdrawn, or abnormally (with result *insufficient_funds*), indicating that the account balance is too small to cover the request. Note that the locking protocols allow two activities to execute operations concurrently only if the operations commute. Two deposit operations commute, since addition is commutative. Two withdraw operations do not commute, however: if the current balance is large enough to cover either request but not both, then the results of the operations depend on the order in which they are executed. Similarly, a deposit operation does not commute with a withdraw operation: if the current balance is not quite large enough to cover the withdrawal, but the current balance plus the amount deposited is large enough, then the results of the operations depend on the order in which they are executed. Thus the locking protocols must prevent activities from executing two withdraw operations concurrently, and from executing withdraw operations concurrently with deposit operations.

Dynamic atomicity allows activities to execute withdraw operations concurrently as long as there is sufficient money in the account to cover all of the requests. For example, the following sequence is dynamic atomic, since it is serializable in the orders *a-b-c* and *a-c-b*:

```
<deposit(10),y,a>
<ok,y,a>
<commit,y,a>
<withdraw(4),y,b>
<withdraw(3),y,c>
<ok,y,c>
<ok,y,b>
<commit,y,c>
<commit,y,b>
```

This sequence would not be allowed, however, by any of the locking protocols.

Similarly, dynamic atomicity allows withdraw operations to be executed concurrently with deposit operations as long as the deposits are not needed to cover the withdrawals. For example, the following sequence is dynamic atomic, since it is serializable in the orders *a-b-c* and *a-c-b*:

```
<deposit(1),y,a>
<ok,y,a>
<commit,y,a>
<deposit(1),y,b>
<ok,y,b>
<withdraw(1),y,c>
<ok,y,c>
<commit,y,b>
<commit,y,c>
```

As above, this sequence would not be allowed by any of the locking protocols.

The difference between dynamic atomicity and the locking protocols appears to be a result of a fundamental limitation in the model used in [Bernstein 81], [Korth 81], and [Schwarz & Spector 82]. That model, which we will call the *scheduler model*, is pictured in Figure 5-1.

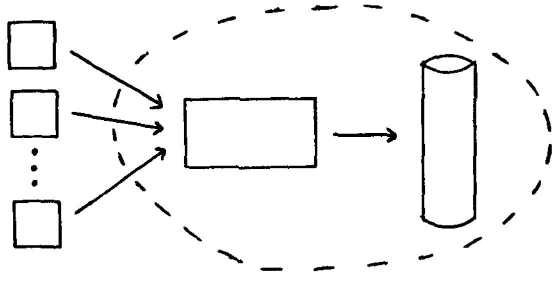

Figure 5-1: The scheduler model.

The boxes on the left represent transactions, which submit invocations to the scheduler in the middle. The scheduler determines the order in which to run operations invoked by transactions, and submits the invocations in that order to the storage module on the right, which processes the operations and returns their results to the transactions. The problem addressed in [Bernstein 81], [Korth 81], and [Schwarz & Spector 82] is to analyze the properties of the scheduler module. The problem that we address is slightly different: we analyze the properties of the interface represented by the dotted line.

The limitations of the scheduler model are illustrated by the following example. Consider a first-in-first-out queue object *x*, with operations to enqueue an integer onto the back of the queue and to dequeue an integer from the front of the queue. Note that an operation to enqueue the integer 1 does not commute with an operation to enqueue the integer 2. Now consider the following execution sequence:

```
<enqueue(1),x,a>
<ok,x,a>
<enqueue(1),x,b>
<ok,x,b>
<enqueue(2),x,a>
<ok,x,a>
<enqueue(2),x,b>
<ok,x,b>
<commit,x,a>
<commit,x,b>
<dequeue,x,c>
<1,x,c>
<dequeue,x,c>
<2,x,c>
<dequeue,x,c>
<1,x,c>
<dequeue,x,c>
<2,x,c>
<commit,x,c>
```

Note that this execution would not be permitted by the locking protocols, since the operations executed by *a* do not commute with the operations executed by *b*. It is, however, permitted by dynamic atomicity, since both equivalent serial executions of *a*, *b*, and *c* (in the orders *a-b-c* and *b-a-c*) are acceptable.

Now consider what happens in the scheduler model. We claim that the scheduler cannot schedule the invocations in the order given here. If it did, the state of the queue object after *a* and *b* commit would be 1122 (reading from front to back); thus, *c* would have to receive 1, 1, 2, and 2, not 1, 2, 1, and 2. This does not correspond to either serial execution of *a* and *b*. Thus, under the scheduler model, the example execution above is not serializable. The reason for this is that the semantics of the operations are determined by the interface between the scheduler and the storage module. The order in which operations are scheduled determines the state of the storage module, and hence the results of subsequent operations. This example illustrates that the constraints imposed by the scheduler model rule out executions that seem acceptable according to our intuitive notion of "atomicity," and that are permitted by dynamic atomicity. Our model imposes no interpretation on the order in which operations are scheduled, and thus is less restrictive than the scheduler model.

Since the scheduler model does not expose the timing of commit events relative to other events, it is impossible even to state the dynamic atomicity property in this model. This problem partially motivated our use of a different model. We are also interested in studying online implementations of atomicity. Our model incorporates both commit and abort events in part to support this study. The scheduler model, since it does not expose commit or abort events, cannot be used to study online implementations. The formal models introduced in [Bernstein & Goodman 82] and [Kanellakis & Papadimitriou 82] to study multi-version schemes have similar problems.

The scheduler model was intended to be used to study the concurrency control problem, which is but one aspect of the more general problem of ensuring atomicity. Our model was designed to be used to study atomicity in as general a setting as possible; thus, we needed to make our model as abstract as possible, and in particular could not fix our model of recovery. This means that we avoid the limitations of the scheduler model illustrated above, but also means, since our model incorporates less fixed structure, that it may be more difficult to verify implementations.

5.2. Using Semantic Information

One view of the results presented in this paper is that we have explored ways of using user-specified semantic information about the behavior of a system to achieve greater concurrency while staying within the bounds of atomicity. Atomicity is particularly useful since it allows one to reason about the partial correctness of an individual activity without regard to other activities. This makes systems easier to modify and extend. In addition, our focus on local atomicity properties means that programmers of activities need not be concerned with maintaining atomicity, and that programmers of objects can verify that atomicity is preserved without knowing what other objects are in the system; they need know only what local atomicity property is used throughout the system.

All three local atomicity properties presented in this paper make use of user-specified semantic information about the objects in a system. Hybrid atomicity also makes use of user-specified semantic information about activities. Whether an activity is read-only depends on the specifications of the objects used by the activity; this information will probably be supplied by the programmer. The read-only property is local, however: it is a property of an individual activity that can be verified given the specifications of all objects used by the activity.

Lamport's solution to the audit problem in [Lamport 76] similarly makes use of semantic information. His solution, however, makes use of global information about all activities. In addition, it is not clear how to apply his solution to new situations. Hybrid atomicity, in contrast, makes use of purely local information, and does so in a systematic fashion that makes it clear when it is applicable.

6. Conclusions

The results presented in this paper were developed as part of the Argus project at MIT. We have been exploring a methodology, based on atomic actions, for the construction of reliable, distributed programs. The methodology requires the design and implementation of objects that ensure atomicity of activities using them. The results presented in this paper and in [Weihl 83] represent a first step toward a precise understanding of the requirements on such objects.

This paper makes two primary contributions. First, we have presented a model of computation for studying atomicity that is more general than those found in the literature. Our model exposes events relevant both to serializability and to recoverability, permitting them to be studied together. In addition, our model permits the specification of non-deterministic operations, something not permitted by models previously used for studying atomicity. Second, we have presented a formal definition of atomicity. As discussed in Section 1, our treatment is novel in three important respects: it is integrated, it is data-dependent, and it supports modular design. No previous work that we know of has dealt with all three of these issues.

The local atomicity properties presented here are all optimal. The significance of these results is that no other property can allow strictly more concurrency given the same level of information about the system. Hybrid atomicity, in a sense, allows more concurrency than dynamic atomicity, but does so using additional information (namely that certain activities are read-only) and at the expense of a more complicated protocol (timestamps must be generated for all activities, and multiple versions of objects must be retained). It is important to note, however, that these properties represent only upper bounds on the concurrency that an object may permit. An object's specification need not include all sequences permissible given a local atomicity property and a specification of the object's sequential behavior. In many applications, for example, the locking protocols discussed earlier will be more than adequate as implementations of dynamic atomicity.

7. Acknowledgements

I would like to thank all the members of the Argus project for their contributions to the ideas presented here. Gene Stark, in particular, contributed much in the early stages of my work on dynamic atomicity. I would also like to thank Dan Brotsky for his careful readings of numerous drafts of this paper and for his many useful suggestions.

8. References

[Bernstein 81]
Bernstein, P., Goodman, N., and Lai, M.-Y.
Two part proof schema for database concurrency control.
In *Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 71-84.
February, 1981.

[Bernstein & Goodman 82]
Bernstein, P. A. and Goodman, N.
Multiversion concurrency control -- theory and algorithms.
Technical Report, Harvard University Aiken Computation Laboratory, June, 1982.

[Davies 73]
Davies, C. T.
Recovery Semantics for a DB/DC System.
In *Proceedings of the ACM Annual Conference*, pages 136-141. ACM, Atlanta, GA, 1973.

[Davies 78]
Davies, C. T.
Data processing spheres of control.
IBM Systems Journal 17(2):179-198, 1978.

[DuBourdieu 82]
DuBourdieu, D.J.
Implementation of distributed transactions.
In *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 81-94. 1982.

[Eswaren 76]
Eswaren, K.P, Gray, J.N, Lorie, R.A., and Traiger, I.L.
The notions of consistency and predicate locks in a database system.
Communications of the ACM 19(11):624-633, November, 1976.

[Gray 78]
Gray, J.
Notes on Database Operating Systems.
In *Lecture Notes in Computer Science,* Volume 60: *Operating Systems -- An Advanced Course*. Springer-Verlag, 1978.

[Kanellakis & Papadimitriou 82]
Kanellakis, P. and Papadimitriou, C.
On concurrency control by multiple versions.
In *Proceedings of the 1982 ACM Symposium on Theory of Computing*. 1982.

[Korth 81]
Korth, H. F.
Locking protocols: general lock classes and deadlock freedom.
PhD thesis, Princeton University, 1981.

[Lamport 76]
Lamport, L.
Towards a theory of correctness for multi-user data base systems.
Technical Report CA-7610-0712, Massachusetts Computer Associates, October, 1976.

[Lamport 78]
Lamport, L.
Time, clocks, and the ordering of events in a distributed system.
CACM 21(7):558-565, July, 1978.

[Lampson & Sturgis ??]
Lampson, B. W. and H. E. Sturgis.
Crash recovery in a distributed data storage system.
Submitted to CACM.

[Liskov & Scheifler 82]
Liskov, B., and Scheifler, R.
Guardians and actions: linguistic support for robust, distributed programs.
In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 7-19. ACM, January, 1982.
Revised version to appear in TOPLAS.

[Liskov & Zilles 74]
Liskov, B. and Zilles, S. N.
Programming with abstract data types.
In *Sigplan Notices,* Volume 9: *Proceedings of the ACM SIGPLAN Conference on Very High Level Languages*, pages 50-59. ACM, 1974.

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

[Reed 78]
Reed, D.P.
Naming and synchronization in a decentralized computer system.
PhD thesis, Massachusetts Institute of Technology, 1978.
Available as Technical Report MIT/LCS/TR-205.

[Schwarz & Spector 82]
Schwarz, P., and Spector, A.
Synchronizing shared abstract types.
Technical Report CMU-CS-82-128, Carnegie-Mellon University, September, 1982.

[Weihl 83]
Weihl, W. E.
A method for the construction of modular, reliable, concurrent systems.
PhD thesis, MIT, 1983.
Forthcoming.

[Weihl ??]
Weihl, W. E.
Long read-only actions.
Unpublished memo, 1982.

[Weihl & Liskov 83]
Weihl, W. and Liskov, B.
Specification and implementation of resilient, atomic data types.
In *Proceedings of the Symposium on Programming Language Issues in Software Systems*. ACM Sigplan, San Francisco, CA, June, 1983.