Modular Atomic Objects*

Rachid Guerraoui
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne, Switzerland

Keywords: Atomic objects, transactions, compatibility.

A worthwhile approach to achieve transaction atomicity within object-based distributed systems is to confine concurrency control and recovery mechanisms within the shared objects themselves. Such objects, called atomic objects, enhance their modularity and can increase transaction concurrency. Nevertheless, when designed independently, atomic objects can be incompatible, and if combined, do not ensure transaction atomicity anymore. It has been shown that atomic objects can be incompatible when they assume different Global Serialization Protocols (GSPs). We deal with the incompatibility problem by introducing a property of atomic object specifications, named o-atomicity, which is orthogonal to the GSP. Objects that guarantee this property achieve transaction atomicity, whatever the GSP may be. Such objects are compatible, not only with each others, but also with atomic objects that guarantee previously defined GSP-dependent properties, i.e. static atomicity, dynamic atomicity or hybrid atomicity. This is very desirable since most of existing object-based distributed systems rely on these properties. To show how o-atomicity can be ensured, we propose a generic implementation within an object-oriented framework, which we illustrate through a simple banking application.

Introduction

One of the basic claim underlying object-orientation is that of object composition. Objects are modular software components that can be designed and tested independently, and then combined within the same application. While this claim has been supported in a sequential context, much still to be done in concurrent and distributed contexts. In this paper we discuss the problem of composing objects in the context of transactional distributed systems.

Atomic Objects

To preserve the consistency of distributed systems despite concurrent accesses and failures, it is helpful to compose applications out of atomic activities called transactions [8,13,17]. Atomicity (in the sense of [28]) means that incomplete transactions appear as having no effects, and complete transactions appear as executing sequentially according to a *Serialization Order* (SO), determined by some *Global Serialization Protocol* (GSP). A worthwhile approach to achieve transaction atomicity is to confine concurrency control and recovery mechanisms within the shared objects themselves. Such objects, called *atomic objects* [28], enhance their modularity since they can be designed and tested locally, and can increase transaction concurrency by providing appropriate mechanisms to their use and semantics [2]. Nevertheless, when designed *independently*, atomic objects are sometimes *incompatible*, and when combined, do not ensure transaction atomicity anymore [29].

The Incompatibility Problem

Weihl has discussed the incompatibility problem and pointed out its deep relation with GSPs [29]. He grouped GSPs in three classes: *static* GSPs that determine SOs according to timestamps assigned to transactions at creation time, as in [26]; *dynamic* GSPs that determine SOs during transaction execution, as in [8]; and *hybrid* GSPs that determine SOs according to timestamps assigned to transactions at commit time, as in [16]. Weihl showed that objects are incompatible if they assume GSPs belonging to different classes. For example, an object which schedules transactions according to timestamps assigned to them at creation (assuming a static GSP) is incompatible with an object which

---

Received June 1, 1993; revised November 2, 1993; accepted November 2, 1993.

*A preliminary and shorter version of this paper appeared in the Proceedings of the "European Conference on Object Oriented Programming", LNCS 821, Springer Verlag, 1994; Under the title "Atomic Object Composition".

schedules transactions according to a two phase locking protocol [8] (assuming a dynamic GSP). When used together, these objects do not ensure atomicity anymore. As a consequence, applications cannot be composed out of atomic objects that assume GSPs belonging to different classes.

Since in most of existing object-based distributed systems [14,23,30,20,10], atomic objects assume particular GSPs (static, dynamic, or hybrid), their compatibility is limited. In this paper we propose a way to specify atomic objects so that they do not assume any particular GSP: they are compatible with any other atomic object.

Composing Atomic Objects Through O−atomicity

We present a property of object specifications named *orthogonal-atomicity*, or *o−atomicity*, which is *orthogonal* to the GSP. Roughly speaking, an object which guarantees *o−atomicity* assumes that there exists some GSP and schedules transactions so that they appear to execute in the SO, determined by the GSP. We show that *o−atomicity* is sufficient to achieve transaction atomicity, and that, given only the assumption of the existence of some GSP, *o−atomicity* is also necessary.

As long as atomic objects guarantee *o−atomicity*:

- They are compatible not only with each others, but also with objects that assume particular GSPs. In other words, atomic objects that guarantee *o−atomicity* can be reused in most of existing object-based distributed systems [14,23,30,20,10].

- They can be designed independently and provide their own concurrency control and recovery mechanisms. To increase transaction concurrency, objects with *high risks* conflict can for example provide semantic-based pessimistic mechanisms, whereas objects with *low risks* conflict can provide optimistic mechanisms [27,3,15,29,16,7].

- They ensure transaction atomicity whatever the GSP may be. This enables to substitute a GSP by another without affecting the objects. Such a modularity feature is very convenient since the optimality of the GSP is application-dependent [22], and it would be worthwhile to modify it according to the application evolution.

To show how *o−atomicity* can be guaranteed, we propose a generic implementation of an abstract atomic class. We show how to customize this implementation within subclasses of which objects guarantee *o−atomicity*. We present two examples of such subclasses: a subclass of which objects guarantee *o−atomicity* through a pessimistic concurrency control, and a subclass of which objects guarantee *o−atomicity* through an optimistic concurrency control. We then describe a simple banking application composed of bank-account objects from different subclasses.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes our computational model and defines object specifications and transaction atomicity. Section 4 defines *o−atomicity* and presents its characteristics through three main theorems: *sufficiency, relation with GSP-dependent properties*, and *necessity*. Section 5 presents an abstract atomic class, and describes subclasses that provide adequate mechanisms to guarantee *o−atomicity*. Section 6 summarizes our main contribution and suggests some complementary work. Appendix I contains the proofs of the theorems and Appendix II contains the details of the algorithms presented in Section 4.

Related work

The problem of incompatible atomic objects has been discussed both in the context of distributed systems and in the context of multidatabases. In the latter context, the problem consists in integrating different Local Database Systems (LDBS), based on different concurrency control and revovery mechanisms. An atomic object can be viewed as a LDBS and the problem of ensuring global atomicity of transactions accessing different atomic objects is equivalent to that of ensuring atomicity of global transactions involving different LDBSs. The result stated by Weihl *atomic objects are incompatible if they assume GSPs belonging to different classes* [29], has also been identified in multidabases [18].

The incompatibility problem can be adressed using two approaches. The first one consists in assuming *preexisting* incompatible atomic objects (or LDBSs), and proposing *global* mechanisms to combine them. This approach has mainly been considered for multidatabases. Indeed, LDBSs are considered as commercial products created independently, without considering that one day they may need to be integrated [1]. The second approach consists in identifying local compatibility properties that should be ensured by atomic objects (or LDBSs), so that they can be *directly* combined. This approach eliminates the need for any global concurrency control mechanism but makes specific assumptions on atomic objects (or LDBSs).

First approach. Georgakopoulos et al. [9], as well as Pu [24], have proposed two global certification mechanisms that collect information about local SOs, and abort transactions that lead to global cycles. The first mechanism determines local SOs by forcing local conflicts between every pair of transactions accessing the same LDBS. The second mechanism uses information exported by LDBS about their SOs. Mehrotra et al. [18] have generalized this idea by showing that if every LDBS exports its local SO, designing a global concurrency control mechanism for multidatabases comes down to designing a centralized mechanism.

Second approach. In the context of distributed systems, Weihl [28,29] has defined three local compatibility properties of atomic object specifications: *static atomicity*, *dynamic atomicity* and *hybrid atomicity*. Objects that guarantee *dynamic atomicity* assume a *dynamic* GSP, objects that guarantee *static atomicity* assume a *static* GSP, and objects that guarantee *hybrid atomicity* assume a *hybrid* GSP. Objects are compatible if and only if they guarantee the same property. In the context of multidatabases, Breibart et al. [5,6] followed the same idea by defining a property named *strong recoverability* (also named *Commitment Ordering* in [25]), and a property named *rigorous serializability*. Breibart et al. have showed that if each LDBS ensures *strong serializability* then a global atomic commitment protocol is sufficient to achieve global atomicity, and if each LDBS ensures *rigorous serializability* then global atomicity is achieved as long as every transaction is not committed until all its operations are terminated. *Strong serializability* and *rigorous serializability* are very similar to *dynamic atomicity* and both rely on *dynamic* GSPs. All these properties are thus GSP-dependent.

In this paper we follow the second approach and we define a local compatibility property of objects, named $o{-}atomicity$, which enables object composition without the need of any global concurrency control mechanism. The main characteristic of $o{-}atomicity$ is its GSP independence. Objects that guarantee this property are compatible, not only with each others, but also with objects that guarantee other GSP-dependent properties. $O{-}atomicity$ is presented as a property of object specifications. The specification model we consider is the one defined by Weihl [28]. This model is quite intuitive and has been successfully used in various works on atomic objects [28,29,22,16]. The implementation examples of $o{-}atomicity$ we propose can be viewed as a generalization of [22], where Ng proposed a way to implement atomic objects in a GSP independent manner. Whilst he has presented a particular pessimistic mechanism, we describe a generic framework which gathers both optimistic and pessimistic implementations.

(a) H =
<deposit(5), T, O1>
<ok, T, O1>
<withdrawal(5), T, O2>
<ok, T, O2>
<commit(), T, O1>
<commit(), T, O2>

(b) $H_{\{O1\}}$ =
<deposit(5), T, O1>
<ok, T, O1>
<commit(), T, O1>

(c) $H_{\{O2\}}$ =
<withdrawal(5), T, O2>
<ok, T, O2>
<commit(), T, O2>

Fig.1. Histories

System Model

This section describes our model and defines atomic object specifications and transaction atomicity. As in [28,15,29,16], our definition of atomicity is based on *serial* specifications of atomic objects, and integrates both *recoverability*, and *serializability* [4].

We consider a system compound of atomic objects (called simply objects in the following) and noted $O, O1, O2, ..Om$, and transactions noted $T, T1, T2, ..Tn$. Objects and transactions are called components. Transactions are sequential monolithic processes communicating through shared objects. Nested transactions in the sense of [21,11] are not considered here. A transaction invokes an operation on an object (or simply invokes the object), receives the matching reply, invokes another operation etc. The transaction ends either by committing or aborting. In both cases, invoked objects are notified about the transaction outcome through specific operations *commit()* and *abort()*. These operations are assumed to be provided by the objects, and invoked by the transaction and managers. When a *commit()* (resp. *abort()*) operation for a transaction T is invoked on an object O, T is said to commit (resp. abort) at O. A transaction is *terminated* when it commits or aborts at all objects it has invoked.

Histories

To depict computations, we use the simple event-based model presented in Weihl's thesis [28]. It represents computations as finite sequences of *events* called *histories* and noted $H, H1, H2, ..Hk$. Events are either an operation invocation of an object, or a reply to an invocation. Each event involves exactly one transaction and one object.

For our purpose (i.e. to define $o{-}atomicity$), we consider histories that contain only terminated transactions. We assume that a transaction is not allowed to commit at some objects and abort at others [13]; a transaction waiting for a reply (pending transaction) cannot commit; and a transaction cannot invoke any operation after it commits. We say that a history H *involves* a component C (object or transaction), if some event in H involves C. Given $\{C1, ..Ck\}$ a set of components involved in a history H, we note $H_{\{C1,..Ck\}}$ the subsequence of H containing all events involving components in $\{C1, ..Ck\}$. We say that a history is *local* to an object if it involves only this object; a history is *global* if it involves at least two objects; a history is *failure-free* if all the involved transactions commit; and a history is *sequential* if events involving different transactions are not interleaved.

Example 1 (history): Figure 1 (a) shows a history H representing a bank transfer. H involves a transaction T, and two bank-account objects $O1$ and $O2$, providing $balance()$, $deposit()$, and $withdrawal()$ operations. T invokes $deposit(5)$ on $O1$, $withdrawal(5)$ on $O2$, and then commits at both $O1$ and $O2$. Figure 1 (b) shows the history $H_{\{O1\}}$, which is local to $O1$, and figure 1 (c) shows the history $H_{\{O2\}}$, which is local to $O2$.

Events in H are:

- $< deposit(5), T, O1 >$: T invokes $deposit(5)$ on $O1$.
- $< ok, T, O1 >$: $O1$ returns a positive reply ok to T.
- $< withdrawal(5), T, O2 >$: T invokes $withdrawal(5)$ on $O2$.
- $< ok, T, O2 >$: $O2$ returns a positive reply ok to T.
- $< commit(), T, O1 >$: $O1$ is notified about the commit of T.
- $< commit(), T, O2 >$: $O2$ is notified about the commit of T.

Serial Specifications

A specification of an object is a set of local histories involving the object. The specification describes how the object interacts with the transactions that invoke it. The *serial* specification of an object O, noted $O.serial$, describes how O interacts with transactions that perform sequentially in a failure-free way (i.e. they all commit). Such a specification contains all local sequential failure-free histories involving the object, and *compatible* with its semantic (see [29] for how to derive a serial specification from an object state machine representation).

Definition 1 (serially possible history): A history H is *serially possible* if for each object O, involved by H, $H_{\{O\}} \in O.serial$.

Example 2 (serial specification): Consider a bank-account object O which returns a positive answer to a $withdrawal()$ if there is enough money in the balance, and returns a negative answer otherwise. Figure 2 (a) shows a local history $H1$ that belongs to $O.serial$, whereas figure 2 (b) shows a history $H2$ that does not. $H2$ is not compatible with the semantic of O, because O refuses $withdrawal(5)$ (i.e. returns no), although it is supposed to have enough money after $deposit(5)$.

(a) H1 =
<deposit(5), T1, O>
<ok, T1, O>
<commit(), T1, O>
<withdrawal(5), T2, O>
<ok, T2, O>
<commit, T2, O>

(b) H2 =
<deposit(5), T1, O>
<ok, T1, O>
<commit(), T1, O>
<withdrawal(5), T2, O>
<no, T2, O>
<commit, T2, O>

Fig.2. H1 is compatible with the semantic of O whereas H2 is not

Atomicity

We consider atomicity in the sense of [29,16]. A transaction is atomic if it appears to execute in an *all-or-nothing*, and *isolated* manner. This definition encompasses both *recoverability* and *serializability* [4]. An atomic history is one of which all transactions are atomic. Informally, given $committed(H)$ the set of all the committed transactions involved by a history H, H is atomic if the transactions of $committed(H)$ appear to perform in a sequential way, i.e. they can be executed in a sequential way and have the same effects. We give a precise definition of atomicity through hypothetical sequential failure-free histories. Given a history H, and a total order γ, called a Serialization Order (SO), on

(a) H1 =
<withdrawal(5), T2, O2>
<withdrawal(5), T1, O1>
<ok, T1, O1>
<ok, T2, O2>
<deposit(5), T1, O2>
<ok, T1, O2>
<deposit(5), T2, O1>
<ok, T2, O1>
<commit(), T1, O1>
<commit(), T1, O2>
<commit(), T2, O1>
<commit(), T2, O2>

(b) $perm(H1,(T2;T1))_{\{O1\}} =$
<deposit(5), T2, O1>
<ok, T2, O1>
<commit(), T2, O1>
<withdrawal(5), T1, O1>
<ok, T1, O1>
<commit(), T1, O1>

(c) $perm(H1,(T2;T1))_{\{O2\}} =$
<withdrawal(5), T2, O2>
<ok, T2, O2>
<commit(), T2, O2>
<deposit(5), T1, O2>
<ok, T1, O2>
<commit(), T1, O2>

(d) H2 =
<withdrawal(5), T2, O2>
<no, T2, O2>
<withdrawal(5), T1, O1>
<no, T1, O1>
<deposit(5), T1, O2>
<ok, T1, O2>
<deposit(5), T2, O1>
<ok, T2, O1>
<commit(), T1, O1>
<commit(), T1, O2>
<commit(), T2, O1>
<commit(), T2, O2>

(e) $perm(H2,(T2;T1))_{\{O1\}} =$
<deposit(5), T2, O1>
<ok, T2, O1>
<commit(), T2, O1>
<withdrawal(5), T1, O1>
<no, T1, O1>
<commit(), T1, O1>

(f) $perm(H2,(T1;T2))_{\{O2\}} =$
<deposit(5), T1, O2>
<ok, T1, O2>
<commit(), T1, O2>
<withdrawal(5), T2, O2>
<no, T2, O2>
<commit(), T2, O2>

Fig.3. H1 is atomic whereas H2 is not

$committed(H)$, we introduce $perm(H, \gamma)$ the sequential failure-free history made of events of H involving transactions of $committed(H)$ in the SO γ. We now define atomicity as follows:

Definition 2 (atomicity): A history H is atomic if there exists a SO γ on $committed(H)$ such that $perm(H, \gamma)$ is serially possible.

The fact that $perm(H, \gamma)$ is serially possible means that the transactions of $committed(H)$ can be executed in the SO γ, and have the same effects. H is said to be atomic in the SO γ.

Example 3 (atomicity): Figures 3 (a) and 3 (d) show two global histories $H1$ and $H2$ involving two bank-account objects $O1$ and $O2$, and two transactions $T1$ and $T2$. Intuitively, $H1$ is atomic because the transactions of $committed(H)$, $T1$ and $T2$, can be executed in the SO $(T2;T1)$ and have the same effects. More accurately, $H1$ is atomic in the SO $(T2;T1)$ because $perm(H1, (T2;T1))$ is serially possible: $perm(H1, (T2;T1))_{\{O1\}} \in O1.serial$ and $perm(H1, (T2;T1))_{\{O2\}} \in O2.serial$ (figures 3 (b) and 3 (c)). $H2$ is not atomic because neither $perm(H2, (T2;T1))$ nor $perm(H2, (T1;T2))$ is serially possible: $perm(H2, (T2;T1))_{\{O1\}} \notin O1.serial$, and $perm(H2, (T1;T2))_{\{O2\}} \notin O2.serial$ (figures 3 (e) and 3 (f)).

(a) H1 =

<deposit(3), T2,O>
<ok, T2, O>
<withdrawal(5), T1, O>
<no, T1, O>
<deposit(2), T2, O>
<ok, T2, O>
<commit(), T1, O>
<commit(), T2, O>

(b) perm(H1, (T1;T2)) =

<withdrawal(5), T1, O>
<no, T1, O>
<commit(), T1, O>
<deposit(3), T2,O>
<ok, T2, O>
<deposit(2), T2, O>
<ok, T2, O>
<commit(), T2, O>

(c) perm(H1, (T2;T1)) =

<deposit(3), T2,O>
<ok, T2, O>
<deposit(2), T2, O>
<ok, T2, O>
<commit(), T2, O>
<withdrawal(5), T1, O>
<no, T1, O>
<commit(), T1, O>

Fig.4. H1 belongs to O.behavior only if T1->T2

Behavioral Specifications

The behavioral specification of an object O, noted $O.behavior$, is a set of histories that describes how O behaves in case of concurrency and failures. It is obvious that the behavioral specification of an object contains its serial specification.

Definition 3 (possible history): A history H is *possible* if for each object O involved by H, $H_{\{O\}} \in O.behavior$.

Our goal is to draw a property of object behavioral specifications which ensures that all possible histories are atomic. Such a property must require more than the atomicity of local histories (see the example below). In fact, objects must ensure the atomicity of local histories according to some common SO.

Example 4 (possible history): Figure 3 (d) shows a global history $H2$ which is not atomic although local histories are so. The local history $H2_{\{O1\}}$ is atomic in the order $(T1;T2)$ but not in the order $(T2;T1)$ $(perm(H2, (T2;T1))_{\{O1\}} \notin O1.serial$ (note that $perm(H_{\{O\}}, \gamma) = perm(H, \gamma)_{\{O\}}$), whereas $H2_{\{O2\}}$ is atomic in the order $(T2;T1)$ but not in the order $(T1;T2)$ $(perm(H2, (T1;T2))_{\{O2\}} \notin O2.serial)$. Therefore, to ensure atomicity, $O1$ and $O2$ must avoid such a history: $H2$ must be impossible. On the contrary, figure 3 (a) shows a history $H1$ which is atomic because local histories, $H2_{\{O1\}}$ and $H2_{\{O2\}}$, are atomic according to the common SO $(T2;T1)$.

The global protocol in charge of establishing a common SO is noted GSP (Global Serialization Protocol). To ensure atomicity, objects must follow the same GSP. Weihl has distinguished (in [29]) three classes of GSPs: *static* GSPs that determine SOs according to timestamps assigned to transactions at creation; *dynamic* GSPs that determine SOs during transaction execution; and *hybrid* GSPs that determine SOs according to timestamps assigned to transactions at commit time. Given this classification, he defined three properties of object behavioral specifications: *static atomicity, dynamic atomicity* and *hybrid atomicity.* Objects that guarantee *static atomicity* (resp. *dynamic atomicity* and *hybrid atomicity*) assume a static GSP (resp. a dynamic GSP and a hybrid GSP). Objects are incompatible if they guarantee different GSP-dependent properties.

In the following, we introduce a property of object behavioral specifications, named $o\text{-}atomicity$ (*orthogonal atomicity*), which enables to ensure atomicity, whatever the GSP may be.

The $O-atomicity$ Property

In this section we define the $o-atomicity$ property and we present its characteristics through three theorems.

Definition of $O-atomicity$

We assume that there exists some GSP, whatever it may be, which determines for each history H in the system, a SO noted $\tau[H]$, on all the transactions involved by H. We note $\tau[H]_{committed(H)}$ the restriction of $\tau[H]$ to $committed(H)$.

Definition 4 ($o-atomicity$): **The behavioral specification $O.behavior$ of an object O satisfies the $o-atomicity$ property, if each local history H in $O.behavior$ is atomic in the SO $\tau[H]_{committed(H)}$.**

Example 5 ($o-atomicity$): Consider a bank-account object O such that $O.behavior$ satisfies $o-atomicity$. Assume $H1$ is a local history involving O, and two transactions $T1$ and $T2$ (figure 4 (a)), such that the GSP establishes that $T1 \longrightarrow_{\tau[H1]} T2$ ($T1$ must be serialized before $T2$). $H1$ belongs to $O.behavior$ since it is atomic in the order $(T1;T2)$: $perm(H1,(T1;T2))$ is serially possible (figure 4 (b)). $H1$ would not belong to $O.behavior$ if the GSP established that $T2 \longrightarrow_{\tau[H1]} T1$, because $H1$ is not atomic in the order $(T2;T1)$: $perm(H1,(T2;T1))$ is not serially possible (figure 4 (c)).

The $o-atomicity$ property can be viewed as a correctness criteria of the set of histories that could be generated by an object. In this sense, it constitutes a characterization of the concurrency control and failure recovery mechanisms used by the object.

Characteristics of $O-atomicity$

The following theorem implies that $o-atomicity$ is a sufficient condition to achieve history atomicity.

Theorem 1 (**sufficient condition**): **if each object behavioral specification satisfies $o-atomicity$, then every possible history is atomic.**
Proof outline: we show that if H is a possible history, then it is atomic in the SO $\tau[H]_{committed(H)}$ (see Appendix I).

Since $o-atomicity$ does not rely on any particular GSP, this theorem means that objects which guarantee $o-atomicity$ ensure history atomicity whatever the GSP may be. Such objects can thus be designed and tested within applications relying on different GSPs, and then reused within the same application. The most efficient GSP can be chosen according to the application semantic [22]. The GSP can be changed for another according to the evolution of the application, without affecting the objects.

The following theorem points out the compatibility relation between $o-atomicity$ and GSP-dependent properties.

Theorem 2 (**relation with GSP-dependent properties**): **if each object behavioral specification satisfies either $o-atomicity$ or *static atomicity* (resp. $o-atomicity$ or *dynamic atomicity*, $o-atomicity$ or *hybrid atomicity*), then every possible history is atomic.**
Proof outline: we show that every possible history H is atomic in the SO $\tau[H]_{committed(H)}$ determined by the static (resp. dynamic or hybrid) GSP (see Appendix I).

This theorem implies that atomic objects which guarantee $o-atomicity$ are compatible, not only with each others, but also with objects which guarantee GSP-dependent properties, i.e. *static atomicity*, *dynamic atomicity*, or *hybrid atomicity*. Note that these objects must all guarantee the same GSP-dependent property because the GSP-dependent properties are incompatible with each others. The theorem is quite intuitive since *static atomicity* (resp. *dynamic atomicity*, or *hybrid atomicity*) implies history atomicity assuming a static (resp. dynamic, or hybrid) GSP, whereas

$o-atomicity$ implies history atomicity, whatever the GSP may be.

The following theorem implies that, given only the assumption of the existence of some GSP, $o-atomicity$ is also a necessary condition to achieve history atomicity.

Theorem 3 (necessary condition): If $o-atomicity$ is not satisfied by an object behavioral specification, then there exists a possible history that is not atomic.
Proof outline: we suppose that there exists an object of which the behavioral specification does not satisfy $o-atomicity$, and we show that we can build a global possible history that is not atomic (see Appendix I).

This theorem means that there is no property of object specifications weaker (in the sense that it allows more histories) than $o-atomicity$, which, based only on the assumption of the existence of a GSP, suffices to ensure the atomicity of global histories.

An Abstract Atomic Class

In this section we propose examples of how to implement objects that guarantee $o-atomicity$. We consider an object-oriented framework: objects are instances of classes and classes are organized in an inheritance hierarchy. An object O, instance of a class A, is said to be derived from a class A' if A is a subclass of A'. We describe an abstract class, named *AtomicObject*, which provides the basic state variables and operations of a generic implementation. From this class, we derive a subclass of which objects guarantee $o-atomicity$ through a pessimistic concurrency control, and a subclass of which objects guarantee $o-atomicity$ through an optimistic concurrency control. To illustrate these implementations, we describe a simple banking application composed of objects from the two subclasses. To point out the GSP orthogonality of $o-atomicity$, we first assume a static GSP, then a dynamic GSP.

Basic State Variables and Operations

The state variables provided by the basic abstract class *AtomicObject* are: a log of invocations and a set of transaction identifier lists. For an object O, derived from *AtomicObject*, the log noted Log_O, contains accepted invocations, i.e., invocations for which O has performed the invoked operations. We note $T(inv)$ the invoking transaction. The status of an invocation inv in Log_O, noted $status(inv)$, is either *active* if $T(inv)$ is not terminated yet, or *committed* if $T(inv)$ is committed. The set of transaction identifier lists is used to maintain dependencies between transactions. For each transaction T of which an invocation has been accepted by O, a *dependency list*, noted $D_O(T)$, is created.

The main operations provided by the class *AtomicObject* are *abort()*, *ask-commit()*, and *commit()*, invoked by the transaction managers, and *access()* and *order()*, invoked internally by the objects. These operations are defined for an object O derived from *AtomicObject*, as follows:

- $order(T_i, T_j)$: This operation implements the GSP. It is internally invoked by O to know the SO between any pair of different transactions (Ti, Tj): $order(Ti, Tj) = true$ means that Ti is serialized before Tj; $order(Ti, Tj) = false$ means that Tj is serialized before Ti; $order(Ti, Tj) = unknown$ means that the SO is yet unknown. Since every subclass of *AtomicObject* inherits the operation *order()*, all objects derived from *AtomicObject* follow the same GSP. Changing the GSP comes down to change the implementation of the operation *order()* within the class *AtomicObject* (see examples below (4.5)).

- *abort(T)*: This operation is invoked on O when a transaction T, that invoked O, is aborted. O discards the effects of T by performing appropriate *reverse operations*, and deletes its invocations and its dependency list. We assume here that each operation has a corresponding reverse operation, e.g. the reverse operation of *deposit(x)* is *withdrawal(x)*.

- *ask-commit(T)*: This operation is invoked on O when a transaction T, that invoked O, is about to commit. O returns *ok* if every transaction in $D_O(T)$ is committed and serialized before T; O returns *no* if there exists a transaction in $D_O(T)$ which is either aborted or serialized after T; otherwise O delays the operation $ask-commit()$ (until the notification of another transaction termination), since some transactions in $D_O(T)$ are not yet terminated, or their SO with T is still unknown.

- *commit(T)*: This operation is invoked on O when a transaction T, that invoked O, is committed. O changes the status of T invocations in its log (from *active* to *commit*).

- *access(inv)*: This operation is the core of the concurrency control algorithm. It is internally invoked by O every time a transaction $T(inv)$ invokes O. The *access()* operation is based on three predicates: $aCond(), wCond()$, and $cCond()$. $aCond()$ is evaluated for each invocation inv, and represents the condition under which $T(inv)$ must be aborted; $wCond()$ is also evaluated for each invocation inv, but represents the condition under which $T(inv)$ must be delayed (until a termination notification); $cCond()$ is evaluated for every pair of invocations

(inv, inv') (where inv' is an invocation in Log_O), and represents the condition under which $T(inv')$ must be added to $D_O(T(inv))$. Customizing the generic concurrency control implementation comes down to define the three predicates within subclasses of *AtomicObject*.

Given the three predicates $aCond()$, $wCond()$, and $cCond()$, the *access()* algorithm is the following:

$access(inv)$
    {
    $await := true$;
    while $(await)$
        {
        if $(aCond(inv))$ **then** $abort(T(inv))$;
        if $(wCond(inv))$ **then** $wait(T(inv))$;
        else $await := false$;
        }
    $\forall inv' \in Log_O$
        if $(cCond(inv, inv') = true)$ **then**
            $add(T(inv'), D_O(T(inv)))$;
    }

Atomic Subclass with Pessimistic Concurrency Control

The class *PessAtomicObject* is built by inheriting the state variables and operations from the basic class *AtomicObject*, and by defining the predicates $aCond()$, $wCond()$ and $cCond()$ as follows:

- $aCond(inv) = (\exists inv' \in Log_{O1}$ **such that** $((T(inv) \neq T(inv')) \wedge (order(T(inv), T(inv')) = true))$;

- $wCond(inv) = (\exists inv' \in Log_{O1}$ **such that** $((T(inv) \neq T(inv'))$ $\wedge((order(T(inv), T(inv')) = unknown)$ $\vee\,(status(inv') = active))))$;

- $cCond(inv, inv') = false$.

(a) H1 =	(b) H2 =
<withdrawal(5), T1, O1>	<withdrawal(5), T1, O1>
<ok, T1, O1>	<ok, T1, O1>
<balance(), T2, O2>	<deposit(5), T1, O2>
<5, T2, O2>	<ok, T1, O2>
<balance(), T2, O1>	<balance(), T2, O2>
<0, T2, O1>	<10, T2, O2>
<deposit(5), T1, O2>	<balance(), T2, O1>
<ok, T1, O2>	<0, T2, O1>
<commit(), T1, O1>	<commit(), T1, O1>
<commit(), T1, O2>	<commit(), T1, O2>
<commit(), T2, O1>	<commit(), T2, O1>
<commit(), T2, O2>	<commit(), T2, O2>

Fig.5. Transfer and consultation histories

Broadly speaking, an object $O1$ instance of *PessAtomicObject*, aborts a transaction $T(inv)$ if $T(inv)$ cannot be performed in the SO ($T(inv)$ arrives too late). $T(inv)$ is delayed if the SO is yet unknown, or if $T(inv)$ can interfere with another transaction which invoked $O1$; otherwise inv is accepted. The resulting algorithm is described in Appendix II.

Theorem 4 ($o-atomicity$): **every object of the class** *PessAtomicObject* **guarantees the** $o-atomicity$ **property.**
Proof outline: we show that the object forces, *a priori*, a sequential execution of the invoking transactions in the SO given by the operation *order()* (see Appendix II).

Atomic Subclass with Optimistic Concurrency Control

The class *OptAtomicObject* is built by inheriting the state variables and operations from the basic class *AtomicObject*, and by defining the predicates $aCond()$, $wCond()$ and $cCond()$ as follows:

- $aCond(inv) = (\exists inv' \in Log_{O2}$ **such that** $((T(inv) \neq T(inv')) \wedge (order(T(inv), T(inv')) = true)))$;

- $wCond(inv) = false$;

- $cCond(inv, inv') = (((T(inv) \neq T(inv'))$ $\wedge((status(inv) = active)$ $\vee\,(order(T(inv), T(inv')) = unknown)))$

An object $O2$ instance of *OptAtomicObject*, accepts every invocation inv unless $T(inv)$ arrives too late. Therefore, inv can be accepted although an invocation inv' of another transaction is still active, or the SO between $T(inv)$ and $T(inv')$ is yet unknown. Nevertheless, at the invocation of $ask-commit(T(inv))$, $O2$ requires $T(inv')$ to commit and to be serialized before $T(inv)$; otherwise $T(inv)$ is aborted. The resulting algorithm is described in Appendix II.

Theorem 5 ($o-atomicity$): **every object of the class** *OptAtomicObject* **guarantees the** $o-atomicity$ **property.**
Proof outline: we show that the object ensures, *a posteriori*, a sequential execution of the invoking transactions in the SO given by the operation *order()* (see Appendix II).

A Simple Banking Application

To illustrate the proposed implementations, we describe a simple banking application where a transaction $T1$ executes a bank transfer from a bank-account object $O1$ to a bank-account object $O2$, while another transaction $T2$ executes a consultation of $O1$ and $O2$. $O1$ is derived from PessAtomicObject and $O2$ is derived from OptAtomicObject[1]. Figure 5 (a) depicts a non-atomic history $H1$, whereas figure 5 (b) depicts an atomic one $H2$. In the following we show how $O1$ and $O2$ guarantee that only $H2$ is possible. To illustrate the GSP orthogonality, we consider first a static GSP, and second a dynamic GSP.

Static GSP: Suppose the SO is known statically upon transaction creation, i.e. the operation *order()* within the abstract class *AtomicObject* implements a static GSP. Each transaction holds a timestamp which is assigned to it at the time of creation. The timestamps reflect the SO. Hence, when a transaction T invokes an operation on an object, the latter knows the SO between T and any transaction of which an invocation has already been accepted by O. $H1$ is not possible: (1) if the SO is $(T2;T1)$, $O1$ will accept $< withdrawal(5), T1, O1 >$ and will abort $T2$ when receiving $< balance(), T2, O1 >$, since $T2$ will arrive too late; (2) if the SO is $(T1;T2)$, $O2$ will accept $< balance(), T2, O2 >$, and then will abort $T1$ when receiving $< deposit(5), T1, O2 >$, since $T1$ will arrive too late. $H2$ is possible if the SO is $(T1;T2)$ (sequential execution). If the SO is $(T2;T1)$, $O2$ will accept $< deposit(5), T1, O2 >$ then will abort $T2$ as $< balance(), T2, O2 >$ will arrive too late.

Dynamic GSP: Suppose now the SO is determined dynamically according to how transactions invoke operations on objects, i.e., the operation *order()* within the abstract class *AtomicObject* implements a dynamic GSP. An object knows the SO between two transactions Ti and Tj if either they are both committed or Ti has committed and Tj invokes the object later. $H1$ is not possible: $O1$ will delay $< balance(), T2, O1 >$ waiting for $T1$ to commit whereas $O2$ will require $T2$ to commit before allowing $T1$ to do so; this situation will lead to a deadlock and one of the transactions will be aborted (we suppose here that deadlocks are detected and resolved by aborting transactions). $H2$ is possible: $O1$ will accept $< withdrawal(5), T1, O1 >$, will delay $< balance(), T2, O1 >$ (and block $T2$), and then will resume $T2$ at the commit time of $T1$.

Concluding Remarks

This paper delas with the problem of composing atomic objects in the context of transactional distributed systems. We have discussed how to specify atomic objects so that they can be combined with any other atomic object, and ensure transaction atomicity. Atomic objects can be incompatible when they assume different GSPs. When combined, such objects do not ensure transaction atomicity anymore. Starting from this observation, we have proposed a property of object specifications, named $o-atomicity$, which ensures transaction atomicity whatever the GSP may be. Broadly speaking, an object behavioral specification satisfies the $o-atomicity$ property, if each of its local histories is atomic in the order determined by the GSP. $O-atomicity$ represents exactly what must be ensured by an atomic object, which assumes only that there exists some GSP (without assuming any particular one).

$O-atomicity$ enhances atomic object compatibility. It enables atomic objects to be designed and tested within various applications relying on different GSPs, and then reused within the same application. Within an application, a GSP can be changed for a more efficient one according to the application semantic, without affecting atomic objects. Furthermore, atomic objects that guarantee $o-atomicity$ are compatible, not only with each others, but also with objects that guarantee GSP-dependent properties. This is very desirable since most of existing transactional systems rely on these properties [14,23,30,20,10]. Our work can be viewed as a generalization of [22], where Ng proposed a way to design atomic objects without assuming any particular GSP. Whilst he proposed a particular pessimistic mechanism, we have considered a more general specification framework that enables pessimistic and optimistic implementations.

We believe that designing efficient concurrency control and recovery algorithms that ensure $o-atomicity$, is an important issue to be addressed before $o-atomicity$ can be practically applied to object-based distributed systems. The aim of the algorithms proposed in the paper was rather to show the feasibility of our approach. An interesting and complementary question, is how to modify existing algorithms that ensure GSP-dependent properties, so that they ensure $o-atomicity$. Such a facility would be of great help in multidatabases and *federative* distributed systems, where a major problem is how to safely combine *preexisting* components.

Acknowledgments

I am very grateful to Jean Ferrié, Benoît Garbinato, Oscar Nierstrasz, André Schiper, Bill Weihl, Gerhard

Weikum, and Jeannette Wing for their helpful comments.

Appendix I

Theorem 1 (sufficient condition): if each object behavioral specification satisfies $o{-}atomicity$, **then every possible history is atomic.**

Proof: Consider a possible history H **(i)**. We note $\tau[H]$ the SO (established by the GSP) of the transactions involved by H, and we note $\tau[H]_{committed(H)}$ the restriction of $\tau[H]$ to $committed(H)$. Assume that, $\forall O$ involved by H, $O.behavior$ satisfies $o{-}atomicity$ **(ii)**.

1: By definition 2, assumption **(i)**$\Rightarrow \forall O$ involved by H, $H_{\{O\}} \in O.behavior$ **(1)**.

2: By definition 4, assumption **(ii)** $\Rightarrow \forall O$ involved by H, $H_{\{O\}}$ is atomic in the order $\tau[H_{\{O\}}]_{committed(H_{\{O\}})}$ **(2)**.

3: By definition 3, **(2)** $\Rightarrow \forall O$ involved by H, $perm(H_{\{O\}}, \tau[H_{\{O\}}]_{committed(H_{\{O\}})}) \in O.serial$ **(3)**.

4: Since $perm(H_{\{O\}}, \tau[H_{\{O\}}]_{committed(H_{\{O\}})}) = perm(H, \tau[H]_{committed(H)})_{\{O\}}$, then by definition 1, **(3)** $\Rightarrow perm(H, \tau[H]_{committed(H)})$ is serially possible **(4)**.

5: By definition 3, **(4)** $\Rightarrow H$ is atomic (in the order $\tau[H]_{committed(H)}$). $\square$

Theorem 2 (relation with GSP-dependent properties): if each object behavioral specification satisfies either $o{-}atomicity$ **or** *static atomicity* **(resp.** *dynamic atomicity*, **or** *hybrid atomicity*)**, then every possible history is atomic.**

Proof: Consider a possible history H **(i)**, and assume that A is the set of objects involved by H of which behavioral specification satisfy $o{-}atomicity$ **(ii)**, and A' is the set of objects involved by H of which behavioral specification satisfy *static atomicity* **(iii)** (resp. *dynamic atomicity*, or *hybrid atomicity*).

1: By definition 2, assumption **(i)** $\Rightarrow \forall O \in (A \cup A')$, $H_{\{O\}} \in O.behavior$ **(1)**.

2: By reformulating the definition of *static atomicity* (resp. *dynamic atomicity*, or *hybrid atomicity*), given in [29], assumption **(ii)** and **(1)** $\Rightarrow \exists$ $\tau 1[H_A]$, a SO determined by a static (resp. dynamic or hybrid) GSP of the transactions involved by H_A, such that $\forall O_j \in A$, $H_{\{O_j\}}$ is atomic in the order $\tau 1[H_{\{O_j\}})]_{committed(H_{\{O_j\}})}$ **(2)**.

3: By definition 3, **(2)** $\Rightarrow \forall O_j \in A$, $perm(H_{\{O_j\}}, \tau 1[H_{\{O_j\}}]_{committed(H_{\{O_j\}})}) \in O_j.serial$ **(3)**.

Using the same static (resp. dynamic or hybrid) GSP, we can extend the SO $\tau 1[H_A]$ of the transactions invoking objects in A, so that to obtain a SO $\tau[H]$ of the transactions invoking also objects in A'.

4: By definition 4, assumption **(iii)** and **(1)** $\Rightarrow \forall O_i \in A'$, $H_{\{O_i\}}$ is atomic in the order $\tau[H_{\{O_i\}}]_{committed(H_{\{O_i\}})}$ **(4)**.

5: By definition 3, **(4)** $\Rightarrow \forall O_i \in A'$, $perm(H_{\{O_i\}}, \tau[H_{\{O_i\}}]_{committed(H_{\{O_i\}})}) \in O_i.serial$ **(5)**.

6: **(3)** and **(5)** $\Rightarrow \forall O \in (A \cup A')$, $perm(H_{\{O\}}, \tau[H_{\{O\}}]_{committed(H_{\{O\}})}) \in O.serial$ **(6)**.

7: By definition 1, **(6)** $\Rightarrow perm(H, \tau[H]_{committed(H)})$ is serially possible **(7)**.

8: By definition 3, **(7)** $\Rightarrow H$ is atomic (in the order $\tau[H]_{committed(H)}$). $\square$

Theorem 3 (necessary condition): If $o{-}atomicity$ **is not satisfied by an object behavioral specification, then there exists a possible history that is not atomic.**

Proof: Suppose that there exists an object O, such that $O.behavior$ does not satisfy $o{-}atomicity$ **(i)**.

1: By definition 4, assumption **(i)** $\Rightarrow \exists H1 \in O.behavior$, such that $H1$ is not atomic in the order $\tau[H1]$ **(1)**; where $\tau[H1]$ is the SO determined by the GSP. We note $T1, T2..Tn$ the series of the transactions of $committed(H1)$, in the order $\tau[H1_{committed(H1)}]$).

2: By definition 3, **(1)** $\Rightarrow perm(H1, \tau[H1]_{committed(H1)}) \notin O.serial$ **(2)**.

Now assume another object Oc representing a counter, such that $Oc.behavior$ satisfies $o{-}atomicity$ **(ii)**. Oc provides an incrementation operation that returns its current value. We can build a local history $H2$, involving the transactions of $committed(H1)$, such that $\tau[H2] = \tau[H1_{committed(H1)}]$ ($H2$ is the sequential failure-free history involving the transactions of $committed(H1)$, in the order $\tau[H2]$; $H2 = perm(H2, \tau[H2])$ (figure 6). An important characteristic of $H2$ and the counter semantic is that the only order γ, such that $perm(H2, \gamma) \in Oc.serial$ is $\tau[H2]$ **(iii)**.

3: By definition 3, assumption **(iii)** $\Rightarrow$ the only order γ in which $H2$ is atomic, is the order $\tau[H2]$ **(3)**.

4: By definition 4, assumption **(ii)** and **(3)** $\Rightarrow H2$ belongs to $Oc.behavior$ **(4)**.

5: By definition 2, **(1)** and **(4)** $\Rightarrow H1 \cup H2$ is a possible history **(5)**.

6: By definition 3, **(2)** and assumption **(iii)** $\Rightarrow (H1 \cup H2)$ is not atomic **(6)** (because the only order γ, such that $perm((H1 \cup H2), \gamma)_{Oc} \in Oc.serial$ is $\tau[H2]$, whereas $perm(H1, \tau[H2]) \notin Oc.serial$).

7: $(H1 \cup H2)$ is thus possible and not atomic. $\square$

```
H2 =
<increment(), T1, Oc>
<1, T1, Oc>
<increment(), T2, Oc>
<2, T2, Oc>
...
<increment(), Tn, Oc>
<n, Tn, Oc>
<commit(), T1, Oc>
<commit(), T2, Oc>
...
<commit(), Tn, Oc>
```

Fig.6. A counter history

Appendix: II

Theorem 4 ($o-atomicity$): **every object of the class** *PessAtomicObject* **guarantees the** $o-atomicity$ **property.**

An object $O1$, instance of the class *PessAtomicObject*, has the following *access()* algorithm:

$access(inv)$
{
$await := true$;
while $(await)$
{
if $(\exists inv' \in Log_{O1}$ **such that**
$((T(inv) \neq T(inv'))\wedge$
$(order(T(inv), T(inv')) = true)))$
then $abort(T(inv))$;
if $(\exists inv' \in Log_{O1}$ **such that**
$((T(inv) \neq T(inv'))\wedge$
$((order(T(inv), T(inv')) = unknown)$
$\vee(status(inv') = active))))$
then $wait(T(inv))$;
else $await := false$;
}
}

Proof of theorem 4: Assume $O1$ is an instance of *PessAtomicObject*.
1: When $O1$ receives an invocation inv from a transaction $T(inv)$, $O1$ checks whether there exists an invocation inv' in Log_{O1}, of a different transaction $T(inv')$, such that inv' is still active, i.e. *status(inv') = active*. In this case $T(inv)$ is delayed until a termination invocation (*commit* or *abort*) arrives at O. Therefore, $O1$ delays interfering transactions and forces a strict sequential execution **(1)**.
2: if $T(inv)$ invokes $O1$ after a transaction $T(inv')$ serialized before it ($T(inv)$ comes too late), T(inv) is either delayed if inv' is still active, or aborted if inv' is already committed. As a result, $O1$ schedules transactions according to the SO given by the operation $order()$ **(2)**.
3: (1) and **(2)** $\Rightarrow \forall H_{O1}\in O1.behavior$, $H_{\{O1\}}$ corresponds to a serial execution of transactions in the SO $\tau[H_{\{O1\}}]$ **(3)**.
4: (3) $\Rightarrow \forall H_{\{O1\}} \in O1.behavior$, $H_{\{O1\}}$ is atomic in the SO $\tau[H_{\{O1\}}]$. $\Box$

Theorem 5 ($o-atomicity$): **every object of the class** *OptAtomicObject* **guarantees the** $o-atomicity$ **property.**

An object $O2$, instance of the class *OptAtomicObject*, has the following *access()* algorithm:

$access(inv)$
{
if $(\exists inv' \in Log_{O2}$ **such that**
$((T(inv) \neq T(inv'))$
$\wedge(order(T(inv), T(inv')) = true)))$
then $abort(T(inv))$;
$\forall inv' \in Log_{O2}$
if $((T(inv) \neq T(inv')) \wedge ((status(inv) = active)$
$\vee(order(T(inv), T(inv')) = unknown)))$
then $add(T(inv'), D_{O2}(T(inv))$;
}

Proof of theorem 5: Assume $O2$ is an instance of *OptAtomicObject*.
1: $O2$ accepts every invocation inv, unless it has already accepted an invocation inv', of a transaction $T(inv')$, that is serialized before $T(inv)$; in which case $T(inv)$ is aborted (it arrived too late).
2: When $O2$ accepts an invocation inv, after it has accepted an invocation inv' which is still active, $O2$ requires that, at commit time, $T(inv')$ should have committed and serialized before $T(inv)$. If $T(inv')$ comes to invoke $O2$ later, $T(inv')$ will be attached a contradictory termination condition which might not be satisfied at commit time. In this case, both $T(inv)$ and $T(inv')$ will be aborted. Therefore, out of order and interfering transactions will be aborted (*a posteriori)*.
3: (1) and **(2)** $\Rightarrow \forall H_{\{O2\}}\in O2.behavior$, $H_{\{O2\}}$ corresponds to a serial execution of transactions in the SO $\tau[H_{\{O2\}}]$ **(3)**.
4: (3) $\Rightarrow \forall H_{\{O2\}} \in O2.behavior$, $H_{\{O2\}}$ is atomic in the SO $\tau[H_{\{O2\}}]$. $\Box$

Notes

1. In [12], we give more details on bank-account subclasses of PessAtomicObject and OptAtomicObject. We describe concrete implementations of these subclasses within an object-based distributed system. In both subclasses, we increase concurrency by taking into account the bank-account semantics. More precisely, we refine the predicates $aCond()$, $wCond()$ and $cCond()$ by considering the commutativity of two *deposit()* executions, and the commutativity of two *balance()* executions.

References

R. Alonso, H. Garcia Molina and K. Salem - Concurrency control and recovery for global procedures in federated database systems - Data Engineering - Vol 10, Num 3 - 1987.

M.S. Atkins and M.Y. Coady - Adaptable Concurrency Control for Atomic Data Types - ACM Transactions on Computer Systems - Vol 10, Num 3 - 1992.

B.R. Badrinath and K. Ramamritham - Semantic-

Based Concurrency Control: Beyond Commutativity - ACM Transactions on Database Systems - Vol 17, Num 1 - 1987.

A.J. Bernstein, V. Hadzilacos and N. Goodman - Concurrency Control and Recovery in Distributed Database Systems - Addison Wesley - 1987.

Y. Breibart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz - On rigorous transactions scheduling - IEEE Transactions on Software Engineering - Vol 17, Num 9, pp 954.960 - 1991.

Y. Breibart, H. Garcia Molina and A. Silberschatz - Overview of Multidatabase Transaction Management - The VLDB Journal - Vol 1, Num 2 - 1992.

P.K. Chrysanthis, S. Raghuram and K. Ramamritham - Extracting Concurrency From Objects: A Methodology - ACM Proceedings of the SIGMOD International Conference on Management of Data - pp 108.117 - 1991.

K.P. Eswaran, J.N. Gray, R.A.Lorie and I.L. Traiger - The notion of consistency and predicate locks in a database system - Communications of the ACM - Vol 19, Num 11 - 1976.

D. Georgakopoulos, M. Rusinkiewicz and A. Sheth - On Serializability of Multidatabase Transactions Through Forced Local Conflicts - IEEE Proceedings of the International Conference on Data Engineering - pp 314.323 - 1991.

R. Guerraoui, R. Capobianchi, A. Lanusse and P. Roux - Nesting Actions through Asynchronous Message Passing: the ACS protocol - Proceedings of the European Conference on Object Oriented Programming - LNCS (615), Springer Verlag, pp 170.184 - 1992.

R. Guerraoui - Nested Transactions: Reviewing The Coherency Contract - Proceedings of the International Symposium on Computer and Information Science - pp 152.160 - 1993.

R. Guerraoui - Towards Modular Concurrency Control for Distributed Object Oriented Systems - IEEE Proceedings of the International Workshop on Future Trends in Distributed Computing Systems - pp 240.247 - 1993.

J.N. Gray - Notes on database operating systems - In Operatings Systems: An Advanced Course - LNCS, Springer Verlag, pp 393.481 - 1978.

M.P. Herlihy and J. Wing - Avalon: language support for reliable distributed systems - IEEE Proceedings of the International Symposium on Fault-Tolerant Computing - pp 89.95 - 1987.

M.P. Herlihy and W.E. Weihl - Hybrid Concurrency Control for Abstract Data Types - ACM Proceedings of the Symposium on Principles of Database Systems - pp 201.220 - 1988.

M.P. Herlihy - Apologizing Versus Asking Permissions: Optimistic Concurrency Control for Abstract Data Types - ACM Transactions on Database Systems - Vol 15, Num 1 - 1990.

N. Lynch, M. Merrit, W. Weihl and A. Fekete - Atomic Transactions - Morgan-Kaufman - 1994.

S. Mehrotra, R. Rastogi, H.F. Korth and A. Silberschatz - The Concurrency Control Problem In Multidatabases: Characteristics And Solutions - Technical Report 91-37, Department of Computer Science, Univ of Texas at Austin - 1991.

S. Mehrotra, R. Rastogi, Y. Breibart, H.F. Korth and A. Silberschatz - A Transaction Model for Multidatabase Systems - IEEE Proceedings of the International Conference on Distributed Computing Systems - pp 56.63 - 1992.

M. Mock, R. Kroeger and V. Cahill - Implementing Atomic Objects with the Relax Transaction Facility - Computing Surveys - Vol 5, Num 3 - 1992.

J.E.B. Moss - Nested Transactions: An Approach to Reliable Distributed Computing - MIT Press, MA - 1985.

T.P. Ng - Using Histories to Implement Atomic Objects - ACM Transactions on Computer Systems - Vol 7, Num 4 - 1989.

G.D. Parrington and S.K. Shrivastava - Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems - Proceedings of the European Conference on Object Oriented Programming - LNCS, Springer Verlag - pp 233.247 - 1988.

C. Pu - Superdatabases for Composition of Heterogeneous Databases - IEEE Proceedings of the International Conference on Data Engineering - pp 548.555 - 1988.

Y. Raz - The Principle of Commitment Ordering - Proceedings of the International Conference on Very Large Data Bases - pp 292.312 - 1992.

D.P. Reed - Naming and Synchronization in a Decentralized Computer System - Ph.D Thesis, MIT - 1978.

P.M. Schwarz and A.Z. Spector - Synchronizing Shared Abstract Data Types - ACM Transactions on Computer Systems - Vol 2, Num 3 - 1984.

W. E. Weihl - Specification and Implementation of Atomic Data Types - Ph.D Thesis, MIT - 1984.

W. E. Weihl - Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types - ACM Transactions on Programming Languages and Systems - Vol 11, Num 2 - 1989.

W. E. Weihl - Linguistic Support for Atomic Data Types - ACM Transactions on Programming Languages and Systems - Vol 12, Num 2 - 1990.