Autonomy Requirements in Heterogeneous Distributed Database Systems [1]

*Panos K. Chrysanthis**, *Krithi Ramamritham***

* Dept. of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260
** Dept. of Computer Science, University of Massachusetts, Amherst, MA 01003

Abstract

In the context of multidatabase systems and heterogeneous distributed database systems, it has been observed that autonomy of the component databases has to be violated in order to maintain traditional database and transaction properties. However, very little work exists that systematically analyzes (a) the semantics of autonomy and (b) the implications of autonomy *vis a vis* correctness specifications and database protocols. Hence, this paper is aimed at characterizing the different types of autonomy by focusing on transaction management and showing the relationships between autonomy requirements and database protocols. As a case-study, we investigate the autonomy implications of the two-phase commit protocol and its multidatabase variants. Our analysis shows that these protocols involve tradeoffs between the autonomy of the transactions, with respect to accessing the data objects, and the autonomy of the transaction management system, with respect to responding to the transaction management primitives. As a result, this paper brings out the practical considerations involved in selecting between alternative protocols.

1 Introduction

Heterogeneous Distributed Database Systems (also called *Multidatabase systems* (MDBS)) logically integrate multiple pre-existing databases systems providing a uniform and transparent access to data stored in these databases. MDBSs respond to the needs of organizations to interoperate their databases already in service that support their own applications and users. An MDBS allows each local database system to continue to operate in an independent fashion. That is, an MDBS preserves the *autonomy* of the local database systems, meaning that the MDBS design (ideally) does not require any changes to existing databases and transactions, and to the local database management systems (DBMS).

Consistency of data is the primary issue in all systems in which data is dispersed over multiple databases, and in which both updates and retrievals are supported. Whereas consistency entails control over all data across the multiple databases, autonomy implies lack of any such global control. In traditional distributed databases, full consistency is ensured by *serializability* in conjunction with *failure atomicity* at the cost of autonomy [BHG87]. In the context of MDBS, it has been observed that autonomy of individual nodes or database systems has to be violated in order to maintain traditional database and transaction properties. In fact, different, and quite often inconsistent, names have been associated with different types of autonomy requirements. However, very little work exists that systematically analyzes (a) the semantics of autonomy requirements and (b) their implications *vis a vis* correctness specifications and database protocols. Hence, this paper is aimed at achieving the following:

- characterizing different types of autonomy, with emphasis on transaction management,
- explicitly showing the effect of database protocols on autonomy, and
- identifying the tradeoffs between different types of autonomy.

[1] This material is based upon work supported by the National Science Foundation under grants IRI-9109210 and IRI-9210588 and a grant from University of Pittsburgh.

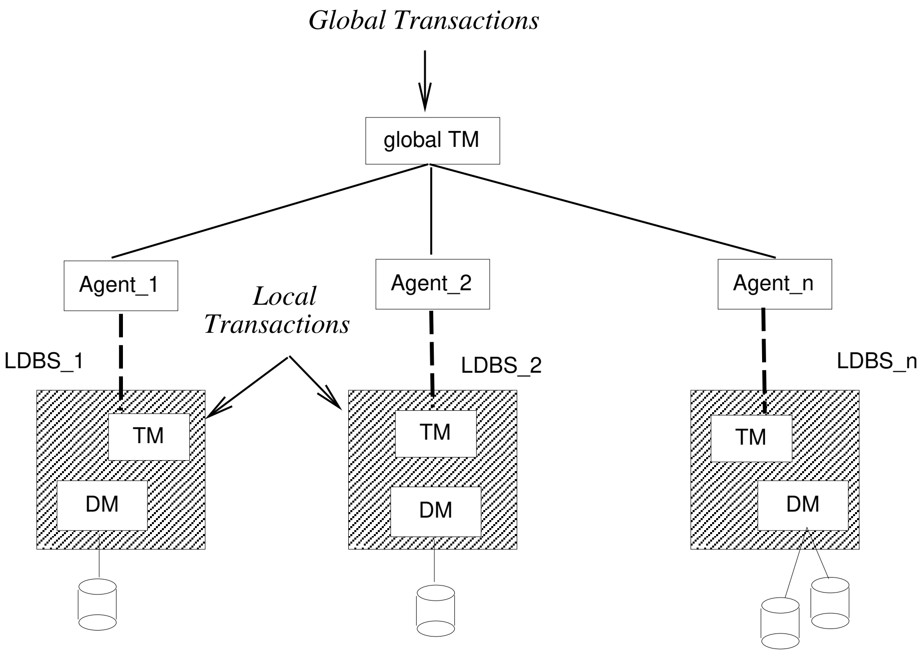

Figure 1: A Multidatabase System Model

Because of its extensive treatment in the literature [BST90, WV90, SKS91, MR+92], we have chosen the standard two-phase commit protocol and its multidatabase variants to serve as a detailed case study in our analysis. We also examine protocols designed to maintain consistency in multidatabases. Our analysis shows that in addition to the autonomy of the transaction management component of the component databases, it is important to also consider the autonomy of individual transactions; protocols entail tradeoffs between these two types of autonomy.

In this paper, we are focusing on what is usually termed *execution autonomy* [DE89, VE91, SL90, SKS91, BGS92] which refers to the ability of a local DBMS to execute operations and transaction management primitives submitted directly to it without any external interference. We give the term a broader connotation, by viewing execution autonomy from the perspective of both the transactions and the transaction management system. Specifically, we examine the implications of a particular protocol on the database operations that a transaction can (or must) invoke and on the transaction management operations a database system can (or must) invoke.

The rest of this paper is organized as follows. In Section 2, our model for transactions, databases, and multidatabases is introduced. Sections 3 and 4 form the crux of the paper. Section 3 deals with autonomy and correctness requirements while Section 4 discusses the implications, tradeoffs as well as practical considerations when database protocols are examined in the context of autonomy. Section 5 concludes the paper.

2 Databases, Multidatabases and Transactions

To set the stage to compare the autonomy implications of different database protocols, it is important to first introduce the models assumed for the database systems, multidatabase system, and the transactions.

As was mentioned earlier, an MDBS is built on top of a number of existing database systems (Figure 1). These database systems, also referred to as *nodes* of the MDBS, are traditional database systems that ensure serializability and failure atomicity.

Two types of transactions execute in an MDBS:

1. *Local transactions* that access data from only a single database and execute under the

control of the local DBMS.

2. *Global transactions* that access data from multiple databases and execute under the control of the MDBS.

Local transactions are submitted directly to the transaction manager (TM) of a local DBMS and the MDBS is not aware of their existence. Neither is a local DBMS aware of the existence of global transactions which are submitted directly to the MDBS. A global transaction G is decomposed into several *subtransactions* g_i, each of which executes on some DBMS.

Sitting above each local database system is an *agent* who is responsible for different aspects of the execution of subtransactions and in particular, of the commit protocol needed to atomically commit the subtransactions of a global transaction. These agents serve as the interface between the coordinator of a global transaction, i.e., the global TM, and the local database systems. The resulting splitting of control is a manifestation of the tension that exists between autonomy and consistency requirements, and in a multidatabase system the tradeoffs involved depend on how this split is achieved.

A transaction model defines the significant events associated with transactions that conform to that model. For instance, for the atomic transaction model, the model considered in this paper, the set of significant events that are associated with a transaction, denoted by SE, includes *Begin, Commit*, and *Abort.* A transaction also invokes operations on objects, resulting in object events.

We assume that every (local) DBMS supports a set of transaction management events, denoted by *TME. Begin*, *Commit*, *Abort*, and *Restart* belong to this set. The first three are executed, respectively, in response to the *inv(Begin)*, *inv(Commit)*, and *inv(Abort)* events associated with transactions.

We will be using ACTA formalism [CR91], a first-order logic based formalism, to precisely state transaction properties, correctness requirements, as well as the behavior of transaction processing mechanisms. In ACTA, these three aspects of a database system can be expressed as constraints on histories generated by the execution of transactions.

Using ACTA we can relate $Commit_t$ and $inv(Commit_t)$ as follows:

$$\forall t \; Commit_t \in H \Rightarrow (inv(Commit_t) \rightarrow Commit_t).$$

(The predicate $\epsilon \rightarrow \epsilon'$ is true if event ϵ *precedes* event ϵ' in H. It is false, otherwise.) Thus, the above statement states that for the event $Commit_t$ to be in the history H, i.e., for the system to have committed transaction t, it is necessary that t must have invoked the Commit operation.

The *Abort* event can also be invoked by the local DBMS in response to internal events, denoted by *IE*. The *Restart* event which may be invoked by a local deadlock detector when resolving deadlocks is an example of an internal event. *Restart* is a significant event corresponding to the abort and subsequent restart of a transaction.

This is formally expressed as follows:

$$\forall t \; Abort_t \in H \Rightarrow ((inv(Abort_t) \rightarrow Abort_t) \vee \exists \epsilon \in IE \;\; (\epsilon_t \rightarrow Abort_t)).$$

3 Specifying and Classifying Autonomy Requirements

Informally, *autonomy* represents the ability of the transactions and of the database system to execute events without any curtailment – other than those necessary for maintaining the consistency (and security) of the data. That is, this execution autonomy represents the ability of a database system to decide about the events that pertain to (the transactions executed by) it. Given that there are mainly two types of events in a database system, namely significant events and events corresponding to operations on an object, autonomy can be specified and analyzed along two dimensions: *data access autonomy*, which captures the aspects of the invocation of object events by transactions, and *transaction management autonomy*, which captures the aspects of the invocation of significant events pertaining to the transactions executing under the control of a database system.

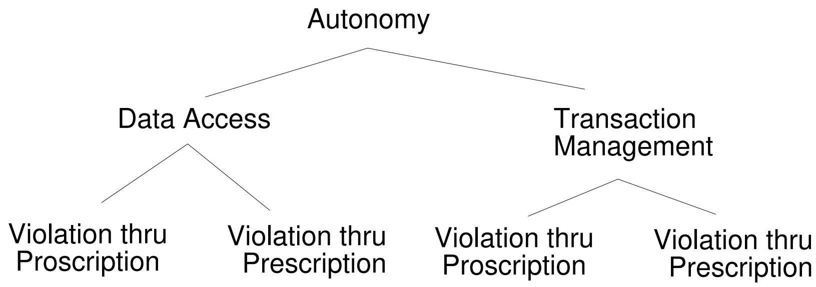

Figure 2: Dimensions of Autonomy

Also, autonomy can be studied with respect to requirements and constraints imposed on these events. There are two possible ways that autonomy can be violated: (1) by constraining or proscribing the execution of an event and (2) by requiring or prescribing the execution of an event. We refer to the former as *autonomy violation through proscription* and to the latter as *autonomy violation through prescription.* (see Figure 2).

Protocol specifications typically take the following from:

$$Condition \Rightarrow requirement.$$

Condition is a predicate on the history H of events as well as the state of the database. *requirement* is a predicate that relates to the proscription or prescription of events. Three forms of *requirement* must be specifically mentioned.

$$\epsilon \in H, \qquad \epsilon \rightarrow \epsilon', \qquad \epsilon \in SE, \epsilon' \in TME_n.$$

where TME_n denotes the transaction management events supported by node n.

For instance, the following is a specification that violates autonomy of a node via proscription, assuming that $\epsilon \in TME_n$:

$$Condition \Rightarrow \neg(\epsilon \in H).$$

Similarly,

$$Condition \Rightarrow (\epsilon \rightarrow \epsilon')$$

prescribes that ϵ be constrained to execute before ϵ'. Suppose

$$protocol\_specs \Rightarrow \forall n \ (\epsilon \in TME_n)$$

and ϵ is not normally required (see Section 2) to belong to TME of a node. Then the above violates transaction management autonomy.

Based on the above, finer classification of execution autonomy can be defined as follows:

DEFINITION 3.1: *A node n has transaction management autonomy with respect to transaction t_i* iff it is not forced to or prevented from executing a significant event (i.e., transaction management event) pertaining to t_i. That is, autonomy violations through proscription or prescription of events pertaining to t_i do not occur on node n.

DEFINITION 3.2: *A node n has transaction management autonomy* iff it has transaction management autonomy with respect to all transactions.

DEFINITION 3.3: *A transaction t has data access autonomy with respect to node n* iff it is not forced to or prevented from executing an object event[2] (i.e., data access event) relating to (data on) node n. That is, autonomy violations through proscription or prescription of object events pertaining to (data on) node n do not occur.

[2]Here we are assuming that a transaction is allowed to access all the data items in a database. That is, we are ignoring issues pertaining to security-related access control policies. These can be factored in by qualifying this definition appropriately.

DEFINITION 3.4: *A transaction t has data access autonomy* if it has data access autonomy with respect to all the nodes that it visits.

We are now in a position to define autonomy of a multidatabase system.

DEFINITION 3.5: *A Multidatabase system has (execution) autonomy* iff all its transactions have data access autonomy and all its nodes have transaction management autonomy.

Even though design autonomy which is the ability of not having to made any changes to the local DBMS in order to accommodate the MDBS system has been considered in the literature to be as a separate form of autonomy, many violations of design autonomy can be seen as instances of the violations of data access or transaction management autonomies. Consider the prescription by a database protocol of a transaction management event that is not usually supported by a node. This prescription is considered to be a violation of (node) design autonomy but it is also a violation of transaction management autonomy given our previous discussion. For another example, consider a transaction design which requires transactions to predeclare the set of all the objects they expect to access. Invocation of object events on any object outside this set is proscribed. The proscription of access to some objects leads to a violation of transaction data access autonomy. This is shown formally below:

$$\forall t\ \forall p\ \forall ob\ (p_t[ob] \in H) \Rightarrow (ob \in Predeclare(t)), \text{ i.e.,}$$
$$((ob \notin Predeclare(t)) \Rightarrow \neg(p_t[ob] \in H))$$

where $Predeclare(t)$ is the set of objects that a transaction t has predeclared.

It is not difficult to see that data access autonomy affects the data manager (DM) components of database systems since they have to ensure that data access restrictions imposed on transactions are followed. Whereas in a typical distributed database system transaction manager (TM) components are responsible for transaction management, in a multidatabase system, TMs on the individual database systems as well as the agent will be responsible. We return to this effect of autonomy on database components in the next section when we evaluate the effect of autonomy on database protocols.

We mentioned in the introduction the conflict between consistency requirements and autonomy. Before we deal with these conflicts in Section 4, it is important to note that three main approaches have emerged to address the issue of data consistency in MDBSs, each preserving different aspects of local autonomy [RP92]. The first approach attempts to guarantee multidatabase global serializability since serializability is a widely used correctness criterion [DE89, WV90, Pu88, PV88, GRS91]. This approach also includes proposals for commit protocols suitable for MDBSs [BST90, GRS91, SKS91]. The second approach replaces serializability with other correctness criteria since serializability is considered very constraining when applied to multidatabase environments. In most cases, these correctness criteria are relaxations of serializability, such as, *quasi-serializability* [DE89] and *cooperative serializability* [Ch91]. The third approach re-defines or extends the traditional transaction model to a transaction model more suitable for MDBSs with different correctness properties (See [Elm91] for a description of other *extended transaction models* proposed for different systems.) In this paper we confine ourselves to the traditional transaction model and so study the interplay between correctness criteria and autonomy in the context of this model. This is the subject of the next section.

4 Autonomy Implications of Database Protocols

In this section, we illustrate the implications for autonomy of database protocols by examining *atomic commitment protocols* that ensure failure atomicity of global transactions. We analyze the *standard two-phase commit (2PC)* protocol used in traditional distributed database systems [BHG87] and a variation of this protocol, called *emulated 2PC* (E2PC) [SKS91, MR$^+$92], explicitly designed to meet the needs of autonomy requirements in multidatabases. In fact three versions of E2PC are studied in order to show how correctness criteria can be traded off against autonomy and how different types of autonomy can be traded off against each other.

4.1 Autonomy Implications of the 2PC Protocol

The 2PC protocol has two phases, the *voting phase* during which (the coordinator of) a global transaction G requests subtransactions of the global transaction to enter the *prepare to commit* state, and the *decision phase* during which the global transaction commits if all the subtransactions are *prepared to commit* or aborts if any participant has *decided to Abort*. When a (sub)transaction is in the prepare to commit state, it can neither commit nor abort until it receives the final decision from the global transaction. This constraint is the essence of the 2PC protocol which ensures the atomicity of a global transaction, preventing subtransactions from unilaterally committing or aborting.

To investigate the autonomy properties of 2PC, we model each request by the coordinator of the global transaction as a significant event associated with subtransactions and each response as a significant event associated with the TMs of each local database. Thus, global transactions can invoke *Begin*, *PrepareToCommit*, *Commit* and *Abort*, the events in SE_{g_i} below. The *PrepareToCommit* and *DecidedToAbort* events are executed by the local databases where the subtransactions execute, in addition to the *Begin, Commit* and *Abort* events as described in Section 2. These events are in TME_n for each node n.

Definition 4.6: *Axiomatic definition of 2PC*
G denotes a global transaction with n subtransactions, g_i, $i = 1...n$.
SE_{g_i} = {Begin, Commit, Abort, PrepareToCommit}
TME_n = {Begin, Commit, Abort, PrepareToCommit, DecidedToAbort }

1. $\forall g_i \in G\ (PrepareToCommit_{g_i} \in H \Rightarrow$
   $inv(PrepareToCommit_{g_i}) \in H \wedge DecidedToAbort_{g_i} \notin H)$
2. $\forall g_i \in G\ (DecidedToAbort_{g_i} \in H \Rightarrow PrepareToCommit_{g_i} \notin H)$
3. $\forall g_i \in G\ (inv(Commit_{g_i}) \in H \Rightarrow PrepareToCommit_{g_i} \in H)$
4. $\forall g_i \in G\ (inv(Abort_{g_i}) \in H) \Rightarrow \exists g_j \in G\ (DecidedToAbort_{g_j} \in H)$
5. $\forall g_i \in G\ (Commit_{g_i} \in H \Rightarrow (inv(Commit_{g_i}) \rightarrow Commit_{g_i}))$
6. $\forall g_i \in G\ (Abort_{g_i} \in H \Rightarrow (PrepareToCommit_{g_i} \in H \Rightarrow (inv(Abort_{g_i}) \rightarrow Abort_{g_i})))$

The first two axioms, Axiom 1 and 2, capture the voting phase of 2PC protocol whereas the rest, Axioms 3 to 6, the decision phase.

Axiom 1 states that the TM of g_i sends a *PrepareToCommit* response, only if it receives a *PrepareToCommit* request from the coordinator G and it has not already sent *DecidedToAbort* response. (When the TM of g_i sends the *PrepareToCommit* response it guarantees that it is prepared to commit if the coordinator commits the global transaction. When the *PrepareToCommit* response is sent, g_i is said to enter the prepare to commit state and stays in this state until it is committed or aborted.) Axiom 2 states that the TM of a subtransaction sends a *DecidedToAbort* message to the coordinator, only if it has not already sent a *PrepareToCommit* response. As opposed to *PrepareToCommit*, a *DecidedToAbort* message is not required to be a response to a *PrepareToCommit* request; it can be sent before *PrepareToCommit* in case a subtransaction aborts before the 2PC protocol begins.

Axiom 3 states that the coordinator invokes *inv(Commit)* only if it receives *PrepareToCommit* responses from the TMs of all the subtransactions. Axiom 4 states that the coordinator invokes *inv(Abort)*, if the TM of even one subtransaction has sent a *DecidedToAbort* message. Axiom 5 states that the commitment of a subtransaction g_i can occur only after *inv(Commit)* by the coordinator. The last axiom, Axiom 6, states that in case a subtransaction aborts, if its TM had sent the *PrepareToCommit* response (i.e., the subtransaction had entered the prepare to commit state) then the subtransaction can abort only after *inv(Abort)* by the coordinator occurs.

The constraint that a subtransaction cannot be committed or aborted while being in the prepare to commit state is captured by the following lemma. The proof of this lemma using the axiomatic definition of 2PC is given in the appendix.

Lemma 1: $\forall g_i \in G\ PrepareToCommit_{g_i} \in H \Rightarrow \neg(\beta \rightarrow \gamma)$
where $\beta \in \{Commit_{g_i}, Abort_{g_i}\}$ and $\gamma \in \{inv(Commit_{g_i}), inv(Abort_{g_i})\}$

This lemma states the proscription of the commit and abort events of g_i until the occurrence of *Commit* invocation (*inv(Commit)*) or *Abort* invocation (*inv(Abort)* events, *when* g_i *is in the prepare to commit state.* The *Commit* event cannot be invoked by a local DBMS unless a transaction invokes the $inv(Commit)$ event.

$$\forall t\ Commit_t \in H \Rightarrow (inv(Commit_t) \to Commit_t)$$

and hence, the proscription of the commit event does not constitute a violation of autonomy. However, this is not the case with the *Abort* event since an abort can be caused by events other than *inv(Abort).*

$$\forall t\ Abort_t \in H \Rightarrow (inv(Abort_t) \to Abort_t) \vee \exists \epsilon \in IE\ (\epsilon_t \to Abort_t).$$

Here are the implications of this lemma:

- The transaction management autonomy of a database is violated due to proscription of the *abort* event under certain conditions. Consequently, 2PC violates the multidatabase system's execution autonomy.
- Each database, i.e., the nodes of the system, must support the prepare to commit state as captured by the *PrepareToCommit* and *DecidedToAbort events.*

  $$2PC \Rightarrow \forall n(\{PrepareToCommit, DecidedToAbort\} \subset TME_n)$$

  Such a prescription of what a database must support is beyond of what is expected from a traditional database system (as assumed in Section 2) and is according to our definition, a violation of transaction management autonomy through prescription.

Note, however, that if all databases provide for prepare to commit and we change our assumptions in Section 2 accordingly, then no violation of transaction management autonomy occurs.

4.2 Autonomy Implications of the Emulated 2PC Protocol

The Emulated 2PC (E2PC) protocol was designed explicitly with the above autonomy violations of 2PC in mind. The E2PC protocol is based on the notion of *redo* transactions. In this, operations on objects invoked by transactions are classified into *Read* and *Write* operations. The idea is that the commitment of a global transaction can be decided just between the coordinator and the (Multidatabase) agents, i.e., without the participation of the local databases. In particular, this protocol obviates the need for a database to support the prepare to commit state. If, after a subtransaction of a global transaction says that it is prepared to commit, the subtransaction is aborted but the final decision is to commit the global transaction, the writes of the aborted subtransaction are performed subsequently by a *redo* transaction. This implies that (1) the state of the database against which the redo transaction executes should be the same as the one seen by the aborted subtransaction and (2) the redo transaction should not invalidate any other active or committed (sub)transaction.

A number of schemes have been proposed to cope with ensuring the consistency of a database in the presence of redo transactions. In the rest of this section, we will discuss the autonomy ramifications of three schemes.

- In the first two schemes, which we refer to as *MSR-based E2PC*, are based on a correctness criterion called M-serializability [MR+92] rather than serializability.
- The third scheme, which we refer to as *abort-based E2PC* protocol, achieves consistency of redo transactions by aborting all the (active) transactions that conflict with the aborted subtransactions and hence, the redo of the subtransaction observes the same database state as the one seen by the subtransaction. That is, it emulates an execution where the subtransaction is not aborted but instead the other transactions suffer an internal abort [SKS91].

We would like to note that for ease of discussion, throughout this section we assume that all transactions perform updates, that is, there are no *read-only* transactions.

4.2.1 MSR-based E2PC

The MSR-based E2PC protocol is based on the notion of *Multidatabase serializability (M-serializability)* [MR+92]. The idea is that, since a redo transaction $Redo(g_i)$ is composed of the write operations of its corresponding subtransaction g_i, $Redo(g_i)$ *depends* on the read operations of g_i and hence, g_i and $Redo(g_i)$ should be considered together as a *pair* in a history irrespective of the abortion of g_i in the history. That is, database consistency is preserved by serializing all other transactions executing on the same node with respect to the object events invoked by the pair $\{g_i, Redo(g_i)\}$.

Definition of M-serializability

In order to examine the autonomy properties of MSR-based 2PC protocol, we will formally define M-serializability in terms of serialization ordering requirements induced by *conflicting* operations invoked on the same object by different transactions. In general, two operations conflict if their execution order matters.

Let T be the set of transactions executing at a node.
Let P_i be a *subtransaction,redo transaction* pair, $P_i \subseteq T$.
Let $\mathcal{C}_p$ be a binary relation on transactions in T.
Let H be the history of events relating to transactions in T.

Definition 4.7: $\forall\, t_i,\ t_j,\ t_k\ \in\ T, t_i \neq t_j, t_i \neq t_k, t_j \neq t_k\ \forall P_l \subseteq T$

$$(t_i\ \mathcal{C}_p\ t_j), \text{ if}$$
$$\exists ob\ \exists p,q\ ((t_i \notin P_l, t_j \notin P_l\ (conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))) \vee$$
$$(t_i \notin P_l, t_j \in P_l, t_k \in P_l\ (conflict(p_{t_i}[ob], q_{t_k}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_k}[ob]))) \vee$$
$$(t_i \in P_l, t_j \notin P_l, t_k \in P_l\ (conflict(p_{t_k}[ob], q_{t_j}[ob]) \wedge (p_{t_k}[ob] \rightarrow q_{t_j}[ob]))))$$

In this definition, $\mathcal{C}_p$ represents a serialization ordering requirement. The first clause expresses how an ordering requirement between two transactions which do not belong to the same pair is directly established when they invoke conflicting operations on a shared object. This is similar to the clause found in the classical definition of conflict serializability. The other two clauses reflect the fact that when a transaction establishes an ordering requirement with another transaction, the same requirement is established between the transactions in their corresponding pairs.

Let LT be the set of local transactions executing at a node, $LT \subseteq T$.
Let GT be the set of subtransactions of global transactions and
redo transactions executing at a node, $GT \subseteq T$.

Definition 4.8: H is *M-serializable* iff

$$\forall t\ (((t \in LT) \wedge (Commit_t \in H)) \vee (t \in GT))\ \ \neg(t\ \mathcal{C}_p^*\ t)$$

where $\mathcal{C}_p^*$ is the transitive closure of the relation $\mathcal{C}_p$. In words, a history H is M-Serializable if and only if in H there does not exist a committed local transaction or a committed or aborted subtransaction that is related to itself through $\mathcal{C}_p^*$. That is, C_p^* is a partial order. As mentioned above, aborted subtransactions have to be considered because for every subtransaction g_i that aborts, there is a *pair* $\{g_i, Redo(g_i)\}$, and it is with respect to such pairs that other transactions are serialized.

The above specification of M-serializability reveals that a *pair* is an instance of two cooperative transactions which maintain some consistency properties and M-serializability is a form of Cooperative serializability (CoSR) [Ch91, RP92].

Specification of MSR-based E2PC

The specification of the MSR-based E2PC protocol, first of all, differs from the 2PC protocol in Axiom 6 which defines the abort behavior of the subtransactions. (Notice that it is Axiom 6 that causes the proscription of the *Abort* event in the 2PC protocol — this can be seen in the proof of Lemma 1 in the Appendix.) Here, $Redo(t)$ denotes the *redo* transaction whose update (write) operations are the same as transaction t. In addition, it does not require any new significant events to be supported by local databases. In the emulated 2PC protocol, the *PrepareToCommit* and *DecidedToAbort* events are supported by the agent that sits above each local database system. ASE_n denotes these events that the agent at node n responds to.

DEFINITION 4.9: *Axiomatic definition of E2PC*
G denotes a global transaction with n subtransactions, g_i, $i = 1...n$.
$SE_{g_i} = \{$Begin, Commit, Abort, PrepareToCommit$\}$
$ASE_n = \{$PrepareToCommit, DecidedToAbort $\}$
$TME_n = \{$Begin, Commit, Abort $\}$

1. $\forall g_i \in G \ (PrepareToCommit_{g_i} \in H \Rightarrow$
   $inv(PrepareToCommit_{g_i}) \in H \wedge DecidedToAbort_{g_i} \notin H)$
2. $\forall g_i \in G \ (DecidedToAbort_{g_i} \in H \Rightarrow PrepareToCommit_{g_i} \notin H)$
3. $\forall g_i \in G \ (inv(Commit_{g_i}) \in H \Rightarrow PrepareToCommit_{g_i} \in H)$
4. $\forall g_i \in G \ (inv(Abort_{g_i}) \in H) \Rightarrow \exists g_j \in G \ (DecidedToAbort_{g_j} \in H)$
5. $\forall g_i \in G \ (Commit_{g_i} \in H \Rightarrow (inv(Commit_{g_i}) \rightarrow Commit_{g_i}))$
6. $\forall g_i \in G \ (Abort_{g_i} \in H \Rightarrow$
   $(inv(Abort_{g_i}) \in H \vee (inv(Commit_{g_i}) \in H \Rightarrow Commit_{Redo(g_i)} \in H)))$

The above Axiom 6 states that if a subtransaction g_i is aborted (by the local DBMS) but commit decision has been reached, its corresponding redo transaction must be executed and committed.

By involving only standard significant events, the above Axiom 6 does not violate node transaction management autonomy through prescription of unsupported events. However, this axiom is sufficient only if we have a way to achieve M-serializability. In order to guarantee M-serializability, it is sufficient to control the serialization ordering of transactions so that cyclic orderings are prevented, particularly those involving pairs of transactions. Two ways to control ordering requirements is to place restrictions (1) on the objects accessed by transactions and (2) on the object events invoked by transactions. We consider additional axioms for achieving M-serializability by examining these possibilities.

The first scheme, termed MSR-E2PC (I), prevents subtransactions from accessing any objects that are accessed by local transactions [BST90].

Let GT be the set of subtransactions at a node.
Let LT be the set of local transactions at a node.
Let L_{ob} be the set of objects accessed only by local transactions at a node.
Let E_{ob} be the set of objects accessed only by global transactions at a node.

α. $\forall ob \ \forall p \ (L_{ob} \cap E_{ob} = \phi) \wedge$
$(\forall l \in LT \ ((ob \notin L_{ob}) \Rightarrow \neg(p_l[ob] \in H)) \wedge$
$\forall g_i \in GT \ ((ob \notin E_{ob}) \Rightarrow \neg(p_{g_i}[ob] \in H))$

β. $\forall g_i \in GT \ \neg(g_i \ \mathcal{C}_p^* \ g_i)$

Axiom α states the proscription of object events invoked by local transactions and subtransactions at a node. That is, local transactions and subtransactions operate on disjoint sets of objects, L_{ob} and E_{ob} respectively. In this way, cyclic orderings due to subtransaction/redo-transaction pairs will involve only subtransactions and hence, they can be handled at the MDBS level [Axiom β]. Other cyclic orderings involving individual (local) transactions are handled by the local DBMSs since every DBMS ensures serializability (as assumed in Section 2).

In this case, MSR-based E2PC (I) protocol preserves a node's transaction management autonomy at the expense of transaction data access autonomy at the node. It violates both local and global transaction data access autonomy since a subtransaction is proscribed from invoking events on objects not in E_{ob} and a local transaction is proscribed from invoking events on objects not in L_{ob} [Axiom α].

The second scheme, termed MSR-E2PC (II), also ensures M-serializability but places less restrictions on the objects accessed by the transactions. It prevents cyclic orderings through restrictions placed on the object events invoked by the transactions. As mentioned above, M-serializability classifies object events into Read and Write events.

Let GT be the set of subtransactions at a node.
Let LT be the set of local transactions at a node.
Let L_{ob} be the set of objects accessed by local transactions at a node and which global transactions can read.
Let G_{ob} be the set of objects accessed by global transactions at a node and which local transactions can read.
Let E_{ob} be the set of objects accessed only by global transactions at a node.

a. $\forall ob\,\forall p\ (L_{ob} \cap G_{ob} \cap E_{ob} = \phi) \wedge$
$(\forall l \in LT\ ((p_l[ob] \in H) \Rightarrow ((ob \in L_{ob}) \vee ((ob \in G_{ob}) \wedge (p = Read)))) \wedge$
$\forall g_i \in GT\ ((p_{g_i}[ob] \in H) \Rightarrow ((ob \in G_{ob} \cup E_{ob}) \vee ((ob \in L_{ob}) \wedge (p = Read))))$

b. $\forall t_i, t_j \in GT \cup LT\ ((Write_{t_i}[ob] \rightarrow Read_{t_j}[ob]) \Rightarrow$
$((Commit_{t_i} \rightarrow Read_{t_j}[ob]) \vee (Abort_{t_i} \rightarrow Read_{t_j}[ob])))$

Axiom a gives the semantics of the L_{ob}, G_{ob} and E_{ob} object sets by stating the proscription of object events invoked by local transactions and subtransactions at a node. The effect of these proscriptions is that cyclic orderings due to read-write and write-read conflicts involving local transactions and subtransactions (e.g., $(Write_t[ob] \rightarrow Read_{g_i}[ob])$ and $(Write_{g_i}[ob'] \rightarrow Read_t[ob'])$, where t is a local transaction) are prevented.

Axiom b is the condition for avoiding cascading aborts by requiring that for any two transactions t_i and t_j, if t_j reads an object previously written by t_i, then t_j reads the object after t_i has either committed or aborted.

This alternative is less restrictive than the previous one because it permits global and local transactions to access common objects and hence, global and local transactions are allowed to interact by invoking *Read* and *Write* events. While this alternative is less restrictive, its specification reveals that it still violates transaction data access autonomy in more specific ways. It violates transactions data access autonomy

- by proscribing certain object events that can be invoked by certain transactions [Axiom a];
- by prescribing under which condition a *Read* event can occur [Axiom b].

4.2.2 Abort-based E2PC

The specification of the abort-based E2PC protocol has the same six axioms as the MSR-based E2PC protocol [Definition 4.9], differing only in the way of ensuring the consistency of redo transactions whose commitment is required by Axiom 6. In the abort-based E2PC protocol, consistency of a redo transaction is achieved by placing restrictions on the significant events associated with other locally executing transactions at the node where the redo transaction executes.

Even though the abort-based E2PC protocol was developed independently of MSR-based E2PC protocol, note that it also satisfies M-serializability and in this sense, it can be viewed as a third MSR-based E2PC scheme.

Let l be a local or global subtransaction.

$$\forall g_i \in G\ (Commit_{Redo(g_i)} \in H \Rightarrow$$
$$\forall l \in ConflictTr(g_i)((Abort_l \rightarrow Begin_{Redo(g_i)}) \wedge (Commit_{Redo(g_i)} \rightarrow Restart_l))$$

$ConflictTr(g_i)$ is a set of transactions which concurrently perform operations that conflict with those of g_i.

This axiom states that if g_i's redo transaction is executed, it is committed only after all the locally executing transactions which conflict with g_i are aborted. Any locally executing transaction t that conflicts with g_i is restarted after $Redo(g_i)$ commits.

	transaction management	*data access*
2PC	proscription of *Abort* prescription of $PrepareToCommit$ prescription of $DecidedToAbort$	none
MSR-E2PC (I)	none	proscription of object events
MSR-E2PC (II)	none	proscription of *Read* proscription of $Write$
abort-E2PC	proscription of *Restart* prescription of $Commit_{Redo}$ prescription of *Abort*	none

Table 1: Tradeoffs in Multidatabase 2PC Variants

The abort-based E2PC protocol neither prescribes nor proscribes any object event. Hence, it does not violate transaction data access autonomy, but it violates a node's transaction management autonomy in substantial ways by prescribing the abort of certain transactions when a subtransaction is aborted while a commit decision has been reached for the global transaction. Also, it constrains the restart of the aborted transactions.

By involving only the standard significant events pertaining to subtransactions and local transactions, the above axiom does not require any additional significant events to be supported by a database. Note, however, that the above axiom implies that the semantics of the *Commit* event of redo transactions are different from the semantics of the standard *Commit* event associated with the local transactions and subtransactions (as assumed in Section 2). Both of these *Commit* events are expected to be supported by each local database system.

In summary, the implications of the above axiom for autonomy are similar to the implications of Lemma 1 in the context of the 2PC protocol:

- The transaction management autonomy of a database is violated due to prescription of the *abort* event and proscription of the *Restart* event under certain conditions.
- Each node of the system, must support the special semantics associated with the commitment of redo transactions as captured by the $Commit_{Redo(t)}$ *event*:

$$\textit{abort-based E2PC} \Rightarrow \forall n(\{Commit_{Redo(t)}\} \subset TME_n).$$

4.3 Discussion of the Tradeoffs involved in Dealing with problems of 2PC

Table 1 summarizes the findings of the previous subsections. Recall that MSR-E2PC (I) refers to the first scheme that ensures M-serializability by preventing subtransactions from accessing objects that are accessed by local transactions and MSR-E2PC (II) refers to the second scheme that ensures M-serializability which also permits interactions between local and global transactions.

The table shows that some form of autonomy violation occurs for each of the commit protocols. Thus, the choice of a specific protocol depends on practical considerations. As discussed in Section 3, different forms of autonomy violations have implications for different components of a local DBMS. For example, if the TM of a DBMS cannot be changed to support additional transaction management events (for example to ensure atomic commitment via 2PC), then one of the MSR-based E2PC protocols has to be considered.

Both MSR-based E2PC protocols violate data access autonomy which affects the DM, and consequently, require modification of the DM if the DM does not support access control to objects in the database. At the risk of database inconsistency, the access control problem can be alleviated by assuming that transactions will be designed so that they observe any restrictions imposed on their access to objects. This could be considered as not being very different from the usual assumption made about transactions, that they are designed to obey the integrity constraints on the database.

The abort-based E2PC protocol can be used if the TM supports customization of significant events allowing us to tailor the semantics of the significant events for different types of transactions.

5 Conclusions

Our characterization of autonomy has brought out a finer classification of execution autonomy than discussed heretofore in the literature. We also showed that violations of what is usually termed design autonomy often lead to violations of execution autonomy.

We have shown that it is possible to analyze the behavior of multidatabase protocols with respect to their autonomy properties. Towards this end we have axiomatized the behavior of variations of protocols designed to ensure global transaction atomicity. This helped us to identify with the tradeoffs entailed by the different protocols. The identification of these tradeoffs will help in choosing among alternative approaches to transaction management in multidatabase systems.

We chose to study the two-phase commit protocol and its variations since they are perhaps the most investigated of multidatabase protocols. But, just as we analyzed these protocols designed to ensure failure atomicity, other multidatabase protocols designed to maintain data consistency in MDBSs can be analyzed in terms of their autonomy properties. For example, in the *ticket scheme* [GRS91], global serializability in a MDBS is achieved by forcing all subtransactions executing on a node to read and write a special object, called the *ticket*. Local transactions cannot access the ticket. Hence, the ticket scheme violates data access autonomy in a way similar to the E2PC protocols.

Another example is the notion of *quasi-serializability* [DE89] which is a relaxation of serializability. Quasi-serializability ensures data consistency provided that data dependencies do not exist across nodes.

$$\begin{aligned} &\forall g_i, g_j \in G, i \neq j \ \ \forall ob, ob' \ \forall a \\ &\qquad \neg(Read_{g_i}[ob, a] \rightarrow Write_{g_j}[ob', fn(a)]) \end{aligned}$$

That is, the value that a subtransaction g_j of a global transaction G executing on node j writes to an object ob' is not a function of a value a previously read by another subtransaction g_i of G executing on a node i.

Clearly, quasi-serializability violates data access autonomy. Checking for the data independence of subtransactions executing in different nodes requires program data dependency analysis which is outside the functionality of any DBMS and hence, there is no way that a DBMS can be changed to support it. However, since it involves only subtransactions of global transactions, such a check can be done through off-line analysis similar to that needed for MSR-E2PC or the ticket scheme.

We believe that the work presented in this paper is a necessary first step to understand the various facets and implications of autonomy. In particular, we have shown that it is possible to analyze the behavior of multidatabase protocols with respect to their autonomy properties. Such an analysis brings out the practical tradeoffs involved in achieving integration.

References

[BHG87] Bernstein P. A., V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[BS88] Breitbart Y. and A. Silberschatz. Multidatabase Update Issues. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1988.

[BST90] Breitbart Y., A. Silberschatz and G. Thompson. Reliable Transaction Management in a Multidatabase System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 215–224, May 1990.

[BGS92] Breitbart Y., H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *VLDB Journal* Vol.1, No.2, 1992.

[Ch91] Chrysanthis P. K. *ACTA, A Framework for Modeling and Reasoning about Extended Transactions.* Ph.D. Thesis. Department of Computer and Information Science, University of Massachusetts, Amherst, September 1991.

[CR91] Chrysanthis, P. K. and Ramamritham, K. A Formalism for Extended Transaction Models. In *Proceedings of the seventeenth International Conference on Very Large Databases*, September 1991.

[DE89] Du W. and A. K. Elmagarmid. Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 347–355, August 1989.

[Elm91] Elmagarmid A. K. (Editor). *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992.

[GRS91] Georgakopoulos D., M. Rusinkiewicz and A. Sheth. On Serializability of Multidatabase Transactions through Forced Local Conflicts. In *Proceedings of the IEEE Seventh International Conference on Data Engineering*, 1991.

[MR+92] Mehrotra S., R. Rastogi, Y. Breitbart, H. Korth, and A. Silberschatz. Ensuring Transaction Atomicity in Multidatabase Systems. In *Proceedings of the ACM Symposium on Principles of Database Systems*, June 1992.

[PV88] Pons J. and J. Vilarem. Mixed Concurrency Control: Dealing with Heterogeneity in Distributed Database Systems. In *Proceedings of the Fourteenth International Conference on Very Large Databases*, August 1988.

[Pu88] Pu C. Superdatabases for Composition of Heterogeneous Databases. In *Proceedings of the IEEE Fourth International Conference on Data Engineering* 1988.

[RP92] Ramamritham K. and P. K. Chrysanthis. In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties. In *Distributed Object Management*, Ozsu, Dayal, and Valduriez Ed., Morgan Kaufmann Publishers, 1993.

[RSK91] Rusinkiewicz M., A. Sheth, and G. Karabatis, Specification of Dependencies for the Management of Interdependent Data. *IEEE Computer*, 12(12):46–54, December 1991.

[SL90] Sheth A. and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[SKS91] Soparkar N., H. Korth and A. Silberschatz. Failure–Resilient Transaction Management in Multidatabases. *IEEE Computer*, 24(12):28–36, December 1991.

[WV90] Wolski A. and J. Veijalainen. 2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase. In *Proceedings of PARBASE-90 Conference*, February 1990.

[VE91] Veijalaine J. and F. Eliassen. The S–transaction Model. *Bulletin of the IEEE Technical Committee on Data Engineering*, 14(1):55–59, March 1991.

A Proof of Lemma 1

In order to prove Lemma 1 (page 10), we use the proof rule:

$$(\epsilon \rightarrow \epsilon') \Rightarrow \neg(\epsilon' \rightarrow \epsilon)$$

and the following lemma which follows directly from Axioms 3 and 4:

LEMMA 2:

$$\neg((inv(Abort_{g_i}) \in H) \wedge (inv(Commit_{g_i}) \in H))$$

Proof of Lemma 1: We will prove the lemma in four parts, each corresponding to a particular combination of (β, γ).

1. $\forall g_i \in G\ ((PrepareToCommit_{g_i} \in H) \Rightarrow \neg(Commit_{g_i} \rightarrow inv(Commit_{g_i})))$

Assume $(PrepareToCommit_{g_i} \in H)$. Assume $(Commit_{g_i} \rightarrow inv(Commit_{g_i}))$. $(Commit_{g_i} \rightarrow inv(Commit_{g_i}))$ implies that $(Commit_{g_i} \in H)$ is true and according to Axiom 5, $(inv(Commit_{g_i}) \rightarrow Commit_{g_i})$ is also true. However, $(inv(Commit_{g_i}) \rightarrow Commit_{g_i})$ implies $\neg(Commit_{g_i} \rightarrow inv(Commit_{g_i}))$ which contradicts the assumption.

2. $\forall g_i \in G\ ((PrepareToCommit_{g_i} \in H) \Rightarrow \neg(Commit_{g_i} \rightarrow inv(Abort_{g_i})))$

Assume $(PrepareToCommit_{g_i} \in H)$. Assume $(Commit_{g_i} \rightarrow inv(Abort_{g_i}))$. $(Commit_{g_i} \rightarrow inv(Abort_{g_i}))$ implies that both $(Commit_{g_i} \in H)$ and $(inv(Abort_{g_i}) \in H)$ are true. From Axiom 5, $(Commit_{g_i} \in H)$ implies $(inv(Commit_{g_i}) \rightarrow Commit_{g_i})$ which in turn implies $(inv(Commit_{g_i}) \in H)$. Thus, $(inv(Commit_{g_i}) \in H) \wedge (inv(Abort_{g_i}) \in H)$ which contradicts Lemma 2.

3. $\forall g_i \in G\ ((PrepareToCommit_{g_i} \in H) \Rightarrow \neg(Abort_{g_i} \rightarrow inv(Commit_{g_i})))$

Assume $(PrepareToCommit_{g_i} \in H)$. Assume $(Abort_{g_i} \rightarrow inv(Commit_{g_i}))$. $(Abort_{g_i} \rightarrow inv(Commit_{g_i}))$ implies that $(Abort_{g_i} \in H)$ and $(inv(Commit_{g_i}) \in H)$. Since $(PrepareToCommit_{g_i} \in H)$ is true, $(Abort_{g_i} \in H) \Rightarrow (inv(Abort_{g_i}) \rightarrow Abort_{g_i})$, according to Axiom 6. $(inv(Abort_{g_i}) \rightarrow Abort_{g_i}) \Rightarrow (inv(Abort_{g_i}) \in H)$. Thus, $(inv(Commit_{g_i}) \in H) \wedge (inv(Abort_{g_i}) \in H)$ which contradicts Lemma 2.

4. $\forall g_i \in G\ ((PrepareToCommit_{g_i} \in H) \Rightarrow \neg(Abort_{g_i} \rightarrow inv(Abort_{g_i})))$

Assume $(PrepareToCommit_{g_i} \in H)$. This implies $(Abort_{g_i} \in H) \Rightarrow (inv(Abort_{g_i}) \rightarrow Abort_{g_i})$, according to Axiom 6. $(inv(Abort_{g_i}) \rightarrow Abort_{g_i})$ implies $\neg(Abort_{g_i} \rightarrow inv(Abort_{g_i}))$.

□