Active Database Management of Global Data Integrity Constraints in Heterogeneous Database Environments[1]

Lyman Do and Pamela Drew[2]

Department of Computer Science

Hong Kong University of Science and Technology

Clear Water Bay, Hong Kong

email: {*lyman, pam*}*@cs.ust.hk*

Abstract

Today, enterprises maintain many, disparate information sources over which complex business procedures and applications are executed. The informal and *ad hoc* characteristics of these environments make the information in them very prone to inconsistency. Yet, the flexibility of application execution given to different parts of an organization is desirable. This paper introduces a new mechanism in which the execution of asynchronous, pre-existing, yet related, applications can be harnessed. A multidatabase framework that supports the concurrent execution of these applications with heterogeneous, distributed transactions is presented. Using this framework, we introduce an intuitive conceptual model and algorithm for the enforcement of interdatabase constraints and outline an implementation based on active database technology.

Contact Information:
Pamela Drew
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
FAX: 852 358-1477
Phone: 852 358-6971
email: *pam@cs.ust.hk*

[2]Shortened Version in Proceedings of IEEE 11th International Conference on Data Engineering, Taipei, March, 1995.

[2]This research was supported by the Sino Software Research Center, contract numbers SSRC92/93.006 and SSRC93/94.EG10.

Contents

1 Introduction

Today's business enterprises can be viewed as a network of multiple heterogeneous information sources over which various complex business procedures are executed. These information sources are the cornerstone of many organizational activities from decision making to the rapid development of new information-based products and services. Hence, the quality and accessibility of this information can be vital to an organization's success. These information systems are traditionally built to automate existing data-intensive business functions, such as billing, that are otherwise performed manually in separate organizational entities. By automating these functions separately, an enterprise typically ends up with many stand-alone systems between which related information may be distributed and not shared.

Using this set of information systems is an ever changing set of enterprise-wide operations. As business requirements change, new procedures are introduced and others become obsolete. Many of these procedures are composed of activities that manipulate information at different sites [2, 4, 12, 13]. While many of the tasks in these activities are automated, they are frequently invoked at different times and by different parts of the organization. In some cases, certain dependencies between the applications can be modeled and should be enforced [2, 33]. In other cases, these applications can be executed with little constraint on their order or the time elapsed between their completion. In still other cases, an activity may require all steps to be concurrently satisfied immediately. Furthermore, any particular step of a procedure could update some information which, unbeknownst to it, has related information stored in other parts of the organization to which the update should be propagated.

This type of environment is common and very prone to inconsistent management of data across the enterprise. Most of these procedures are executed through informal, and frequently manual processes, so rules about their execution are not always enforced. Further, it is unlikely, and inappropriate, for a particular office procedure, such as "update the customer's address", to propagate its updates across the enterprise, especially if the application was designed originally in a stand-alone fashion.

Automated support to rectify these problems must strike a balance between reliably maintaining the consistency of the information stored in the environment and supporting flexible, autonomous execution of existing applications that comprise complex enterprise operations. The multidatabase framework described in this paper takes a step in this direction. It is a transparent transaction processing substrate upon which complex enterprise

operations can execute. It has the following characteristics:

1. Flexible execution of autonomous, pre-existing applications is supported seamlessly and forced invocations of them are triggered only when regular business procedures fail. These business procedures may be manual or supported by an automated work flow management system [4, 12].

2. Interrelated data stored in heterogeneous databases can be maintained consistently even if an application updates only part of the interrelated information.

3. Existing applications operating on interrelated information can execute concurrently with new distributed applications that need to share the same data.

Most of the prior work on multidatabase transaction management [5, 6, 18, 19, 22, 27, 31] has focussed on ways to integrate the transaction processing algorithms of disparate DBMSs to achieve some correctness criteria such as global serializability [3], or an approximation of it such as quasi-serializability [17]. Some propose that changes be made to the local database system [19, 31], while others propose that restrictions be placed on how global transactions are submitted to the local databases [18] and how they can interact with local transactions [6]. Furthermore, those algorithms that do allow local transactions to execute in the presence of global transactions [22, 28, 29, 38] require global transactions to be initiated by the global transaction manager and tightly control their execution strategies.

The framework proposed in this paper departs from this work in at least three significant ways. First, this framework makes the unique contribution of "loosely coupling" pre-existing, local applications that update related information across disparate stores into multidatabase transactions so that their execution can be managed as a whole. These transactions could be thought of as global transactions that have been designed in a "bottom-up" rather than "top-down" fashion. However, the execution of these loosely-coupled multidatabase transactions is far more flexible than what distributed transaction management protocols prescribe for the execution of global, "top-down", distributed transactions. This framework triggers the execution of pre-existing applications with a transparent mechanism that does not violate their autonomy during regular execution. It combines the advantage of consistent updates across multiple stores usually provided by a global transaction management system with the flexibility of autonomous execution of local applications.

Second, this framework incorporates the management of global "top-down" multidatabase transactions, local transactions that do not access information with related counterparts in other systems, and loosely-coupled "bottom-up" multidatabase transactions into one unified framework. In the prior multidatabase transaction management work outlined above, local transactions are generally modeled as single autonomous transactions that share information with global transactions on an individual basis. However, like liaison transactions executing at their respective local sites on the behalf of a global transaction, a set of local applications distributed at different sites can have relationships between them that need to be managed in a multidatabase framework. The framework proposed here is the only one, to our knowledge, that addresses this very real, and practical, aspect of complex, distributed information processing environments.

And third, unlike prior multidatabase transaction management work, it is not our goal to devise new heterogeneous transaction management algorithms that enforce global serializability. Indeed, the framework can be tailored to a number of different correctness criteria (including global serializability) and transaction management algorithms. We outline one implementation in [15] that allow inconsistency between related information stored at disparate sites through unserializable interleavings of the various types of transactions in the architecture. It is however bounded, and can be removed. In this paper, we outline another implementation of this framework which relies on active database technology, and ensures serializability. In both cases, the overall goal of harnessing existing interrelated transactions together with global and local transactions into one unified multidatabase framework and retaining the autonomous execution of the pre-existing local applications is maintained.

We start, in Section 2, by defining a model for the framework that is useful for understanding the update characteristics in the system. Section 3 details the consistency constraints in multidatabase environments. Section 4 proposes an algorithm and describes how it can maintain the consistency constraints. In section 5, we suggest an implementation of the algorithm based on the active database technologies. And lastly, Section 6 outlines some other research related to this approach and Section 7 concludes the paper.

2 A Multidatabase Framework

In this section, we outline the major architectural components in our framework and a categorization of data and transactions upon which we base our multidatabase management algorithms.

2.1 Architecture

The structure of our multidatabase framework is shown in Figure 1. The multidatabase system (MDBS) is implemented over a set of pre-existing, autonomous database systems, $\{LDB_1, LDB_2, \ldots, LDB_m\}$, each of which manages its own data and supports executing, autonomous applications. The creation of this structure serves two purposes. First, using the global transaction manager, applications can manipulate data in several local databases simultaneously without having to know about the location or characteristics of each. Second, it provides a mechanism to automatically maintain consistency between related pieces of information stored across the local sites even if a transaction is submitted to update only one of them. In this fashion, the architecture provides a transparent overlay onto pre-existing systems which allows them to interoperate in a way that ensures consistency of the related information that they store.

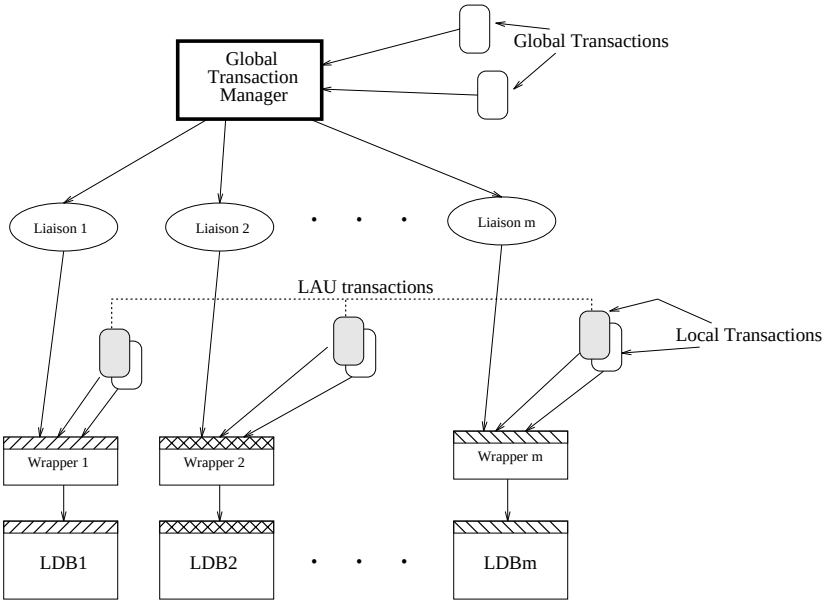

Figure 1: MDBS Model

The transactions executing in this architecture can be partitioned into three types. First, local transactions (LTs) are those submitted to a local site to update information that has no related counterpart stored elsewhere. They are denoted $LT = \bigcup_{i=1}^{m} LT_i$ for m local databases, where $LT_i = \{LT_{i,1}, LT_{i,2}, \ldots, LT_{i,n}\}$ is a set of local transactions in LDB_i. Second, local asynchronous update transactions (LAUs) are those submitted to a local site to update information that does have some related counterpart (e.g. a replica) stored in other local databases in the MDBS. And third, global transactions (GTs) are

those submitted to the global transaction manager to access several data items stored in different local sites. A global transaction, GT_i, consists of a set of subtransactions, $\{GT_{i,1}, GT_{i,2}, \ldots, GT_{i,m}\}$ where subtransaction $GT_{i,j}$ accesses local database LDB_j. A transaction is a sequence of read and write operations and ended with either a *commit* or an *abort* operation.

A local schedule is a sequence of transaction operations executed at a local database. A global schedule is the combination of all local schedules. A global subschedule [38] is global schedule restricted to the set of global transactions in the global schedule.

The execution of global transactions is carried out by the global transaction manager (GTM), shown at the top of Figure 1. The GTM maintains information to translate global transaction requests into local requests (and vice versa) and the current state of the transactions in the system. It communicates with the local systems via a liaison process, depicted as ovals in Figure 1, that executes at a local site. Each liaison is responsible for translating the global request into the local system's language and submit it to the local database (LDB). We assume that these transactions at each local site are managed by a local transaction manager (LTM).

For each local database, there is a wrapper process that maintains information about transactions updating information which has related counterparts stored in other local databases in the MDBS. This information is used to help maintain consistency of the local copy; this process is detailed in section 4. As the slanted lines in Figure 1 depict, the wrapper provides the same interface as the local database so that existing applications can continue to execute without modification in the multidatabase architecture. We assume that the LTMs are not aware of each other, and that if a local transaction is submitted to a LTM, no other LTM is aware of that transaction. LTMs perform their operations without the knowledge of other LTMs or the GTM. Furthermore, we assume that each LTM interface accepts at least read, write, commit, and abort operations.

2.2 Types of data

Intuitively, there are two types of data in this model: *local data* and *global data*. Local data is information that has no related counterpart in other information sites. It was either pre-existing in the local databases or created by local transactions after the development of the MDBS. Global data, on the other hand, is related information that needs to be managed "globally" to maintain its consistency, though this management process may itself be distributed. This information may have been designed independently prior to the

development of the multidatabase or created later by global transactions via the global transaction manager. We refer to global data that are created *prior* to the development of the multidatabase as *conceptually interrelated data* and those that are created *later* by global transactions via the global transaction manager as *interrelated data*. We say this information is *conceptually interrelated* since, for example, there may be data items that are conceptually equivalent, but implemented differently in the autonomous sites. The word "conceptually" describes the property that these "global" data will be updated by local transactions.

There are a number of ways in which information can be interrelated across local databases [2, 37]. One example, that we will use for illustration in the remainder of this paper, is replication. We say that two data items are *conceptually replicated* (*CRep*) if it has been determined that they are in fact referring to the same concept, but have been implemented differently on their respective heterogeneous platforms. We refer to these conceptually replicated data items as CRep data items.

2.3 Local Asynchronous Update Transactions (LAUs)

LAUs allow applications to update conceptually related information at a local site as if it were the only copy. Hence, a pre-existing local application can autonomously update any information in the multidatabase architecture, without changing its code or making other special provisions, and still be guaranteed that all information related to it in other autonomous databases will eventually be updated accordingly. The multidatabase architecture does this with a mechanism that triggers the execution of related LAUs at the corresponding local sites in a very flexible, yet controlled manner. This mechanism is achieved through the wrapper processes at each local site in the architecture.

It is important to note that our multidatabase framework is designed to be as unobtrusive to existing database environments as possible. Hence, the automated mechanisms for consistency maintenance are unbeknownst to users. Furthermore, they work in concert with regular business operations. That is, if regular business operations work properly, the MDBS can detect it, and if they fail, the system ensures that updates are made automatically. In [15], this was achieved by allowing a certain grace period during which the system would wait for regular transactions (local applications) that implemented appropriate LAUs to execute. If the grace period expired, the CRep was automatically brought into a consistent state by the MDBS. This paper describes a more aggressive consistency mechanism in which the MDBS automatically restores consistency without waiting for

users to initiate local applications (LAUs). However, if an LAU is submitted manually (as part of regular business operations), the MDBS can detected that is a duplicate update and simply returns an affirmative status to the user. This mechanism is described in section 4.

For each LAU updating a CRep data item in a local database, we assume there are other LAUs which update the corresponding CRep data items in other local databases. These LAUs are grouped conceptually into a *global LAU* ($GLAU$). A GLAU, $GLAU_1$, is a set of LAUs, $LAU_{1,A}, LAU_{1,B}, \ldots, LAU_{1,M}$, which perform semantically equivalent updates on conceptually replicated data items. The implementation of this architecture described in [15] actually instantiates GLAUs in data structures of the GTM and manages their execution through coordinated activities of the GTM and the wrapper processes. However, in the implementation described in this paper, GLAUs are virtual transactions in the sense that no data structure is created specifically to track their creation, or completion, or which LAUs define them. Instead, the execution of the interrelated LAUs is automatically managed independently by the wrapper processes without creating a GLAU data structure. This is made possible by the use of active database technology, as described in section 5.

3 Consistency Constraints in a Multidatabase Architecture

In this section, we first outline some of the types of consistency constraints which should be enforced in a multidatabase environment. We then introduce a new notation, called a *data dependency* graph, for representing these constraints. A data dependency graph provides a conceptual model for the implementation of constraint enforcement algorithms in multidatabase architectures. The graph serves as a reference model to describes how information is related in a multidatabase architecture and what precedence is used in performing updates. Moreover, a data dependency graph can be used as the basis for a optimization of a consistency management algorithm; a discussion of this optimization process is beyond the scope of this paper.

It is interesting to note that the particular implementation we propose in section 5 does not make explicit use of a data dependency graph or other global information. The consistency enforcement is managed inherently by the active database technology; yet it still adheres to the relationships described by the graph. Another implementation of this algorithm, on the other hand, could require the graph to be stored explicitly in some global

data structures as frequently found in distributed transaction management systems.

3.1 Consistency Constraints

A database is said to be consistent if it satisfies a set of consistency constraints [24, 28]. For example, an Equivalence constraint on replicated data requires the values of replicated data to be equivalent. Consistency constraints can be grouped into two categories. Global consistency constraints are defined over distributed data whereas local consistency constraints are defined over data in a single database. In multidatabase environments, local consistency constraints are maintained by LDBs and the global consistency constraints are maintained by the MDBS.

Global consistency constraints can be partitioned into two categories. The first category are defined on *conceptually interrelated* data created prior to the development of the multidatabase. The second category are defined on interrelated data created by global transactions via the global transaction manager. It is very difficult to enforce the global consistency constraints that belong to the first category. *Conceptually interrelated* data can be accessed by both local and global transactions. In the multidatabase environments, it is almost impossible to coordinate the execution of local transactions and global transactions in a synchronous manner when local autonomy is to be maintained [10, 17]. If local autonomy is preserved, local transactions and global transactions are interleaved arbitrarily. As a result, global consistency constraints in the first category can be violated. The following example illustrates this phenomenon.

Example 1

> Consider a multidatabase that consists of three local databases, LDB_A, LDB_B and LDB_C. There are three CRep data items, x_A, x_B and x_C, located at LDB_A, LDB_B and LDB_C respectively. Suppose there are three LAUs, $LAU_{1,A}, LAU_{1,B}$, and $LAU_{1,C}$, at each local database. Each LAU reads the CRep data item and increases it by 10.
>
> $$LAU_{1,A} : x_A := x_A + 10$$
> $$LAU_{1,B} : x_B := x_B + 10$$
> $$LAU_{1,C} : x_C := x_C + 10$$
>
> There is a *CRep Equivalence constraint* which asserts that the values of three

CRep data items are equivalent:

$$\textit{CRep Equivalence constraint} : x_A \equiv x_B \equiv x_C$$

Assume local transactions can be submitted, executed, and committed autonomously. Suppose $LAU_{1,A}$ is executed and committed, without losing generality. The CRep Equivalence constraint is violated after $LAU_{1,A}$ commits because x_A is now greater than x_B and x_C by 10.

In [15], we defined the inconsistency in the CRep data items of Example 1 as being *temporally inconsistent* and proposed a mechanism that eventually brings all CRep items up to date. Another type of inconsistency, called *interleaved inconsistency*, was also introduced; it is created in CRep data items if other transactions, either local or global, are allowed to update a CRep data item while temporal inconsistency exists (e.g. a new update transaction interleaves with $LAU_{1,A}$, $LAU_{1,B}$ and $LAU_{1,C}$). In this case, database inconsistency could grow in an uncontrolled manner. [15] proposes a mechanism to control this type of inconsistency and remove it when the level becomes too high. The implementation described in this paper, on the other hand, allows temporal inconsistency in the databases but avoids transactions reading temporally inconsistent data and avoids interleaved inconsistency altogether as outlined in section 4.

3.2 Data Dependency Graphs

Data dependency (*DD*) graphs are a new notation for representing consistency constraints in multidatabase architectures. They provide a conceptual model for various implementations of consistency enforcement algorithms. Causal dependency is one special type of relationship depicted in the DD graphs and is a unique property of multidatabase architectures in which LAUs can execute on CRep data items. For two data items, x and y where the value of y is derived from the value of x, if x is a CRep data item, then we say that that y causally depends on a consistent value of x (that is, the CRep Equivalence constraint for x is satisfied). In distributed databases, it is unnecessary to state the causal dependency constraint explicitly because replicated data are consistent. However, in multidatabases, CRep data item are seldom consistent. The Causal Dependency constraint has to be specified explicitly in order to ensure the correct execution of transactions.

The DD graph is a mixture of a directed and an undirected graph. Data are represented as nodes and the edges represent relationships between them. A data item x is related

to data item y if there is an edge between them. We use a directed edge to represent the causal dependency and an undirected edge to represent non-causal relationship. A directed edge from x to y means that the value of y depends on consistent value of x.

In the DD graph, we use a dotted circle to represent a CRep Equivalence constraint. All related CRep data items have undirected edges to a virtual node (represented as dotted circle). In example 1, if the equivalence constraint on x_A, x_B and x_C is maintained, transactions that access the CRep data item x can arbitrary access x_A, x_B or x_C without worrying about which one is the consistent copy. We use a dotted directed edge to represent that CRep Equivalence constraints are always enforced. Note that dotted nodes and edges represent concepts but not real objects or dependency. The following example shows how to use DD graph constructs to represent various relationships between data items.

Example 2

Consider the following consistency constraints:

$$y < z$$

$$x_A \equiv x_B \equiv x_C$$

Suppose there is a transaction that updates y based on the value of x_A, for instance, $y = x_A + 10$ [3]. We use a causal dependency to describe the fact that y depends on the consistent value of x_A. Figure 2 shows the data dependency graph. The virtual node (dotted circle) represents the CRep Equivalence constraint of x_A, x_B and x_C. A virtual directed edge (dotted line) from x to y is used to represent the implication that if the CRep Equivalence constraint is enforced, the value of y can depend on any one of x_A, x_B, or x_C, i.e., y can depend on the virtual data node x.

We observe that the Causal Dependency constraint is guaranteed if the CRep Equivalence constraint is maintained [4]. Therefore, once CRep Equivalence constraints are maintained, Causal Dependency constraints are maintained implicitly. From Example 2, this means that y can use any value from x_A, x_B, and x_C, in its calculation as long as the

[3] This does not imply a consistency constraint $y < x_A$ or $y = x_A + 10$ because only y depends on the value of x_A but not vice versa.

[4] The reverse is not true because a data item can be causally dependent on a consistent CRep data item although the CRep Equivalence constraint is violated.

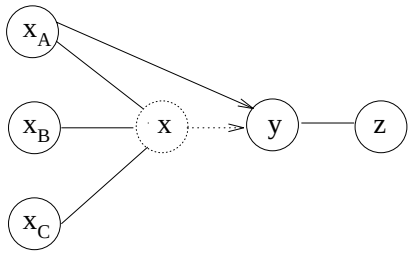

Figure 2: Data Dependency Graph

CRep Equivalence Constraint over x is maintained. In the next section, we show how consistency constraints can be enforced in the multidatabase environments.

4 Enforcing Consistency Constraints in Wrappers

Various techniques for consistency constraint enforcement in distributed database systems [30, 35], such as differential relations or assertions, cannot be used to describe the Causal Dependency constraints and the CRep Equivalence constraints in this architecture. These algorithms will detect that all LAUs are violating the CRep Equivalent contraints and discontinue their execution. In this section, we propose an algorithm that can handle these special types of consistency constraints without aborting any transaction. Then, in the next section, we suggest a possible implementation and show that this implementation can be generalized to handle all types of consistency constraints in multidatabase environments.

4.1 Data Structures

For each CRep data item in the local database, there are two lists of LAUs associated with it in its respective wrapper process. One list is called the *"To-Be-Submitted"* (*TBS*) list and the other is called the *"Have-Committed"* (*HC*) list. The *TBS* list records the sequence of "to-be-submitted" LAUs, which have to be submitted to the LDB in order to restore the consistency of CRep data item. The *HC* list records those "have-committed" transactions, which have already been submitted by the wrapper process and successfully committed. The *TBS* list is used to restore consistency of CRep data items and the *HC* list is used to trap any possible duplicate submission of an LAU from a local application. Each entry in the list stores the LAUs identifier and the transaction's parameters. This information can be used to decide whether two LAUs are semantically equivalent by comparing their identifiers and the parameters. These LAUs in the lists are stored in the chronological order.

For each CRep data item, there is a consistency indicator associated with it in the respective wrapper process. This consistency indicator states whether this CRep data item is consistent with respect to other CRep data items in the MDBS and if it is inconsistent, the consistency indicator will indicate which CRep data item is consistent. If a CRep data item is updated by an LAU, all related CRep data items in other local databases are set to be inconsistent. Referring to the DD graph in Figure 2, if x_B is being updated by an LAU, x_A and x_C are said inconsistent, and so does the virtual node x. A consistency marker is created for each virtual node on the DD graph. The consistency marker points to the CRep data item that is consistent. If the CRep Equivalence constraint is maintained, i.e. all CRep data items are consistent, and equal to each other, the consistency marker points to the respective virtual node.

4.2 Enforcing CRep Consistency and Causal Dependency Constraints

We propose a new *Dynamic Primary Copy* (*DPC*) algorithm to enforce CRep Equivalence and Causal Dependency constraints in our MDBS framework. The CRep Equivalence constraint is violated by asynchronous execution of LAUs. The Causal Dependency constraint is violated if a data item is causally dependent on a CRep data item whose CRep Equivalence constraint is currently violated. The *DPC* algorithm enforces these constraints using the *TBS* and the *HC* lists of CRep data items. Transactions on the *TBS* list of CRep data item x can be classified into two categories. A transaction falls into the first category if x is the only data item it updates. A transaction falls into the second category if it updates data items besides x. The DPC algorithm has two major components — the *virtual execution* which handles the first category of transactions and the *actual execution* which handles the second category of transactions.

Virtual Execution overwrites the inconsistent CRep data item with the consistent value from the CRep data item to which the consistency indicator points. Since the overwriting does not require the actual execution of the transactions on the *TBS* list, the *Virtual Execution* procedure is more efficient than the *Actual Execution* procedure. For those transactions on the *TBS* list updating other data items except the CRep data item, overwriting the CRep data item is not enough and the algorithm has to update all data items that the transaction supposes to update. *Actual Execution* procedure is designed to handle these transactions.

4.2.1 Virtual Execution

The *Virtual Execution* (*VE*) procedure simulates the execution of the transactions on the *TBS* list of the CRep data item x_j by copying the value of the consistent CRep data item x_i over it when a transaction requires an consistent copy. We refer to transactions on a *TBS* list as TBS transactions (TBSTrxns). The *VE* procedure takes advantage of the fact that we know which CRep data item in the MDBS is consistent using the consistency indicator associated with each CRep data item. It is triggered when there is a transaction requesting an CRep data item that is currently inconsistent. We refer to this transaction as a *requesting transaction* (*RTrxn*). We use the following example to illustrate the steps in the *VE* procedure.

Example 3

Suppose there are two CRep data items, x_A and x_B in LDB_A and LDB_B respectively and $x_A = x_B = 10$ initially. The first graph in Figure 3 depicts the initial DD graph. The virtual node x^* shows that x_A and x_B are involved in

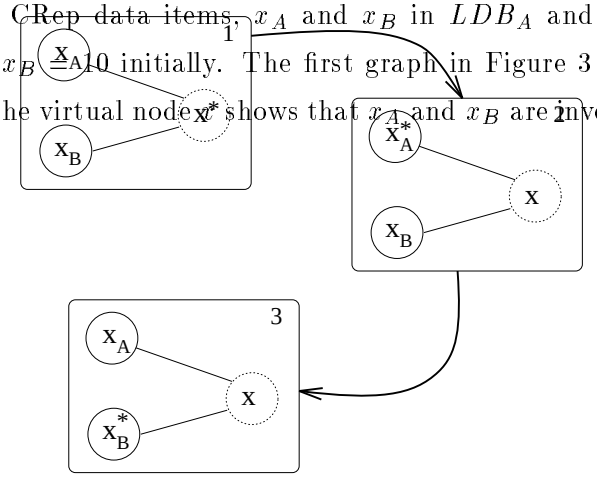

Figure 3: Data Dependency Graph of Example 3

a CRep Equivalence constraint. The asterisk in the virtual node x represents the consistency marker of x and shows that the CRep Equivalence constraint is maintained. There are two GLAUs, $GLAU_1$ and $GLAU_2$ and they have LAUs in both local databases. Suppose all LAUs are accessing x_A or x_B only. Consider the scenario that $LAU_{1,A}$ of $GLAU_1$ is submitted to LDB_A to increase x_A by 10 and committed. The consistency indicator of x_B is set to $FALSE$ by the wrapper process B and $LAU_{1,B}$ is now on the *TBS* list of x_B. This means that $LAU_{1,B}$ has to be submitted before any transaction accesses x_B so that the CRep Equivalence constraint is enforced. The second DD graph in Figure 3 depicts the DD graph after $LAU_{1,A}$ is committed. The consistency marker on x_A node indicates x_A is the consistent copy.

Suppose that $LAU_{2,B}$, which multiplies x_B by 2, is submitted to wrapper B. The *VE* procedure is triggered before the RTrxn, $LAU_{2,B}$, is submitted to LDB_B. The *VE* procedure will copy the consistent value of x, i.e., x_A to x_B, and then move the TBSTrxn, $LAU_{1,B}$, entry from the *TBS* list of x_B to its *HC* list. We refer to the copy operation as the *pre-requisite action* of the transaction $LAU_{2,B}$ and the moving of the TBSTrxn from the *TBS* list to HC list as the virtual execution of the TBSTrxn. x_B is now consistent and the CRep Equivalence constraint is maintained. The RTrxn, $LAU_{2,B}$, can be submitted to LDB_B. After the RTrxn commits, CRep Equivalence constraint is violated again. Now, the x_B is the consistent copy and the $LAU_{2,A}$ is on the *TBS* list of x_A. The respective DD graph is shown as the third DD graph in Figure 3.

The CRep data item marked by the consistency marker is an analog to the primary copy update [8] that deals with data replication in distributed databases. However, in our DPC algorithm, every CRep data item can be the primary copy and the location of the primary copy is dynamically changed. As the TBSTrxn, $LAU_{1,B}$, is executed virtually by copying the effect of its corresponding LAU ($LAU_{1,A}$), we use the *HC* list for x_B to trap any duplicate transaction $LAU_{1,B}$ that might be submitted by a local user manually later on. If the wrapper traps the $LAU_{1,B}$ and determines that it is a duplicate of one on the HC list, it will signal that $LAU_{1,B}$ commits to the user, though it will not actually execute it. We refer to this sequence of operations as the *Virtual Execution* of $LAU_{1,B}$.

4.2.2 Actual Execution

Virtual Execution is not sufficient to maintain consistency if a TBSTrxn updates more than one data item. For example, if the TBSTrxn, $LAU_{1,B}$, in example 3 updates y in addition to x_B, *Virtual Execution* only reflects the update on x_B because it only copies the consistent value of x to x_B but does not update the value of y. *Actual Execution* (AE), on the other hand, will ensure that y is updated as well. The *Actual Execution* procedure submits all updates to the local databases rather than updating wrapper structures as *Virtual Execution* does. Lists and consistency marker manipulations in *Actual Execution* are the same as they are in the *Virtual Execution*.

Example 4

Consider further the scenario in which the TBSTrxn, $LAU_{1,B}$, of example 3, updates y in addition to x_B. When the RTrxn, $LAU_{2,B}$, is submitted to wrapper B, the TBSTrxn, $LAU_{1,B}$, is identified as the pre-requisite action of $LAU_{2,B}$ by the AE procedure and it is submitted to LDB_B by the wrapper process. During the execution of $LAU_{1,B}$, $LAU_{2,B}$ is suspended. When the $LAU_{1,B}$ commits, the x_B becomes consistent and y is updated as well. The RTrxn, $LAU_{2,B}$, can resume its execution (being submitted to the LDB_B by the wrapper process). Nested triggering of transactions may result if the $LAU_{1,B}$ accesses another CRep data item, say z_B, which is inconsistent. In this case, $LAU_{1,B}$ will be suspended until all TBSTrxns of z_B are either virtually or actually executed.

4.2.3 The DPC algorithm

The purpose of the DPC algorithm, shown in Figure 4, is to ensure all CRep data items in T's *Read* set are consistent. Each transaction T accesses two sets of data — the *Read* set and the *Write* set. For each CRep data item, x, in T's *Read* set, an instance of *DPC_Pre(T,x)* is initiated before T is submitted to the local database. The major part of the *DPC_Pre* procedure is to execute all pending transactions on x's *TBS* list. For each TBSTrxn U, if x is the only data item it updates, the *VE* procedure will be executed. Otherwise, the AE procedure will be executed. After all TBSTrxns are committed, the consistency indicator of x will be set to $TRUE$. When all *DPC_Pre* procedures for the *Read* set of T are completed, the data items in the *Read* set are guaranteed consistent and T will be submitted to the local database.

Before T is committed in the local database, for every CRep data items x in the *Write* set of T, *DPC_Post(T,x)* will be executed. The *DPC_Post* procedure updates the corresponding *TBS* list of all related CRep data items in other local databases and their consistency indicator. When the *DPC_Post* complete, T will be committed in local database.

4.3 Correctness of the DPC algorithm

As mentioned previously, the correctness of the algorithms used in this architecture can be tailored to different criteria. The DPC algorithm maintains serializable execution of any transaction, i.e. GLAUs and GTs, that access CRep data items. In addition, this

```
    END;
  ENDIF

  IF TBS(x) is empty THEN
    END;
  ELSE
    IF T is in TBS(x) THEN
      TBS'(x) = all transactions U before T;
    ELSE
      TBS'(x) = TBS(x);
    ENDIF

    WHILE TBS'(x) is not empty
    BEGIN
      U = HEAD(TBS'(x));
      REMOVE U FROM TBS(x) and TBS'(x);
      APPEND U TO HC(x);
      IF number of element in write-set(U) is 1
        Vitual-Execute(U);
      ELSE
        Actual-Execute(U);
      ENDIF
    END /*WHILE*/
    SET consistency-indicator(x) to TRUE;
  ENDIF

END



DPC_Post(T,x)

BEGIN
  For each site i
    FIND the LAU transaction V that belongs
    to the same GLAU of T;
    FIND the CRep y that links to the same
    virtual node in DD graph that x links;
    APPEND V to the end of TBS(y);
    SET consistency-indicator(y) to x;
  ENDFOR
END
```

Figure 4: The Dynamic Primary Copy (DPC) algorithm

architecture contains a GTM to manage the execution of general GTs that can access non-CRep data as well. Depending on the type of concurrency control algorithm implemented in the GTM, the overall consistency criteria of the system will vary. In this section, we describe the different types of schedules and associated correctness criteria in this implementation of our framework.

There are three types of schedules in this architecture. First, a local schedule is a sequence of transaction operations executed at a local database. Local schedules in this framework consist of transactions from local applications, LAUs, and subtransactions of GTs. The LTMs maintain serializable local schedules.

Second, a global subschedule [38] is the global schedule restricted to the set of GTs in the system. Given that a global concurrency control algorithm, such as the one proposed in [23], is implemented in the GTM, the global subschedule is also serializable.

Third, we introduce a new type of schedule for multidatabase architectures, called the *GLAU subschedule*, which is the global schedule restricted to the set of GLAUs in the system. The DPC algorithm ensures that the GLAU subschedule is serializable. Suppose there are two GLAU transactions, $GLAU_1 : LAU_{1,A}, LAU_{1,B}, \ldots, LAU_{1,m}$ and $GLAU_2 : LAU_{2,A}, LAU_{2,B}, \ldots, LAU_{2,m}$, accessing CRep data items $x_A, x_B, \ldots, x_m$ in

$LDB_A, LDB_B, \ldots, LDB_m$ respectively. Assume that $LAU_{1,i}$, accessing x_i, is the first LAU among all LAUs of $GLAU_1$. Later, there will be $LAU_{1,j}$ where $j \neq i$ accessing x_j at LDB_j. Also assume that $LAU_{1,i}$ accesses x_i before $LAU_{2,i}$, without losing generality. Then, the DPC algorithm ensures that all $LAU_{1,j}$, where $j \neq i$, are executed before all $LAU_{2,j}$. The proof is trivial. When $LAU_{1,i}$ accesses x_i and commits, $LAU_{1,j}$, where $j \neq i$, is appended on the respective TBS list of x_i. When $LAU_{2,j}$ executes, $LAU_{1,j}$ will be either virtually or actually executed by the DPC algorithm before $LAU_{2,j}$ executes. $GLAU_1$ is serialized before $GLAU_2$.

The DPC algorithm ensures the serializability of all transactions which access CRep data items. This includes GLAUs and GTs, but not local transactions since LTs, by definition, do not access CRep data items. Hence, we also need to consider the correctness criteria enforced on the interleavings of GLAUs and GTs. It can easily be shown that the DPC algorithm also maintains a serializable execution of these types of transactions. The argument is similar to the one above by replacing one of the GLAU with a global transaction.

Now consider the global schedule that is the interaction of local transactions, LAUs (i.e. GLAUs), and global transactions. If the GTM module in the MDBS is equipped with any traditional distributed concurrency control algorithm in which local indirect conflict is unavoidable, the global schedule in our framework is equivalent to QSR [5] [17]. QSR requires the global subschedule to be serializable, which is maintained by the GTM and the DPC algorithm, and local schedules to be serializable, which are maintained by LTMs. If the GTM module in the MDBS is equipped with a multidatabase concurrency control algorithm, in which local indirect conflict is avoided, such as the one proposed in [23], the global schedule in our framework is serializable. The GTM module enforces serializable interaction between global transactions and local transactions and the DPC algorithm ensures serializable execution of global transactions and LAUs.

It is worth noting that the DPC algorithm is designed explicitly to ensure consistency for multidatabase architectures in which GLAUs exist – which includes most practical heterogeneous database environments. Prior multidatabase algorithms and architectures, such as those designed to maintain QSR [18] and the approach proposed in [23], do not acknowledge the existence of GLAUs, and, therefore, cannot maintain a consistent multidatabase in our framework. These algorithms don't manage the concurrent execution of LAUs with other types of transactions, nor do they maintain the consistency of CRep data

[5] Local indirect conflict is allowed in QSR.

items and or the Causal Dependency constraint. Hence, it is important to understand that though the resulting global schedule in this architecture can be serializable or QSR (depending on the global concurrency control implementation), the architecture itself solves a significantly different, and enhanced, problem from that of prior multidatabase work.

5 Enabling Technologies

The wrapper process has an active role in the MDBS. It monitors the transactions and automatically triggers consistency enforcement mechanism. To facilitate the implementation of the DPC procedures, the wrapper process should fulfill two requirements. First, it should have efficient and effective storage management to manage constraints and relevant information, such as the *TBS* and the *HC* lists. Second, it should provide the facilities to specify and evaluate the constraints, which can be specified as a set of rules. The active database database management systems which embed rules in a DBMS [11, 36] fulfill the requirements of the wrapper process. In this section, we describe an implementation of wrapper processes applying the active database technology.

5.1 An Active Database Implementation

Active database management systems allow rules to be specified declaratively. The *Event-Condition-Action* (*ECA*) rule proposed in [9, 14] allows the triggering event and condition of rules be specified. An event can be a database operation, a temporal event or a signal from arbitrary processes. The condition is a predicate, which can be specify as a database query, and the action part specifies a program. If the condition is satisfied, the action is executed. The system monitors the events, evaluates conditions and triggers the corresponding actions when the conditions become true, and schedules tasks to meet the timing requirements, without user or application intervention.

5.2 ECA rules

For each CRep Equivalence constraint which describes the relationship among data items, $x_1, x_2, \ldots, x_m$, $2m$ ECA rules will be generated: two for each CRep data item. Figure 5 depicts the two generic ECA rules.

The ECA rules enforce the CRep Equivalence constraints and the Causal Dependency constraints at the same time. Recall that the Causal Dependency constraint states that the data item being updated depends on another consistent data item. The *Read ECA* rule

```
CLASS: Update
Parameter: T: Triggering Transaction
           x: CRep accessed by T
EVENT: Update(x)
CONDITION: T is LAU
ACTION: DPC_Pre before Update
        DPC_Post after Update before Commit

CLASS: Read
Parameter: T: Triggering Transaction
           x: CRep accessed by T

EVENT: Read(x)
CONDITION: consistent_indicator(x) = FALSE
ACTION: DPC_Pre before Read
```

Figure 5: The generic ECA rules

ensures that every CRep data item is made consistent before being read. As a result, the Causal Dependency constraint is maintained. The *Update ECA* rule records the activities that may violate the CRep Equivalence constraints.

The *Update ECA* rule ensures that the *DPC_Pre* procedure is executed before the *update* operation of the triggering transaction. This guarantees that any pending transaction on the *TBS* list is executed before the data item is overwritten. The *DPC_Post* procedure is executed before the triggering transaction is committed. The *Read ECA* rule ensures that the *DPC_Pre* procedure is executed before the *read* operation. Figure 6 shows an example of a triggering transaction after the corresponding ECA rules are triggered. The elegant advantage of applying active database technology in this framework is that every rule functions autonomously. There is no global information, such as the data dependency graph, stored in the multidatabase systems. Hence, behavior usually implemented by separate control processes in a MDBS or distributed DBMS can be elegantly distributed to each local site and executed autonomously.

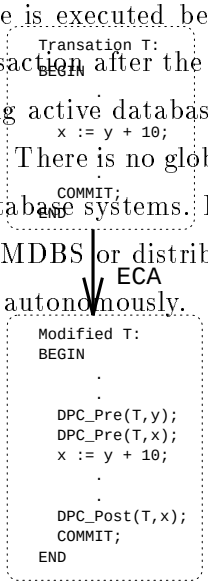

Figure 6: The ECA Triggering

As the wrapper processes implement the event monitoring and rule triggering functions, general data integrity constraints can be specified as ECA rules [25]. As a result, all kinds of data integrity constraints in multidatabase environments can be specified systematically and enforced autonomously.

For general data integrity constraints, the event part specifies the operation that the constraint has to be examined [6]. The condition part specifies the data integrity constraint, such as referential integrity, non-null attribute, unique-key and arithmetic relationships. If the data integrity constraint is violated (the condition becomes true), the action part will abort the triggering transaction.

6 Other Related Work

In addition to the various multidatabase transaction management algorithms outlined in the introduction, research related to our approach can be categorized by three areas: multidatabase architectures, management of interrelated data dispersed across heterogeneous databases, and algorithms for the management of replicated data in distributed systems. In this section, we compare our approach with representative contributions to each of these fields.

Various multidatabase architectures have been proposed in systems such as Omnibase [32], Interbase [20], *A la carte* [16], and GTE's Distributed Object Management [7] project. All of these systems share the approach in our framework of using liaison processes and wrappers to provide access to heterogeneous database management systems. The Local Access Manager of Omnibase and the Remote System Interface of Interbase each act as an interface at respective local sites. The Autonomous Manager in *A la carte* and the Local Application Interface in DOM each take this idea one step further by serving as a placeholder for local concurrency control mechanisms, if a local store does not provide one itself. In either case, the liaison processes act mostly in a passive capacity responding to liaison transaction requests, but provide no global management capabilities. In our framework, on the other hand, the wrapper processes play an important part in the consistency management of the overall architecture. They track the execution of pre-existing transactions over interrelated data in a loosely-coupled way. Thus the wrappers are active contributors to the overall transaction management in the framework rather than passive interfaces.

More closely related to our approach are the agents proposed in [34] for the management of nested transactions in federated database architectures. These agents are comprised of various components, such as an analyzer and a scheduler, that play an active part in the management of local transactions created from nested transactions executing

[6] In most of the cases, it will be the update operation.

over a distributed architecture. The wrapper processes in our framework share this active role, but focus on the management pre-existing transactions over interrelated data instead.

Another related area is that work on the management of conceptually related data in heterogeneous environments. A prominent contribution is the Interdatabase Dependency Schema (IDS) [21, 33] in which data dependencies are specified and managed. Polytransactions [21], which include system generated component transactions, are initiated by the IDS to maintain consistency of data according to the specified data dependencies. Our framework has a similar goal in the management of conceptually related information. However, a primary goal of our framework is to manage the execution of pre-existing transactions that are already updating interrelated information. This goal is achieved in addition to maintaining interdatabase dependencies via global transactions designed in a "top-down" fashion (e.g. Polytransactions in IDS and GTs in our framework).

The work on *Quasi-Serializability* (*QSR*) [17, 18] is relevant to the scheduling of *pre-requisite actions* in the wrapper process during the execution of the DPC algorithm. QSR is a correctness criterion for the MDBS. It is less restrictive than serializability which is difficult to maintain with the local autonomy requirement. QSR requires all local histories to be serializable and the global subschedule to be serializable. It assumes that there is no global data integrity constraint (except the restrictive global replication in which local transactions cannot directly update replicated data) and there is no intra-transaction dependency. The DPC algorithm and the GTM module can produce a global schedule equivalent to QSR. However, our framework focuses on the enforcement of global data integrity constraints on data items at different sites and the management of the execution of pre-existing local transactions which access conceptually replicated data items. Prior multidatabase frameworks and associated correctness criteria, such as QSR, do not manage these types of transactions.

Lastly, algorithms to manage replicated data efficiently, such as Quasi-copy [1] and Lazy Replication [26], have relevance to this work. In the quasi-copy approach, updates are made at a pre-defined primary site or set of sites. Inconsistency is allowed between the primary site and other replicated sites, and can be restored by sending the updated image from the primary site to the replicated sites. However, generally in multidatabase environments, and certainly in the GLAU framework proposed here, it is most unlikely to have a primary site for conceptually replicated data. In our framework, an LAU can update a *CRep* data item at any site. Thus, our framework is more flexible and able to accommodate many different interwoven enterprise operations initiated anywhere in the organization.

The Lazy Replication approach requires clients (transactions in the terminology of this paper) to specify a predefined, causal order that defines which other transactions it depends on. This information is used to determine in what order transactions can access replicated data. This approach is also incompatible with our framework since LAUs are pre-existing, autonomous transactions that are not aware of each other. It is true, however, that *TBS* list maintained by the wrapper process for each *CRep* data item maintains a list of transaction execution. This list is somewhat similar to the predefined ordering in Lazy Replication in that the system tracks information about which transactions should execute to make the item consistent. The difference is that, in our framework, the list is a dynamic structure that can adapt to any sequence of transaction execution, whereas the Lazy Replication approach limits transaction access *a-priori* according to the predefined sequence.

7 Conclusion

We have introduced a framework that can be used to combine the flexible execution of related, pre-existing applications with more traditional distributed transaction processing. Using this framework, information that is related either through some existing enterprise operations, or by basic definitions such as replication, can be managed consistently even if it is stored in heterogeneous databases.

An important design goal of this framework is to be able to accomodate a spectrum of autonomy in multidatabase management. At one end of the spectrum, local systems surrender autonomy to be tightly controlled by a global multidatabase management system; though a rather impractical approach, consistency enforcement can be guaranteed. At the other end of the spectrum, local systems maintain total autonomy with little or no coordination between them; the flexibility of this approach may come at the price of (temporarily) inconsistent information in the architecture.

The approach described in this paper is just one point on this spectrum in which traditional consistency criteria and local autonomy are maintained using active database technology. In this implementation, the wrapper processes and associated triggering procedures are implemented as active database applications. Other future work include study of how the DPC algorithm can be optimized and refined, how this framework supports flexible transactional work flow models and the utility of this architecture in the migration of heterogeneous legacy systems into new computing architectures.

References

[1] R. Alonsa, D. Barbara, and H. Garcia-Molina. Quasi-copies: Effecient data sharing for information retrieval systems. In J.W. Schmidt, S. Ceri, and M. Missikoff, editors, *Advances in Database Technology - EDBT '88*. Springer-Verlag, Berlin, 1988.

[2] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using flexible transactions to support multi-system telecommunications applications. In *Proceedings of 18th International Conference on Very Large Data Bases*, Vancouver, 1992.

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

[4] Y. Breitbart, A. Deacon, H. Schek, A. Sheth, and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *ACM SIGMOD RECORD*, 22(3):23–30, September 1993.

[5] Y. Breitbart and A. Silberschatz. Multidatabase update issues. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data*, pages 135–142, 1988.

[6] Y. Breitbart, A. Silberschatz, and G. R. Thompson. Reliable transaction management in a multidatabase system. In *Proceedings of ACM-SIGMOD 1990 International Conference on Management of Data*, pages 215–224, 1990.

[7] A. Buchmann, M. Tamer Özsu, M. Hornick, D. Georgakppoulos, and F.A. Manola. A transaction model for active distributed object systems. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[8] S. Ceri and G. Pelagatti. *Distributed Databases Principles & Systems*. McGraw-Hill, New York, 1984.

[9] S. Chakravarthy and et. al. HiPAC: A research project in active time-constrained database management. Final technical report, Xerox Advanced Information Technology, Cambride, Mass., July 1989.

[10] P. K. Chrysanthis and K. Ramamritham. Impact of autonomy requirements on transactions and their management in heterogeneous distributed database systems. In *Proceedings of the workshop on interoperability of database systems and database applications*, pages 179–197, University of Fribourg, Switzerland, October 1993.

[11] U. Dayal. Active database management systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness*, Jerusalem, Israel, June 1988.

[12] U. Dayal, H. Garcia-Molina, M. Hsu, B. Kao, and M.C. Shan. Third generation TP monitors: A database challenge. In *Proceedings of ACM-SIGMOD 1993 International Conference on Management of Data*, pages 393–397, Washington, D.C., May 1993.

[13] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long-running activities. In *Proceedings of 17th International Conference on Very Large Data Bases*, August 1991.

[14] U. Dayal and D.R. McCarthy. The architecture of an active database management system. *ACM SIGMOD RECORD*, 18(2), June 1988.

[15] L. Do and P. Drew. The management of interdependent asynchronous transactions in heterogeneous database environments. Submitted to *The Third International Conference on Parallel and Distributed Information Systems*. Also Technical report HKUST-CS94-4, The Hong Kong University of Science and Technology, Department of Computer Science, March 1994.

[16] P. Drew, R. King, and D. Heimbigner. A toolkit for the incremental implementation of heterogeneous database management systems. *The VLDB Journal*, 1(3):241–284, October 1992.

[17] W. Du and A. K. Elmagarmid. Quasi serializablity: a correctness criterion for global concurrency control in InterBase. In *Proceedings of Fifteenth international Conference on Very Large Data Bases*, pages 347–355, 1989.

[18] W. Du, A. K. Elmagarmid, and W. Kim. Maintaining quasi serializability in multidatabase systems. In *Proceedings of Seventh International Conference on Data Engineering*, pages 360–367, Kobe, Japan, 1991.

[19] A. K. Elmagarmid and W. Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of Sixth International Conference on Data Engineering*, pages 37–46, Los Angeles, CA, 1990.

[20] A.K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for InterBase. In *Proceedings of 16th International Conference on Very Large Data Bases*, pp. 507-518, Brisbane, Australia, August 1990.

[21] S. Gantimahapatruni and G. Karabatis. Enforcing data dependencies in cooperative information systems. In *Proceedings of the International Conference of Intelligent and Cooperative Information Systems*, Rotterdam, The Netherlands, May 1993.

[22] D. Georgakopolous, M. Rusinkiewicz, and A. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 314–323, Kobe, Japan, 1991.

[23] D. Georgakopolous, M. Rusinkiewicz, and A.P. Sheth. Using tickets to enforce the serializability of multidatabase transactions. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), February 1993.

[24] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[25] A. Kotz, K. Dittrich, and J. Mulle. Supporting semantic rules by a generalized event/trigger mechanism. In *Proceedings of the International Conference on Extending Database Technology*, Venice, Italy, March 1988.

[26] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Canada, August 1990.

[27] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. The concurrency control problem in multidatabases: Characteristics and solutions. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data*, pages 288–297, 1992.

[28] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Non-serializable executions in heterogeneous distributed database systems. In *Proceedings of the First International Conference on parallel and Distributed Information Systems*, pages 245–252, Miami Beach, Florida, 1991.

[29] S. Mehrotra, R. Rastogi, H.K. Korth, and A. Silberschatz. A transaction model for multidatabase systems. In *Twelfth International Conference on Distributed Computing Systems*, Yokohama, Japan, 1992.

[30] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[31] C. Pu. Superdatabases: Transactions across database boundaries. In *Proceedings of the Third International Conference on Data Engineering*, 1987.

[32] M. Rusinkiewicz, R. Elmasri, B. Czejdo, D. Georgakopoulos, G. Karabatis, A. Jamoussi, K. Loa, and Y. Li. Query processing in a heterogeneous multidatabase environment. In *Proceedings of the First Annual IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, 1989.

[33] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *Computer*, 24(12):55–63, December 1991.

[34] W. Schaad and H. Schek. Federated transaction management using open nested transactions. In *Proceedings of the workshop on interoperability of database systems and database applications*, pages 210–222, University of Fribourg, Switzerland, October 1993.

[35] E. Simon and P. Valduriez. Integrity control in distributed database systems. In *Proceedings of 19th Hawaii International Conference on System Sciences*, pages 622–632, Honolulu, January 1986.

[36] M. Stonebraker. Triggers and inference in database systems. In M.L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*. Springer-Verlag, New York, 1986.

[37] W.S. Tang. *LACE: A Specification Language for Consistent Access to Heterogeneous Database Environments*. Master thesis, The Hong Kong University of Science and Technology, Department of Computer Science, 1993.

[38] A. Zhang and A.K. Elmagarmid. A theory of global concurrency control in multidatabase systems. *The VLDB Journal*, 2(3):331–360, July 1993.