Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports
Department of Computer Science

1994

Principles and Realization Strategies of Intregrating Autonomous Software Systems: Extension of Multidatabase Transaction Management Techniques

Aidong Zhang

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
94-006

Zhang, Aidong and Bhargava, Bharat, "Principles and Realization Strategies of Intregrating Autonomous Software Systems: Extension of Multidatabase Transaction Management Techniques" (1994). *Department of Computer Science Technical Reports.* Paper 1109.
https://docs.lib.purdue.edu/cstech/1109

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

PRINCIPLES AND REALIZATION STRATEGIES OF INTEGRATING AUTONOMOUS SOFTWARE SYSTEMS: EXTENSION OF MULTIDATABASE TRANSACTION MANAGEMENT TECHNIQUES

Aidong Zhang
Bharat Bhargava

Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD TR-94-006
January 1994

Principles and Realization Strategies of Integrating Autonomous Software Systems: Extension of Multidatabase Transaction Management Techniques

Aidong Zhang and Bharat Bhargava
Department of Computer Science
Purdue University
West Lafayette, IN 47907 USA

Abstract

A multidatabase system integrates a set of autonomous local databases that can be accessed as a single unit. Such a multidatabase environment is actually a special case of a more general software development environment in which local components may be either database systems or file systems. This paper discusses the new issues that arise in such software development environments and presents solutions for these issues by extending multidatabase task management techniques.

1 Introduction

A software development environment (SDE) is a distributed heterogeneous software system in which local components can be either database systems or file systems. These local systems originally ran in isolation to support their individual tasks. It then became evident that more complex tasks involving multiple systems could be supported through intersystem cooperation. Consider an example at BNR [BCD+93]. A group of engineers at corporate headquarters is responsible for maintaining switching-system quality. An task software package, the Statistical Analysis System (SAS), is used to analyze available data. From the SAS output, they develop performance and reliability graphs, which are stored in a DB2 database on an IBM mainframe. This information is then be used by design engineers to improve the switching-system design. The integration of these systems must, however, be accomplished without the disruption of local system autonomy.

Such an autonomy feature has been recognized and studied in multidatabase systems. A multidatabase system (MDBS) serves to integrate a set of local database systems at various locations (sites). The central concern of such an integration is the preservation of the local autonomy of the component database systems. Aspects of autonomy such as design, execution, and control have been studied in [GMK88, BS88, DE89, Vei90]. MDBSs process two varieties of tasks. Each local task accesses a local database only and is submitted directly to a local database system. Global

tasks, in contrast, may simultaneously access several local databases and are submitted to an integration phase, where they are parsed into a set of global subtasks to be submitted to local database systems.

Thus, an SDE is a generalized case of an MDBS. The techniques developed in MDBSs may be extended to SDEs. The aspects of the integration on system and language designs have been studied in [TS93, BCD+93]. In this paper, we investigate the task of multidatabase task management techniques to the decentralized software development environment. In such an environment, the conditions that can be placed on local sites must be more relaxed than those which may be in effect in a multidatabase environment. For example, there may be no concurrency control mechanisms in place at local sites, and prepare-to-commit states may also not be supported at local sites. We will discuss approaches that can be developed to ensure the correct execution of global and local tasks in this less restrictive environment.

2 Preliminaries

In this section, we shall provide a precise definition of the system under consideration and introduce the task model that is used in this paper.

2.1 Decentralized System Model

As software development environments usually deal with a large number of local components, a decentralized approach to integration is essential for SDEs. Such a decentralized approach provides a high degree of fault-tolerance, and the system can be easily extended to accommodate new local sites. We therefore anticipate that the decentralized design of global task management will become an important feature of SDEs, particularly of those systems integrating a large number of participating local software systems.

An SDE consists of a set of {local software systems, denoted LSS_i, for $1 \leq i \leq m$}, where each LSS_i comprises an autonomous software system on a set D_i of data at the local site LS_i and a global task manager (GTM). The GTM is decentralized on all machines participating in the SDE. Each LSS is associated with a GTM server (GS), and all machines participating in the SDE can run the GTM interpreter (GI). A global task is submitted by invoking a process of the GTM interpreter at its machine while a local task is submitted directly to a LSS. All machines are connected by a computer network. Figure 1 illustrates this architecture.

The GTM interpreter manages the decomposition and execution of global tasks. In particular, the execution of a global task G_i is controlled by the GTM interpreter process GI_i, which submits the subtasks of G_i to the relevant GTM servers for execution. A GTM interpreter process can independently manage the execution of a global task without requiring any knowledge of the others'

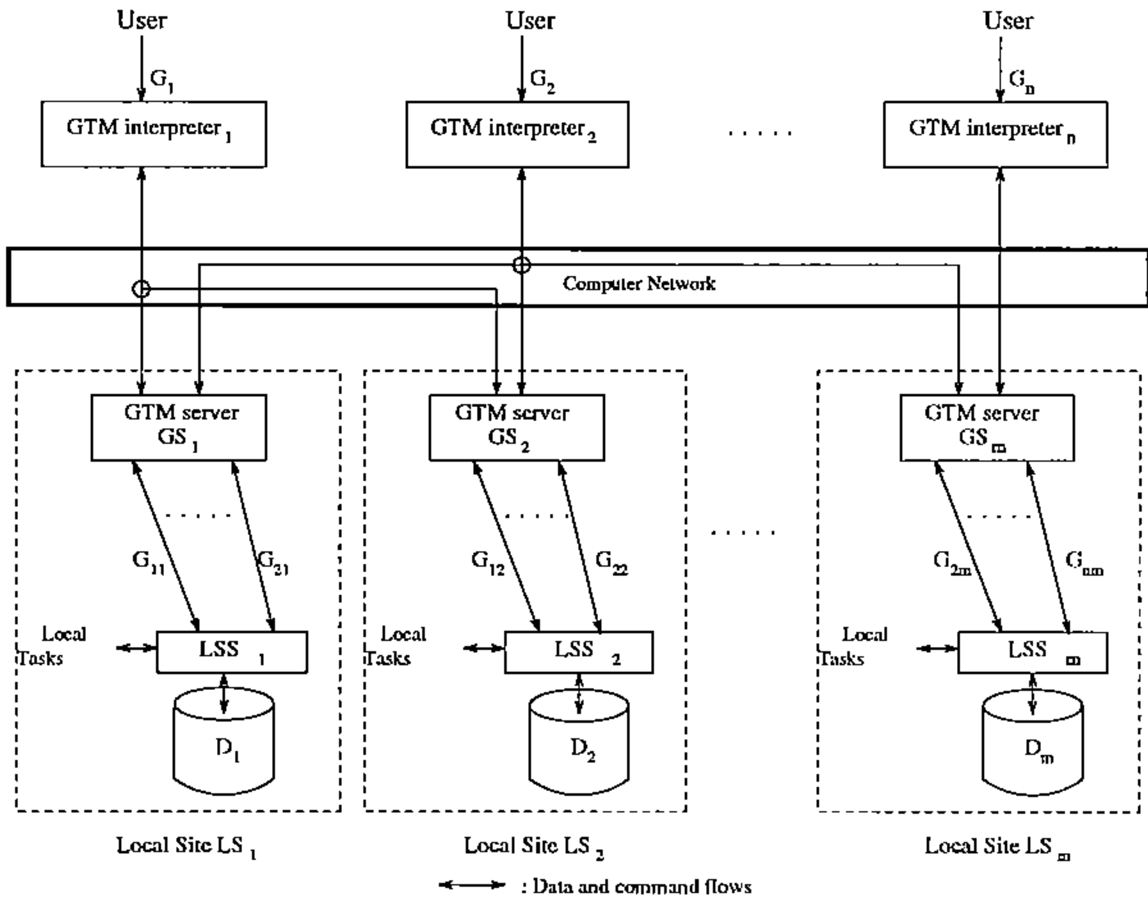

Figure 1: Decentralized SDE architecture

existence.

A GTM server is responsible for the execution of global subtasks received from the GTM interpreter processes. It then submits for execution each global subtask to the LSS at its associated site. The completion of each subtask is acknowledged by the LSS to the GTM server, which, if necessary, returns these results to the GTM interpreter processes. Each GTM server runs independently from other GTM servers and coordinates only with the GTM interpreter processes from which it receives global subtasks.

2.2 Open Nested Flexible Task Specification

We propose a two-level open nested flexible task model by extending the flexible transaction model in the multidatabase environment to include more features. Such an extension is necessary to enrich global task model for advanced tasks.

To support SDE applications that may generate long-running tasks more efficiently, we propose to apply the strategy of Sagas to flexible transactions. An open nested flexible task consists of a set of relatively independent tasks, where each task is a flexible transaction. Here, to be different from Sagas and Interaction, we explore the model that will allow flexibility on the set of tasks. In other words, there may have alternative choices for each task. To the further extension, we allow each task to be a flexible transaction. Thus, our two-level open nested flexible task model provides two-level of flexibility: flexibility on the tasks of an open nested flexible task and flexibility on the subtasks of a flexible task in the open nested flexible task.

In contrast to other extended transaction models, such as open nested transactions [Mos81], multi-level transactions [BSW88], and sagas [GMS87], relaxing atomicity and isolation are the goals for open nested flexible tasks. In a manner similar to traditional transactions, an open nested flexible task must be a unit of consistent and reliable computation. Thus, we must provide the means to justify the consistency and reliability of the execution of an open nested flexible task. Traditionally, the ACID properties (atomicity, consistency, isolation, and durability) [Gra81, HR83, ÖV91] have been advanced as the justification of the consistency and reliability of transactions. Clearly, the concept of atomicity can be relaxed for open nested flexible tasks, since some tasks of an open nested flexible task may be aborted while the open nested flexible task as a whole succeeds.

3 Non-conventional Commit Protocols

In this section, we shall investigate the maintenance of the correct execution of each individual global task in the error-prone SDE.

3.1 Flex-atomicity

The global tasks in SDEs may be more flexible than global tasks in MDBSs. In other words, some actions in a global tasks may not have to be executed or may be replaced by other actions if they fail. For example, in a debugging global task, the failure (existing bugs) of one source code module may be replaced by other version of the same module and also, partial execution of all source code modules may be acceptable. This flexibility allows a global task to adhere to a weaker form of atomicity, which we term *flex-atomicity*, while still maintaining its correct execution in the SDE environment. Flex-atomicity allows a global task to commit even if some subtasks fail, provided that either the failed subtasks are unnecessary to be executed or their alternative subtasks complete.

3.2 Maintaining Flex-atomicity

To preserve the flex-atomicity of global tasks, given the assumption that LSSs can recover from failures, the GTM must ensure either that all global subtasks of a global task that must complete commit or that none of the effects of each global subtask remain permanent.

Preserving the atomicity [BHG87] of global tasks in multidatabase systems has been recognized as an open and difficult issue [SKS91]. Of particular concern is the fact that multidatabases cannot assume that the local databases support a visible prepare-to-commit state for those subtasks in which a subtask has not yet been committed but is guaranteed the ability to commit. This scenario clearly remains problematic in the SDE environment. In such situations, a local software system that participates in an SDE environment may unilaterally fail a global subtask without agreement from the global level (termed *a local unilateral fail*). As a result, it becomes difficult to ensure that a single complete logical action of the subtasks in a global task that must complete is consistently carried out at multiple local sites. The traditional two-phase commit (2PC) protocol developed in distributed database environments thus becomes inadequate to the preservation of the flex-atomicity of global tasks in the SDE environment.

Both forward and backward recovery approaches which utilize the redo, retry, and compensation techniques have been proposed [BST90, MRKS92] for the preservation of the semantic atomicity [GM83] of global tasks. These techniques allow each global subtask to commit unilaterally, requiring either the redo or retrial of aborted global subtasks or the undoing of tentatively committed global subtasks by corresponding compensating tasks. Note that at most one subtask of each global task can be pivot. The compensatable subtasks must be completed before the completion of the pivot subtask, which in turn must complete before the completion of the retriable subtasks. The global complete/fail decision is determined by the outcome of the completion of the pivot subtask. If it fails, all of the compensatable subtasks are compensated for; otherwise the retriable subtasks are

attempted until they complete.

Flex-atomicity is also weaker than semantic atomicity. In order to utilize the above techniques in the SDE environment, we categorize each global subtask as either *retriable*, *compensatable*, or *pivot*. We say that a subtask is *retriable* if it is guaranteed to commit after a finite number of submissions when executed from any consistent database state. A subtask is *compensatable* if the effects of its execution can be semantically undone after commitment by executing a compensating subtask. A compensating subtask ct_i for a subtask t_i must be independent of the tasks that execute between t_i and ct_i. This is because local database autonomy requires that arbitrary local tasks be executable between the time t_i is committed and the time ct_i is executed, and these local tasks can both see and overwrite the effects of t_i during that time. A subtask is a *pivot* subtask if it is neither retriable nor compensatable.

Our investigation starts with the construction of global tasks whose flex-atomicity can be maintained in the SDE environment. In general, the flex-atomicity of a global task may not be ensured without using local prepare-to-commit states. For example, the possession of two or more pivot subtasks that must be executed in a global task may render it difficult to determine a commit order among them which ensures that the global task can move either forward to the commitment of its subtasks or backward to the removal of any partial effects of the committed subtasks. However, if one pivot subtask can be replaced by some other retriable subtask, then the global task may still proceed when the pivot subtask fails. The establishment of the essential properties of global tasks such that their flex-atomicity can be ensured provides a foundation for the design of an advanced global task language. We are also investigating the decentralized commit protocol for ensuring the flex-atomicity of such global tasks [ZNBB94].

4 A Method for Global Concurrency Control

In this section, we shall study the global concurrency control of the execution of both local and global tasks while compensation is used to preserve flex-atomicity. In such an environment, a task may see the partial effect of another task before these partial effects are compensated. As a result, this task may see an inconsistent database state. Thus, the proposal of the compensation technique for preserving flex-atomicity mandates a careful examination of the effect of compensation on the concurrency control of global tasks. A correctness criterion is proposed for ensuring the consistency of the SDE.

In [KLS90], a formal analysis is presented of those situations in which a transaction may see the partial effect of another transaction before these partial effects are compensated. It is then proposed in [LKS91] that, to prevent an inconsistent database state from being seen in a distributed database environment, a global task should be unaffected by both aborted and committed subtasks of another

global task. This theory is termed *isolation of recovery.* A concurrency control correctness criterion, termed *serializability with respect to compensation (SRC)*, is further proposed in [MRKS92] to preserve database consistency in the MDBS environment throughout the execution of global tasks possessing no value dependencies among their subtasks. This criterion prohibits any global task that is serialized between a global task G_i and its compensating transaction CG_i from accessing the local sites at which G_i aborts. However, even if SRC is ensured in a global schedule S and compensating transactions undo the effects of compensated-for global subtasks in S, S may still not preserve multidatabase consistency. The following example illustrates the situation:

Example 1 *Consider an MDBS that has data item a, b at LS_1 and data item c at LS_2. Let the integrity constraints be $a > c$ and $b > c$. Let two global tasks G_1 and G_2 be:*

$G_{11} : r(b)w(b, b-1), \quad G_{12} : r(c)w(c, c-1).$

$G_{21} : r(b)w(a, b).$

Consider global task G_1 that results from database state $a = 1, b = 1, c = 0$, where G_{11} commits, G_{12} aborts, and a global task G_2 executes after G_1. A compensable global task $CG_1 : r(b)w(b+1)$, which is independent of G_2, undoes the effect of G_{11}. G_1, G_2, and CG_1 are serializable in the order $G_1 \rightarrow G_2 \rightarrow CG_1$. There are no data dependencies between subtasks of G_1. The global schedule is SRC. However, the resulting database state, which is $a = 0, b = 1, c = 0$, is inconsistent. □

We realize[1] that inconsistencies such as those illustrated in Example 1 arise because the compensating transaction fails to restore database consistency after it executes. In order to avoid such situations, the compensating transaction must also undo any effects that may have been seen by other global tasks. Compensating transactions must therefore be dynamically constructed to take account any executions that have occurred after the commitment of the global subtasks. Such considerations greatly complicate the task of constructing compensation transactions.

In addition to the situation illustrated above, the SRC criterion is not applicable in instances in which value dependencies are defined on global tasks. The following example is illustrative:

Example 2 *Consider an SDE that has data item a at LS_1, data item b at LS_2, and data item c at LS_3. Let the integrity constraints be $a < c$, $b < c$, and $a = b$. Let a global task G_1 consist of two subtasks:*

$G_{11} : r(a)w(a, a-1), \quad G_{12} : r(b)w(b, b-1).$

Let another global task G_2 be:

$G_{21} : r(a), \quad G_{23} : w(c, a+1).$

Consider an execution of G_1 that results from database state $a = 3, b = 3, c = 5$, where G_{11} commits while G_{12} aborts and G_2 executes after G_1. A compensatable transaction $CG_1 : r(a)w(a, a+1)$,

[1] As suggested by discussions with Dr. Sharad Mehrotra.

which is independent of G_2, then undoes the effect of G_{11}. G_1, G_2, and CG_1 are serializable in the order $G_1 \rightarrow G_2 \rightarrow CG_1$. G_2 does not access the local site where G_1 aborts. However, the resulting database state, which is $a = 3, b = 3, c = 3$, is obviously inconsistent. Note that G_{23} is value dependent on G_{21}. □

Thus, the existing approaches are inadequate to a situation in which value dependencies are present among the subtasks of a global task. Value dependencies, which specify data flow among the global subtasks of each global task, are characteristics of many applications. For example, many applications involve data transfer among different local database sites, generating value dependencies among the subtasks of a global transaction.

We propose a new correctness criterion for the execution of local and global tasks while considering the effects of compensation. In the proposed correctness criterion, serializability is ensured among local tasks, global tasks, and compensating transactions. In addition, the partial effects of the committed subtasks of a global task will not be seen by other global tasks until either the entire global task commits or the partial effects are compensated. Our primary concern is to guarantee multidatabase consistency while still achieving high concurrency in the execution of local and global tasks [BZ94].

4.1 Compensation Serializability

We assume that local tasks preserve only local integrity constraints, while global tasks preserve both local and global integrity constraints. We further assume that no global integrity constraints may be placed on those data items that are updatable by local tasks. Otherwise, such updating may result in the violation of global integrity constraints.

We define a global subtask in global schedule S to be *compensated-for* if it has committed in S and its effects need to be compensated. A global task G_i in global schedule S is *compensated-for* if it has compensated-for global subtasks in S.

When a global subtask commits, the need for compensation has not yet been determined. If this subtask is eventually compensated, its results form the partial effects of a global task that may not be globally consistent. Clearly, local tasks can see such partial effects of a global task, because the execution of a global subtask always preserves local database consistency. As we have discussed earlier, whether other global tasks should be allowed to see such partial effects of a global task is less immediately apparent.

We shall now explore an alternative approach which prevents the partial effects of a global task from being seen by other global tasks before its compensating transaction is executed.

Let $AC(G)$ denote the set of data items that G accesses and commits, and let $WC(G)$ denote the set of data items that G writes and commits. Suppose G_i is a compensated-for global task.

Following the stipulation regarding the independence of CG_i, we see that any write operations of other global tasks can be executed between G_i and CG_i, as long as the local concurrency control criteria are followed. However, the read operations of global tasks must be carefully scheduled to ensure that the partial effects of a global transaction will not be read to other global tasks before it commits. A concurrency control correctness criterion, termed *compensation serializability*, is defined as follows:

Definition 1 *(Compensation serializability) A global schedule S is compensation serializable if S is serializable and, for any global task G_j which is serialized between a compensated-for global task G_i and its compensating transaction CG_i in S, $WC(G_i) \cap AC(G_j) = \emptyset$.*

Thus, in a compensation serializable global schedule, any partial effects of a compensated-for global task that are not globally consistent will remain unseen by other global tasks. In addition, since the execution of a global subtask always results in a consistent local database state, a local task therefore always sees a consistent local database state. As a result, all local tasks in S see consistent local database states, and all global transactions see consistent multidatabase states. We have the following straightforward lemma:

Lemma 1 *Every local (or global) task in a compensation serializable global schedule sees a consistent multidatabase state.*

We claim that a compensation serializable global schedule S always results in a consistent global database state. This is stated and proven succinctly in the following theorem:

Theorem 1 *A global schedule S that is compensation serializable preserves multidatabase consistency.*

Proof: Since S is serializable, we assume that S is conflict equivalent to a serial schedule S' [BHG87]. By the semantics of compensation, the partial effects of compensated-for subtasks in S' are semantically compensated by their compensating transactions. Since no effects of compensated-for subtasks are seen by other global subtasks before they are compensated, any inconsistencies caused by these compensated-for subtasks are restored by their compensating transactions. Let S'' be S' restricted to those transactions that are neither compensated-for subtasks nor their compensating transactions. Thus, S'' consists only of atomic local and global tasks [BHG87]. If each transaction in S'' sees a consistent database state, then S'' preserves the multidatabase consistency. Since all local tasks or global subtasks at each local site in S'' either commit or abort, every local task sees a consistent local database state. Following Lemma 1, every global transaction also sees a consistent multidatabase state. Hence, S'' preserves the multidatabase consistency. □

Following Lemma 1 and Theorem 1, we have the following corollary:

Corollary 1 *A compensation serializable global schedule is correct.*

4.2 Maintaining Compensation Serializability

In this section, we present two GTM scheduling protocols, which ensure compensating serializability on the execution of local and global tasks. We say that a global task is *robustly terminated* if its commit or abort status has been determined.

4.2.1 A Centralized Approach

We now propose a revised transaction-site graph approach to enforcing compensation serializability. For simplicity, we will incorporate the *ticket* method [GRS91] in our approach. With this method, each global task must update the ticket data item at the local site it accesses. Consequently, every global task conflicts with every other global task at each local site where both have subtasks. Serializability on a global schedule can then be ensured if the local schedules are serializable and the serialization graph of its global subschedule is acyclic [GRS91]. Both conservative and non-conservative approaches have been proposed to maintain an acyclic serialization graph of a global subschedule. The conservative approaches, such as site graph [BS88] and serialization events [ED90, MRB+92, Pu88], can avoid a large number of transaction aborts[2] but may provide a low degree of concurrency and involve a high overhead. The non-conservative approaches, such as optimistic ticket method [GRS91], provides a high degree of concurrency but subject to a high percentage of transaction aborts, which may be too expensive. In addition, as pointed out in [HHS93], the non-conservative approaches may severely degrade local task throughput and thus may be undesirable in the SDE environment.

We propose a compromise approach, which effectively combines the ticket method, the conservative serialization graph testing [BHG87], and the site graph to avoid the high overhead on maintaining serialization events, to prevent transaction aborts, and to make the approach fault-tolerant. The conservative serialization graph testing maintains a stored serialization graph (SSG) among global transactions for scheduling purposes. In order to describe the commitment status of global subtasks at local sites, we modify the concept of SSG by adding site nodes and their incident edges, as follows:

Definition 2 *(Stored Transaction-site Graph) The stored transaction-site graph of the execution of a set of global transactions in global schedule S, denoted STG, is a directed graph whose nodes are the global transactions (transaction nodes) and local sites (site nodes) and whose edges are all $G_i \rightarrow G_j (i \neq j)$ and $G_i \rightarrow LS_j$ such that*

[2]Due to non-serializability and deadlocks.

- $G_i \rightarrow G_j$ *if an operation of* G_{ik} *precedes and conflicts with an operation of* G_{jk}, *for* $k = 1, ..., n$.
- $G_i \rightarrow LS_j$ *if* G_i *accesses* LS_j.

If the GTM receives an $< ack_commit, G_{ij} >$ message from LS_j, then edge $G_i \rightarrow LS_j$ is referred to as a committed edge. If the GTM receives an $< ack_abort, G_{ij} >$ message from LS_j, then edge $G_i \rightarrow LS_j$ is referred to as an aborted edge. If the GTM has received no acknowledgement of a commit or abort of G_{ij} from LS_j, then edge $G_i \rightarrow LS_j$ is referred to as a unmarked edge.

We assume that each global task predeclares its read operation set and write operation set. We say that a global task G_j *accesses undeterminedly from* G_i in an STG if (1) $G_i \rightarrow G_j$; (2) there is a local site LS_k such that $w(x)$ and $op(x)$ are operations of G_{ik} and G_{jk}, respectively, and $G_i \rightarrow LS_k$ is not an "aborted" edge; and (3) if there is some G_l such that $G_i \rightarrow G_l$, $G_l \rightarrow G_j$, and $w(x)$ is an operation of G_l, then $G_l \rightarrow LS_k$ is an "aborted" edge.

Let S be a global schedule and $\mathcal{G} = \{G_1, ..., G_n\}$ be a set of global tasks in S. We denote $\text{STG}|_{\mathcal{G}}$ as STG restricted only to transaction nodes in the STG. The GTM scheduling protocol includes an edge insertion rule, an edge deletion rule, and an operation submission rule. Edges incident to a transaction node G_i are inserted into or deleted from the STG only if the rules below are followed:

> Edge Insertion Rule: Insertion of $G_i \rightarrow LS_j$ for each local site LS_j that G_i accesses does not result in G_{ij} accessing undeterminedly from any global task which is previously scheduled in the STG; and insertion of $G_k \rightarrow G_i$ for every previously scheduled G_k in the STG that conflicts with G_i does not result in a cycle in $\text{STG}|_{\mathcal{G}}$.

> Edge Deletion Rule: Edges incident on a global transaction are deleted from the STG as soon as the global task has robustly terminated and has no incoming edges in the STG.

The operations of a global task G_i are submitted to LSSs for execution only if the edges of the global task have been successfully inserted into the STG. The operations are submitted to servers based on the following rule:

> Operation submission rule: Each operation is submitted only after all conflicting operations of previously scheduled global transactions have been acknowledged.

Lemma 2 *Consider two conflicting global tasks* G_i *and* G_j *in an STG.* G_i *is serialized before* G_j *in global schedule* S *if and only if the edges of* G_i *are inserted into STG before the edges of* G_j *are inserted into the STG.*

Proof: (*if*) We need to show that, if the edges of G_i are inserted into the STG prior to the edges of G_j, then G_i is serialized before G_j. Suppose G_i is not serialized before G_j in global schedule S. Since G_i conflicts with G_j, there must be conflicting operations op_i of G_i and op_j of G_j such that op_j is executed before op_i in S. Hence, there is an edge $G_j \rightarrow G_i$ in the STG, which contradicts the assumption.

(*only if*) Conversely, we need to show that, if G_i is serialized before G_j, then the edges of G_i are inserted into the STG prior to the edges of G_j. Suppose the edges of G_j are inserted into the STG before the edges of G_i. Since G_i conflicts with G_j, there must be an edge $G_j \rightarrow G_i$ in the STG. By the operation submission rule, there are conflicting operations op_i of G_i and op_j of G_j such that op_j is executed before op_i in global schedule S. Hence, G_j must be serialized before G_i in S, which contradicts the assumption. □

Theorem 2 *If the submissions of global tasks follow the GTM scheduling protocol, then the serializability of local schedules implies the compensation serializability of global schedules.*

Proof: Clearly, the edge insertion and deletion rules generate an STG which is more restrictive than the SSG. Thus, the global subschedule is serializable. In addition, since each global task conflicts with all other global tasks at each local site where they both have subtasks, the serialization order of global tasks at all local sites is relatively synchronized.[3] Following the discussion in [GRS91, MRB+92, ZE93], the global schedule is serializable.

We now show that, for every global task G_j in global schedule S, if a compensated-for global task G_i is serialized before G_j in S and $WC(G_i) \cap AC(G_j) \neq \emptyset$, then CG_i is not serialized after G_j. Since G_i conflicts with G_j at a local site, for example, LS_k, and G_i is serialized before G_j, then by lemma 2, $G_i \rightarrow LS_k$ must be inserted into the STG before $G_j \rightarrow LS_k$ is inserted. By the edge insertion rule, $G_j \rightarrow LS_k$ can be inserted into the STG only if $G_i \rightarrow LS_k$ is deleted from the STG, or it is an aborted edge. Since $G_i \rightarrow LS_k$ is a committed edge, due to the edge deletion rule, CG_i must commit before the edges of G_j are inserted into the STG. Since CG_i conflicts with G_j, then by Lemma 2, CG_i is not serialized after G_j. □

We have thus presented a method of graph testing which integrates serialization graph testing with the use of transaction-site graph.

4.2.2 A Decentralized Approach

The principal issue in the implementation of a decentralized global transaction management scheme based upon the proposed theory is the synchronization of the relative serialization orders (RSOs) of

[3] For any two global transactions G_i and G_j in $\mathcal{G}$, the serialization orders of all global subtasks of G_i either precede or follow the serialization orders of all global subtasks of G_j at local sites.

global subtasks at all local sites. Our method begins by numbering all GTM servers in an order O with each GTM server maintaining a site-lock. Prior to executing global task G_i, GTM interpreter GI_i must first request all necessary site-locks from the relevant GTM servers in an order consistent with O. The RSO of G_i is determined at all relevant sites only when GI_i has acquired the necessary site-locks. After the RSO of G_i is determined, GI_i releases all held site-locks. During this process, if failures occur, GI_i will request all relevant GTM servers to remove G_i from the pre-determined RSOs and release all held site-locks. Because the site-locks are requested in an order consistent with O and the RSO of G_i is determined only after GI_i holds all necessary site-locks, the correct synchronization of concurrent site-locks request is ensured and correct RSOs of global tasks at all sites are thus guaranteed. After the RSO of G_i is determined, GI_i sends subtasks of G_i to the relevant GTM servers. Using the extra operation method, these GTM servers enforce the chain-conflicting relationships and submit the subtasks for execution according to the pre-determined RSO.

Another crucial issue is avoiding cascading aborts. Maintenance of synchronized commitment orders for global subtasks at all local sites renders them vulnerable to such aborts. Unless global subtasks are executed serially at each site, aborting one subtask may cause abortion of further global subtasks in an attempt to guarantee a synchronized commitment order of global tasks at all local sites. We here propose a *greedy locking method* to prevent cascading aborts which may arise from the concurrent execution of global tasks. This method requires that each GTM server GS_j maintain a dynamic data-lock table which is initially empty. Each entry in the table represents a data-lock for a data item that is currently accessed by global subtasks. This table is maintained according to the following rules:

- The data-lock requests for each data item are queued and granted in a first-in-first-out manner consistent with the RSO of global subtasks.
- A global subtask can request a sharing data-lock for a data item which it only reads, otherwise, an exclusive data-lock for that data item must be requested.
- A sharing data-lock request for a data item is granted only if it has no data-lock established or if all its existing data-locks are sharing.
- An exclusive data-lock request for a data item is granted only if none of its data-locks exists. This request may be satisfied with a semi-exclusive data-lock if the data item has only sharing data-locks; after all existing sharing data-locks for the data item are released, an exclusive data-lock is granted.
- All the data-locks needed by a global subtask must be requested before the execution of the subtask. The data-locks held by a global subtask are released only when one of the following conditions is satisfied:

- The subtask is retriable and has committed.
- The subtask is pivot and has either committed or aborted.
- The subtask is compensatable and the corresponding global transaction is robustly terminated.

The released data-locks will be granted accordingly to the relevant subtasks.

A semi-exclusive lock has the effect of a sharing lock with regard to the lock holder and the effect of exclusive lock with regard to other global subtasks. That is, read operations on a data item of the subtask holding a semi-exclusive lock may be executed, while any upcoming sharing requests for the data are blocked as if the subtask held an exclusive lock on the data. Semi-exclusive locks are designed particularly to allow a high degree of concurrency in the preservation of the pre-determined RSO.

A global subtask can be executed with currently available data-locks, while an operation of the global subtask can be submitted for execution only when the corresponding data-lock has been granted. In this way, the read operations of global subtasks at each local site are executed concurrently to the greatest possible extent. In addition, since each global subtask releases its data-locks after the entire global task is robustly terminated, the resubmitted aborted subtask or the compensating subtask of the subtask can re-use its held data-locks. Moreover, because the global subtasks that follow an aborted subtask G_{ij} in the pre-determined RSO are blocked by G_{ij} through their conflicting operations, the corresponding LSS cannot decide the serializability orders of these global subtasks relative to G_{ij} prior to the commitment of G_{ij}. The aborting of G_{ij} does not therefore trigger the aborting of any additional global subtask and the pre-determined RSO is still preserved. Thus, the greedy locking method prevents cascading aborts while permitting a high degree of concurrency. the following example illustrates the implementation of this method.

Example 3 *Consider three retriable global subtasks submitted to GTM server GS_j for execution on the local database system LSS_j. G_{1j}, G_{2j}, and G_{3j} access data defined as $\{R(\mathrm{a, x, w}), W(\mathrm{z})\}$[4], $\{R(\mathrm{x, w}), W(\mathrm{a, b, y})\}$, and $\{R(\mathrm{a, x}), W(\mathrm{c, d, y, z})\}$, respectively. Assume that the pre-determined RSO is $G_{1j} \rightarrow G_{2j} \rightarrow G_{3j}$. When G_{1j} is executed, the data-lock table is empty, it therefore holds all the necessary data-locks; in this case, sharing data-locks for* a, x, *and* w, *and an exclusive data-lock for* z. *G_{2j} is then submitted and is granted exclusive data-locks for* b *and* y, *sharing data-locks for* x *and* w, *and a semi-exclusive data-lock for* a. *If the execution involves a $w(\mathrm{a})$ operation, G_{2j} will be blocked until the robust termination of G_1; G_{2j}, however, can be proceeded to execute operations $r(\mathrm{a})$, $r(\mathrm{x})$, and $r(\mathrm{w})$ simultaneously with G_{1j}. By the same token, while G_{3j} can be submitted for*

[4] R(...) consists of read-only data, while W(...) consists of other varieties of data.

execution with a sharing data-lock for x *and exclusive data-locks for* c *and* d*, its execution will be blocked either by* z *until* G_{1j} *is committed or by* a *or* y *until* G_{2j} *is committed.*

In this example, we see that both G_{1j} *and* G_{3j} *read* a*, and* G_{2j} *may read/write* a*. Under the terms of a semi-exclusive lock, the reading of* a *is shared by* G_{1j} *and* G_{2j}*, while* G_{2j} *is blocked by its* $w(\mathrm{a})$ *operation and* G_{3j} *is blocked by its* $r(\mathrm{a})$ *and* $w(\mathrm{a})$ *operations. As all three subtasks hold a sharing data-lock for* x*, the* $r(\mathrm{x})$ *operations in the three subtasks can all be performed simultaneously. A high degree of concurrency is thus achieved.*

Assume that G_{1j} *is aborted by* LSS_j*. Because* G_{2j} *is blocked on the first operation that may conflict with an operation of* G_{1j} *(e.g.,* $w(\mathrm{a})$*),* LSS_j *cannot yet arrange the RSO of* G_{1j} *and* G_{2j}*. A similar situation may also arise regarding the RSO of* G_{1j} *and* G_{3j}*. Cascading aborts that might be caused by the aborting of* G_{1j} *are thus avoided.* G_{1j} *can therefore be resubmitted for execution with its data-locks and the pre-determined RSO can still be enforced. After* G_{1j} *is committed, its data-locks are released, allowing* G_{2j} *to proceed, its serializability order relative to* G_{1j} *is then determined by* LSS_j*.* □

Our decentralized global task management algorithm incorporates the approaches of the decentralized concurrency control and atomic commitment proposed above. The execution of a global task G_i consists of three phases:

Phase 1: Determination of relative serializability orders

This phase determines the relative serializability orders (RSOs) of global tasks at local sites, which are activated when the GTM interpreter processes responsible for executing global tasks submit global subtasks to the GTM servers for execution. Each GTM server maintains a site-lock. Let O be an order on all GTM servers. GTM interpreter GI_i, which executes G_i, must request the necessary site-locks from the relevant GTM servers in an order consistent with O before submitting the subtasks of G_i. Site-locks are allocated according to the following rules:

- All site-lock requests received by a GTM server GS_j which is associated with site LS_j are handled in a first-in-first-out fashion; GS_j can process and grant a site-lock request only when its site-lock is available.

- GI_i must be blocked when its current requested site-lock is not available. GI_i submits all its global subtasks accessing LS_j to GS_j when the site-lock is granted from GS_j. GI_i can then send the next site-lock request to the relevant GTM server.

- GI_i releases all held site-locks after all its global subtasks are submitted and cannot request any further site-locks.

At each site, the RSOs of the global subtasks of different global tasks are determined by their site-lock granting orders, while the RSOs of the global subtasks of a global task at a local site are determined by the semantics of the global subtasks. This method of ordering is deadlock-free and totally distributed.

Phase 2: Execution of global subtasks

The execution of global subtasks at each GTM server is invoked by the pre-determined RSO. To implement the *extra operation method* described, before it is invoked, each global subtask G_{ij} has operations $r(x)w(x)$ been inserted directly before its commit operation. Here x is a data item accessed by the global subtask immediately preceding G_{ij} in the pre-determined RSO, if such a global subtask exists. The execution of G_{ij} is carried out by GTM server process SP_{ij} created by GTM server GS_j and must obey both the rules of the *greedy locking method* described for the request of data-locks and the following additional stipulations:

- An operation of G_{ij} is submitted for execution when the corresponding data-lock is granted.
- When an operation is completed, SP_{ij} sends the result to GTM interpreter GI_i, if necessary; GI_i, in turn, sends the result to the GTM servers associated with value-dependency-related global subtasks. If the data for an operation is unavailable, the execution is blocked by the data.
- When SP_{ij} reaches a commit operation, it sends a commit request to both GS_j and GI_i. SP_{ij} can commit G_{ij} only upon receiving approval from both GS_j and GI_i.
- When SP_{ij} executes an abort operation, it reports the abort to GS_j.

Phase 3: Commitment control of global subtasks at a local site

GTM server GS_j approves the commitment of G_{ij} only when all global subtasks that precede G_{ij} in the pre-determined RSO have committed.

This algorithm allows transaction management decisions concerning a global task G_i to be made independently by the individual GTM servers that execute the subtasks of G_i and by the GTM interpreter that executes G_i, based on locally available or coordinating information. This algorithm is therefore fully decentralized, in that each global task can run independently, requiring no knowledge of other global tasks. The GTM can then be distributed among the machines from which global tasks are issued, resulting in an approach which is both flexible and reliable.

5 Conclusions

In this paper, we have discussed those issues that arise when the techniques of multidatabase transaction management are applied to the software development environment. Our investigation represents a first step toward global task management in SDEs, one which may be amplified in the future by additional refinement.

References

[BCD+93] Omran A. Bukhres, Jiansan Chen, Weimin Du, Ahmed K. Elmagarmid, and Robert Pezzoli. InterBase: An Execution Environment for Heterogeneous Software Systems. *IEEE Computer*, 26(8):57–69, August 1993.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases Systems.* Addison-Wesley Publishing Co., 1987.

[BS88] Y. Breitbart and A. Silberschatz. Multidatabase Update Issues. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 135–142, June 1988.

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

[BSW88] C. Beeri, H. Schek, and G. Weikum. Multi-level transaction management, theoretical art or practical need? In *Proceedings of the International Conference on Extending Database Technology*, pages 134–154, March 1988.

[BZ94] Bharat Bhargava and Aidong Zhang. Scheduling with Compensation in Multidatabase Systems. In *Proceedings of the Third International Conference on System Integration*, San Paulo, Brazil, 1994.

[DE89] W. Du and A. Elmagarmid. Quasi Serializability: A Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 347–355, Amsterdam, The Netherlands, August 1989.

[ED90] A. Elmagarmid and W. Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, California, February 1990.

[GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GMK88] H. Garcia-Molina and B. Kogan. Node Autonomy in Distributed Systems. In *Proceedings of the First International Symposium on Databases for Parallel and Distributed Systems*, pages 158–166, Austin, Texas, December 1988.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM Conference on Management of Data*, pages 249–259, May 1987.

[Gra81] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the International Conference on Very Large Data Bases*, pages 144–154, Cannes, France, September 1981.

[GRS91] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. On Serializability of Multidatabase Transactions Through Forced Local Conflicts. In *Proceedings of the 7th Intl. Conf. on Data Engineering*, pages 314–323, Kobe, Japan, April 1991.

[HHS93] Jiandong Huang, San-Yih Hwang, and Jaideep Srivastava. Concurrency control in federated database systems: A performance study. Technical report, University of Minnesota, Department of Computer Science, 1993.

[HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4), July 1983.

[KLS90] H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[LKS91] E. Levy, H. Korth, and A. Silberschatz. A Theory of Relaxed Atomicity. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1991.

[Mos81] J. E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1981.

[MRB+92] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. The Concurrency Control Problem in Multidatabases: Characteristics and Solutions. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 288–297, 1992.

[MRKS92] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. A transaction model for multidatabase systems. In *Proceedings of International Conference on Distributed Computing Systems*, June 1992.

[ÖV91] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Inc., 1991.

[Pu88] C. Pu. Superdatabases for Composition of Heterogeneous Databases. In *Proceedings of the International Conference on Data Engineering*, pages 548–555, February 1988.

[SKS91] Nandit Soparkar, Henry F. Korth, and Abraham Siberschatz. Failure-resilient transaction management in multidatabases. *IEEE Computer*, 24(12):28–36, December 1991.

[TS93] P. Tarr and S.M. Sutton. Programming heterogeneous transactions for software development environments. In *Proceedings of Fifteenth International conference on Software Engineering*, pages 358–369, 1993.

[Vei90] J. Veijalainen. *Transaction Concepts in Autonomous Database Environments*. R. Oldenbourg Verlag, Germany, 1990.

[ZE93] Aidong Zhang and Ahmed Elmagarmid. A theory of global concurrency control in multidatabase systems. *The VLDB Journal*, 2(3):331–359, July 1993.

[ZNBB94] Aidong Zhang, Marian Nodine, Bharat Bhargava, and Omran Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, May 1994.