The Whole Transaction Model for Heterogeneous Multidatabases

Marian H. Nodine and Patrice A. Tegan

Technical Report No. CS–91–63
November, 1991

The Whole Transaction Model for Heterogeneous Multidatabases

Marian H. Nodine and Patrice A. Tegan
Brown University
Providence, RI 02912

Department of Computer Science
Technical Report CS-91-63
November 15, 1991

Abstract

A heterogeneous multidatabase allows a user to manipulate data contained in a collection of individual, autonomous databases managed by different database management systems. In this paper, we introduce the *whole transaction model* for multidatabases. This model is distinguished from other models in that it does not require the global transactions on the multidatabase to be atomic. Rather, the user specifies the atomic steps required to do the multidatabase transaction, but different multidatabase transactions can interleave their steps arbitrarily.

This paper presents a method for defining global transactions in the whole transaction model, and discusses concurrency control and related issues. We also discuss an implementation of this model which includes a language for defining transactions in the heterogeneous multidatabase, and a design for a database server which executes these transactions and enforces concurrency control and compensation-based recovery.

1 Introduction

A heterogeneous multidatabase allows a user to manipulate data contained in a collection of individual, autonomous databases in which the data is not only distributed geographically but also managed by different database management systems.

A heterogeneous database consists of two logical levels. The *local* level is comprised of autonomous databases, hereby referred to as local databases, that access data through *global subtransactions*. We assume that each local database makes its own guarantees of consistency, persistence and correctness. The *global* level accesses and updates data in the set of local databases using *global transactions* via a global interface. These global transactions allow the collection of local databases to cooperate to accomplish some task involving data stored in more than one local database. The *global transaction manager* enforces correctness and persistence guarantees at the global level. We note that such guarantees can be no stronger than those made in the local databases.

We use the following travel agent example, first proposed by Gray [Gra81], to illustrate the use of a heterogeneous multidatabase:

1. Traveler provides travel agent with destination and dates of travel.
2. Travel agent reserves seat(s) on flight(s) with airlines.
3. Travel agent reserves rental car.
4. Travel agent reserves hotel room.
5. Travel agent sends confirmation to Traveler.

The traveler see the travel plans as one (global) transaction of long duration. We can view this transaction as consisting of a series of global subtransactions on the local databases cooperating to make the reservations for the entire trip. The travel agent coordinates the global transaction, and acts as an interface between the different organizations involved in the plans. Note that at least four different local databases are updated: a travel agent database by steps one and five, the airline database by step two, the car rental database by step three, and the hotel database by step four.

In the *whole transaction model*, each step of a global transaction is accomplished by some complete global subtransaction. Thus, the car rental is accomplished by a complete transaction on the car rental database. The global transaction coordinates the global subtransactions. The global transaction also maintains an internal context that it updates based on the responses from the global subtransactions. Within a global transaction, global subtransactions are often dependent on prior global subtransactions; for example the travel agent cannot make hotel and car reservations without first getting the plane reservation, because it needs the final travel dates and destinations. Other global subtransactions are independent and can therefore execute in parallel. For example, once the airline reservation is made the car and hotel reservations can be attempted concurrently.

In the whole transaction model we do not assume that global transactions are atomic, but rather view them as collections of global subtransactions that execute and commit in some constrained order. We specify this order using a variant of the notion of *patterns* as defined in [NSZ90]. The global subtransactions reside in a transaction library. The initiation of a global subtransaction is accomplished by a procedure call. When the global subtransaction completes, the outcome (commit or abort) and the (possibly null) results are returned to the global transaction manager.

Local database management systems are not modified to facilitate the operation of the global transaction manager. Because of the atomicity restriction on the global subtransactions, the effects of a global subtransaction are guaranteed to persist in the local database. As a result of the persistence, a recovery strategy such as compensation must be explored in the event of global transaction aborts. Recall the travel arrangement global transaction. The global transaction is not committed until the traveler has returned from the trip. At anytime prior, the global transaction may be aborted. For example, suppose the traveler decides to cancel the trip before receiving confirmation of the trip arrangements. At this point the global subtransactions have already updated the airline, car and hotel databases. These subtransactions are not at this time aborted, but are compensated for with delete or remove reservation transactions. The code for these compensating transactions resides in the global transaction manager's transaction library.

Because each global subtransaction is actually a piece of code, issues such as how different database management systems and data models communicate with the global transaction manager are not addressed.

We assume that the procedures in the transaction libraries deal with such issues. Instead we focus on higher-level transaction issues such as *synchronization*, *function replication*, and *transaction specification techniques*.

The rest of the paper is organized as follows: In Section 2 we give a summary of the related research. Section 3 describes the global transaction model and architecture in detail. Section 4 defines global transactions and outlines the use patterns to specify global transaction execution and correctness criteria. Recovery issues are discussed in section 5. We describe two languages for defining global transactions in Sections 6. We discuss an implementation in Sections 7, and 8, including data structures, algorithms, and pseudocode. We conclude in Section 9.

2 Related Research

Recently much attention has been given to the problem of managing transactions in heterogeneous multidatabase environments. Some of the earliest work was conducted by Smith et. al. the in Multibase system [SBD+81]. In Multibase access to data in the autonomous local databases is read-only thus there was no study of recovery issues. This work focused mainly on issues of schema mapping and resolving inconsistencies among schemas and data.

Most of the proposed models of transaction management in the heterogeneous multidatabase domain follow traditional transaction management philosophy, requiring atomic updates at the global transaction level and using serializability as a correctness criteria. In his Superdatabases work, [Pu88] takes an optimistic approach to global concurrency control. He requires each local site to provide the serialization order of its transactions to a global manager. The global manager analyzes these orders (O-vectors) to determine which global transactions may not serialize, and commits or aborts the global transactions accordingly. [EH88] and [BST90] are schemes to produce serializable global transactions, but restrict how local data can be accessed. For instance, [BST90] partitions the data in the local databases into a locally-updateable set of data and a globally-updateable set. [Pu88] and others indicate that commitment of global transactions is all but impossible when the local databases have full autonomy in their local commit procedures.

Atomicity at the global level often restricts transaction too severely. As noted by [Gra81], the global transaction for the travel reservation can not really commit until after the traveler has returned from his trip if atomicity and serializability are required at the global level.

[DE89] and [DE90] extend traditional serializability theory, with the notion of quasi-serializability as a correctness criteria for global concurrency control. The idea behind quasi-serializability is that if you restrict global transactions such that there are no data dependencies between global subtransactions initiated by a single global transaction, then correct histories are ones where serializability is enforced among the global transactions and also among the transactions in each local database. [Bar90] makes similar assumptions and conclusions in his work.

Elmagarmid et.al. [ELLR90] offer a model in which the flow of the global transaction is controlled by a predicate petri net. Participating database systems are assumed to be autonomous. This paper addresses issues of function replication and non-vital functions. However, this work still assumes that global transactions execute quasi-serializably. This model also strict dependency assumptions, and does not address recovery well.

Garcia-Molina et.al. [GS87, GGK+90] propose sagas and nested sagas as a way of ameliorating the problems associated with long-duration transactions (though not in the context of heterogeneous multidatabases). Sagas are sequences of short-duration transactions operating in a traditional transaction framework. A saga is atomic and requires serializability of the short-duration transactions. As with the whole transaction model, sagas use compensation as a basic tool in recovery.

Work on compensation in recovery, though not explicitly defined for heterogeneous multidatabases, has been done by Korth et.al. [KLS90]. They have done work on compensation, explicitly examining issues of dependencies among committed transactions and how they affect compensation strategies.

Finally, previous work by Nodine et.al. [NSZ90] deals with the use of more flexible correctness specifications in the context of cooperative transactions. This work on multidatabases takes a similar approach to the pattern-based synchronization and recovery schemes defined for cooperative transactions.

3 The Whole Transaction Model

3.1 Modeling Issues

There are many difficult issues that need to be addressed in heterogeneous multidatabase research. Most of these issues can be classified into *application* issues, that concern the nature of the global data model and global transaction management, and *presentation* issues, that concern how information stored as specified in the local databases can be presented to the users according to some common, global schema.

Application issues mainly concern the nature of the global transactions and their management. This includes how global transactions are defined and execute. In the area of global transaction management, the application issues include how correct execution is defined for a set of global transactions, as well as issues of synchronization and recovery when a global transaction aborts or a process fails. These issues are the ones we address in this paper.

Presentation issues are commonly referred to as *encoding* and *decoding* issues. This is because they concern mapping information presented according to one or more local database schemata to/from the presentation as defined in the global schema; in other words, they are issues of schema translation. Many of the presentation problems that have been addressed in the literature bear strong resemblance to schema evolution problems. These include such issues as dealing with null values and dealing with merging or splitting objects in the database. These issues are known to be very difficult. Thus, we do not address them in this paper, but rather assume that encoding or decoding is accomplished explicitly in hand-coded subroutines. These subroutines retrieve explicit information from some local database, and make specific internal decisions about how to resolve any presentation problems.

3.2 Underlying Assumptions

The whole transaction model makes several assumptions both about the local databases that comprise the multidatabase and the nature of the transactions that are supported at the global level.

The assumptions we make about local databases are meant to maximize the amount of autonomy they have. Basic definitions of autonomy are specified in [DE89]. *Design autonomy* concerns the assumptions the global transaction manager makes about the data models and transaction management strategies of the local databases. We assume that all global subtransactions are atomic, serializable, and recoverable. We make no assumptions about the underlying data models at all; thus, the local databases may be *heterogeneous* in terms of the data models they support. *Communication autonomy* concerns the assumptions the global database makes about interactions among the local databases. We do not assume that the local databases can communicate with each other; they need not even know about each others' existence. Thus, we assume full communication autonomy. We also make no assumptions concerning the *distribution* of the local databases. *Execution autonomy* concerns any assumptions on how the local databases execute their transactions. One assumption we make with respect to execution autonomy is that the local databases can execute any transactions, including both global subtransactions generated by the global transaction manager and *independent transactions* initiated by other agents.

We view the global database as a set of local databases cooperating to execute global transactions in a structured manner. Thus, the *global database* contains at most the set of information contained in the collection of local databases forming the heterogeneous multidatabase.

Unlike most multidatabases, we do not assume that global transactions are atomic. Rather, we view a global transaction as defining how a set of global subtransactions can *cooperate* to do some task that spans multiple local databases. This is similar to our earlier work on *cooperative transactions* [NSZ90]. However, we do assume that global transactions are defined by sequences of operations that share state and work together to accomplish a specific task. The definition for correctness of a global transaction is a topic for discussion in this paper.

Because the global transactions are not atomic, they may interact with each other through the sharing of data. Also, the global transactions may interact with independent transactions in the local databases, in that the independent transactions may affect parts of the local database that are currently being accessed by the global transactions. In this model, we assume that these interactions are not critical, and can be ignored during normal database operation. This is not ideal; however, any attempts to take these interactions into account during normal operation and/or failure recovery lead to situations where either concurrency among

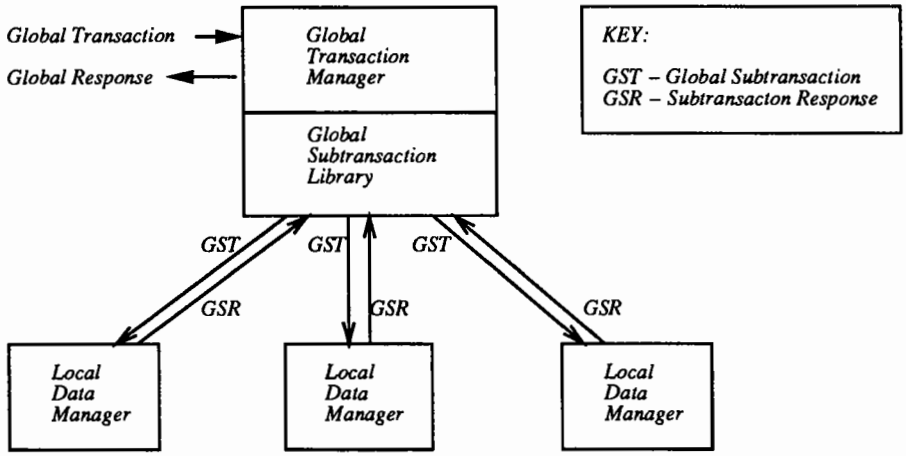

Figure 1: A Multidatabase Architecture Using the Whole Transaction Model.

the global transactions is overly restricted or the design autonomy of the local databases needs to be further restricted to preserve the recoverability of the global transactions. These issues are discussed in detail in Section 5.

Global subtransactions within a single global transaction may also affect each other because the results of one global subtransaction can be placed in a local variable that is later read by some other local transaction. We assume that these dependencies are necessary and must be preserved in a correct history.

3.3 An Architectural Overview

Figure 1 shows a basic architecture that implements the whole transaction model.

All of the global transactions in the heterogeneous multidatabase are submitted to the *global transaction manager*, which handles the standard database functions such as transaction synchronization, deadlock detection, and global database failure and recovery. Global transactions execute by initiating *global subtransactions.*

The *global subtransaction library* has a set of parameterized procedures that can be called to initiate specific transactions in the local database. The library defines all possible global subtransactions that can occur in the database. A library routine interacts with a local database using the local DML. Once the library routine receives the response, it can translate it into the global DML. The result is then returned from the library routine as a return value, along with an indication of whether the global subtransaction committed or aborted.

The *local data managers* are the servers responsible for executing the global subtransactions. All accesses and updates in the heterogeneous multidatabase are ultimately executed by the local data managers. The global data manager requires that the local data managers execute their transactions atomically, and that those transactions be recoverable by the local data manager. However, we assume that the local managers are completely independent of the global data manager in defining their local data model and in managing their data.

3.4 The Global Database

A global database is a time-varying collection of local databases. No information is retrievable from the global database that is not stored in some local database. In other words, the set of information *GDB* that a global database can provide access to is the union of the sets of information available in its current collection of local databases.

$$GDB \subseteq \cup_i LDI_i$$

where LDI_i is the set of information available from local database I. Note that while the global database does not store extra information independent of the local databases, it may choose *not* to provide information

that is stored in some local database.

The *global schema* defines the information available from the global database, and the form in which it is accessed. It is defined according to some *global data model.* Each local database has its own *local schema* defined according to its own *local data model.* The local schema defines the information available from the local database, and the form in which it can be accessed.

3.5 Global Transactions

A *global transaction* does some task, has some internal control structure, and initiates operations on local databases in some partial order. Global transactions access and/or update information in one or more local database.

In the whole transaction model, transactions on the global database begin with a *transaction begin delimiter* and end with a *transaction end delimiter* which may be either a *commit* or an *abort.* Global transactions contain partial orders of atomic, global subtransactions submitted to the local databases. If one local transaction precedes another in the partial order, this means that the earlier transaction commits or aborts before the later transaction begins.

Formally, the execution of a global transaction is a partial order

$$GT = (HE_i, <^i_{GT})$$

where

1. $HE_i = ((\cup_i GST_i) \cup (\cup_j GD_j))$.
2. GST_i is an atomic transaction on a local database initiated by this global transaction.
3. GD_j is a transaction begin or transaction end delimiter for the global transaction.
4. If HE_l is the transaction begin delimiter for the global transaction, then $\forall i \neq l, (HE_l <_{GT} HE_i)$.
5. If HE_m is the transaction end delimiter for the global transaction, then $\forall i \neq m, (HE_i <_{GT} HE_m)$.
6. If GST_i reads information internal to transaction GT that was updated by GST_j, then $(GST_j <^i_{GT} GST_i)$.
7. If GST_i and GST_j access the same local database, then either $(GST_j <^i_{GT} GST_i)$ or $(GST_i <^i_{GT} GST_j)$. The order is defined by the serialization order of the two global subtransactions on the local database.

3.6 Global Subtransactions and Local Histories

A *local data manager* processes *global subtransactions* that are initiated by the global transaction manager, and *independent transactions* that are initiated in other ways. We model a global subtransaction as a partial order of operations, in completion order:

$$GST = (O_k, <^k_{GST})$$

where O_k is an operation in the local data manager's DML. Similarly, we model an independent transaction as an analogous partial order

$$IT = (O_k, <^k_{IT}).$$

We model the activities on the local database in terms of its *local history.* This history is a partial order of operations initiated by the global subtransactions and independent transactions, in the order they were completed. When projected from the local history, the partial order of operations associated with a single global subtransaction or independent transaction T_i must be consistent with the partial order $<^k_{T_i}$. Thus, we have that a local history

$$LH = (O_i, <^i_{LH})$$

over some set of global subtransactions and independent transactions $T = GST \cup IT$ on a single local database has the following properties:

1. O_i is an operation initiated by some transaction $T_j \in T$, specified in the local DML.
2. $((\cup_k <^k_{GST}) \cup (\cup_k <^k_{IT})) \subseteq <^i_{LH}$ is the ordering relation for the total order.
3. For any two conflicting operations $o_1, o_2 \in O$, either $(o_1 <^i_{LH} o_2)$ or $(o_2 <^i_{LH} o_1)$, where conflict is defined by the local data manager.

By definition, the correctness criterion for the local database is serializability. (For a definition of serializability, see [BHG87].)

3.7 The Global History

Let S_{GT} be the set of global transactions submitted to the global transaction manager. The *global history* contains the partial order of global subtransactions initiated by the global transactions in S_{GT}, in the order that the global subtransactions completed (committed or aborted). It also contains the *transaction begin* and *transaction end* delimiters for each global transaction. The partial order represented by the projection of the global subtransactions associated with a single global transaction must be consistent with its own partial order GT. We define a global history GH as follows:

$$GH = (HE_i, <^i_{GH}),$$

where

1. HE_i is the history entry in $((\cup_j GST_j) \cup (\cup_k TD_k))$.
2. GST_j is the set of global subtransactions submitted as a part of some global transaction in S_{GT}.
3. TD_k is the set of transaction delimiters, where for each transaction $GT_j \in S_{GT}$ there is exactly one *transaction begin* delimiter and for each complete transaction $GT_j \in S_{GT}$ there is one *transaction end* delimiter.
4. $(\cup_j <^k_{GT_j}) \subseteq <^i_{GH}$ is the ordering relation for the partial order, where $GT_j \in S_{GT}$.
5. For any two global subtransactions GST_1 and GST_2 on the same local database, if HE_1 is the history entry for GST_1 and HE_2 is the history entry for GST_2, then either $(HE_1 <^i_{GH} HE_2)$ or $(HE_2 <^i_{GH} HE_1)$.
6. If global subtransaction GST_1 reads information internal to the global transaction that was produced by global subtransaction GST_2, and if HE_1 is the history entry for GST_1 and HE_2 is the history entry for GST_2, then $(HE_2 <^i_{GH} HE_1)$.

4 Global Transactions

In the whole transaction model, global transactions span multiple local databases. A global transaction accomplishes some task, and is defined as a *pattern* of *actions* that must occur for that task to successfully complete. An *action* is a unit of work that is accomplished by completing a single global subtransaction on a single database. Since we allow function replication, there may be more than one global subtransaction that can accomplish some specific action. A *pattern* defines the actions that are necessary to accomplish the global transaction's task, and serves as a correctness specification for that task. Patterns may specify alternatives if different sequences may be used to accomplish the task. They may also specify actions for the same global transaction that may occur in parallel.

This definition is similar to the patterns we defined for cooperative transactions [NSZ90]; however it differs significantly in that a pattern tells some global transaction manager how to run the entire transaction. Once the global transaction is fully-specified and sent to the manager, it should be able to initiate all global subtransactions and do all computations necessary for completion of the transaction without additional input from the definer of the global transaction. Another difference is that we do not allow conflicts to be specified within the patterns that define the global transactions. Because of our assumptions about autonomy, we cannot guarantee that a conflict would be enforced in the way the global transaction definer would like. For example, an independent transaction on a local database could execute a conflicting global subtransaction without the knowledge of the global transaction manager.

4.1 Transaction Definition

Global transactions consist of one or more atomic *actions*. These actions must conform to some *pattern* in order to correctly execute the global transaction. The pattern specifies, at any time, what actions may be done next. Actions may also be interspersed with computations relevant to the global transaction, which use information internal to it. A computation may indicate, for instance, which action to choose to do next.

We also allow the user certain flexibility in specifying global transactions. In particular, we will try to address the issues of *function replication* and *non-vital actions*.

Function replication means that we allow the same objective to be accomplished in basically equivalent ways in different databases. There are two ways in which this can occur in a global transaction. One is that an action may specify alternative global subtransactions. For instance, if the action is making a plane reservation, alternative global subtransactions might be to use United airlines or to use Delta airlines. Another way we allow it is to use alternative actions. For instance, the traveler may want to fly to another city and rent a car, or he may want to rent a car at home and drive to the other city.

Non-vital actions do not need to be completed for the global transaction to complete. For example, renting a car in a foreign city may be desirable; however, if no cars are available, taking taxis or buses is a viable alternative. Non-vital actions are desirable, but not necessary. Therefore, a non-vital action will always be attempted, but its failure does not cause the global transaction to fail.

4.1.1 Global Transaction Specifications

The pattern associated with a global transaction is specified as an augmented finite state automaton, called a *global transaction specification (GTS)*. States in the GTS represent states in the execution of the global transaction, and may have associated computations. In concurrent execution, states may be grouped together to form *state sets*, where each state in the set represents the execution state of one thread in the parallel execution. Arcs in the GTS go from state sets to state sets, and represent actions that need to complete for the global transaction to change state. The start state set of the GTS represents the beginning of the pattern for the global transaction, and final state sets represent possible ends of the pattern. We can also view a global transaction as a regular expression in some language, where the terminal symbols in the language are actions.

We can now formally define GTS as follows:

$$GTS =< N, K, \Sigma, \Delta, s, c, F >$$

where

N is the name of the GTS,

K is a finite set of states,

Σ is an alphabet, where $\sigma \in \Sigma$ is the name of some action or NULL,

Δ is a transition function from $2^K \times \Sigma \rightarrow 2^K$,

$s \subseteq K$ is the initial state set,

$c \subseteq K$ is the current state set, and

$F \subseteq 2^K$ is the set of final state sets. The final state sets are disjoint.

4.1.2 Action Specifications

An action is a set of global subtransactions, exactly one of which must succeed for the arc to be traversed. Because the action must be completely specified in the global transaction, and because the user may wish to specify a preferred order in which to try the global subtransactions, the set of global subtransactions in an action is partially ordered.

A global subtransaction is specified within an action as a call to some procedure in the GTM's transaction library. This includes both the name and the required arguments.

In addition to normal actions, there is also the NULL action; arcs labeled with the NULL action are automatically traversed.

4.2 Correctness

In this section, we will discuss correctness issues only for global histories where there has been no failure or abort of a global transaction. Correctness criteria for histories that contain global transactions that have failed or aborted will be discussed in the section on recovery (Section 5). This section also defines correctness only for global histories that are *complete*; that is, all transaction represented in the history have either committed or aborted.

Definition 4.1 *A global history* GH *is* correct *when every global transaction* $\mathrm{GT}_i \in S_{GT}$ *satisfies the* pattern criterion *and the* delimiter criterion*; and every global subtransaction initiated by some* $\mathrm{GT}_i \in S_{GT}$ *satisfies the correctness criterion of the local database on which it was executed.*

- Pattern criterion:
  Let GH_T *be the projection of history entries for* GT_i *from* GH*, excluding the entries containing the transaction begin and transaction end delimiters. Let* $\mathrm{GST}_1 \cdots \mathrm{GST}_n$ *be the partial order of global subtransactions represented by those history entries. Let* GTS_i *be the global transaction specification for* GT_i*. Then* $\mathrm{GST}_1 \cdots \mathrm{GST}_n$ *is acceptable by the global transaction specification* GTS_i *and its associated action specifications.*

- Delimiter Criterion:
  The projection of history entries for GT_i *from* GH*, begins with an entry containing the transaction begin delimiter and ends with an entry containing the transaction end delimiter.*

This effectively states that a global history is correct if each global transaction executes its steps in a correct order. The steps of different global transactions can interleave arbitrarily.

It is desirable for the global transaction manager to be able to determine as it operates whether or not the history being produced leads to a correct history. This means that we need to ensure that patterns are specified in such a way that valid prefixes of correct histories are easily and efficiently recognizable.

5 Recovery Issues

5.1 Basic Issues

We assume in this paper that global transactions are not necessarily atomic. Specifically, once a global subtransaction initiated by some global transaction commits, other global transactions will be able to "see" the effects of that action before the global transaction itself commits.

We also assume in this paper that, if this global transaction later undoes the effects of some global subtransaction, it is not critical that other global transactions that have seen those effects be notified of the changes. For example, if Bob tries to make a reservation on a particular flight, and can't get a seat because the plane is full, he usually tries another flight or places himself on a waiting list. Thus, Bob himself is responsible for explicitly making provisions so that he can get a seat if one opens up later. This assumption is similar to the ones made in *sagas* [GS87, GGK+90].

Because of these assumptions, the best recovery option for the whole transaction model is one based on *compensation.* Compensation is generally used in situations where it is expedient or necessary to commit some of the effects of a transaction before committing the entire transaction. If the transaction is later aborted, the effects are not undone, but rather are *compensated for* by issuing compensating operations to remove the effects that were committed. For example, if a transaction makes a reservation, and the passenger changes his mind, the transaction can compensate by canceling the reservation.

In the whole transaction model, global subtransactions are atomic. Each global transaction must commit before any global subtransaction that depends on it (i.e., follows it in the ordering $<_{GT}$) can begin. Thus, we assume that effects in the local databases commit (and are made visible) before the global transaction commits. However, we also assume that every global subtransaction GST_i has a corresponding compensating subtransaction CST_i that can be run on the same database to undo its effects if the global transaction aborts.

One problem with compensating transactions is that changes can occur in the local database that cause the compensating transaction not to succeed. For instance, consider what happens if some step deposits money in some bank account. Clearly, if other transactions on that local database withdraw enough money from the account, then a compensating withdraw subtransaction would not succeed. In this paper, we assume that such situations need to be remedied manually. However, in future work we explore ways to constrain the operations on the local database so that this situation is prevented.

5.2 Correctness in the Presence of Recovery

Since we are using compensation as a recovery mechanism, the database history will contain records for compensating subtransactions as well as global subtransactions and global transaction begin and end delimiters. Thus, in the presence of compensation-based recovery, a correct global history *GH* can be defined.

Definition 5.1 (Equivalence) *Two histories H_1 and H_2 are* equivalent *if, starting at the same global database state S_a, their executions produce the same resulting global database state S_b.*

Lemma 5.1 (Commutation Lemma) *Let T_1 and T_2 be adjacent entries in the same global history* GH *representing either global subtransactions or compensating subtransactions, with $T_1 <_{GH} T_2$. Then T_2 can backwards commute with T_1 leaving a history equivalent to* GH *if and only if the following conditions hold:*

1. *The two transactions they represent did not affect the same local database.*
2. *If the two entries both represent global subtransactions GST_1 and GST_2 generated by the same global transaction* GT, *then $GST_1 \not<_{GT} GST_2$.*
3. *If the two entries both represent compensating subtransactions CST_1 and CST_2 generated by the same global transaction* GT, *and if GST_1 and GST_2 are the global subtransactions that were compensated for by CST_1 and CST_2, respectively, then $GST_2 \not<_{GT} GST_1$.*

PROOF: Let $I(T_1)$ and $O(T_1)$ be information accessed and written by the transaction represented by entry T_1, respectively. We take a conservative approach that an access can be either a read or a write. We call $I(T_1)$ the *input set* of T_1, and $O(T_1)$ its *output set.* As with T_1, let $I(T_2)$ and $O(T_2)$ be the input and

output sets for T_2. Assume that each global subtransaction or compensating subtransaction behaves the same given the same input set. Then the only way that the backwards commute can affect the state of the database at the end of the history is if one of the global subtransaction or compensating subtransactions (transitively) reads the output of the other global subtransaction or compensating subtransaction.

Define a *path* as some place where the output set of one global subtransaction or compensating subtransaction affects the input set of the other. The two places a global subtransaction or compensating subtransaction can write to are its own local database and the variables internal to its global transaction, so these are the two places where a path can exist. Now, if the two global subtransactions or compensating subtransactions do not access the same local database, no path can exist through the local database, even indirectly. If both are global subtransactions, and one accesses a local variable that the other has written, the value of that variable may affect the outcome of the global transaction. If so, however, they must be ordered by the ordering relation $<_{GH}$ or the outcome of the global transaction would be nondeterministic. Since we do not allow commutation in this case, no path can exist this way. If both are compensating subtransactions, then there is a path between them if and only if there was a path between their corresponding global subtransactions. However, we do not allow such operations to commute either, so no path can exist here either.

Since no paths exist from one transaction to another, the two are independent, and therefore the order in which they are executed cannot affect the final state of the database. □

Lemma 5.2 (History Equivalence Lemma) *A history H is* equivalent *to a global history* GH *if it can be generated by repeatedly commuting entries in* GH *according to the conditions specified in the commutation lemma.*

PROOF: The commutation lemma defines a set of equivalence classes for global histories. H and GH are in the same equivalence class. □

We also need to define new rules for traversing a global transaction specification when compensating subtransactions can be issued. A compensating subtransaction causes an arc traversal in the *inverse* direction from the current state along the arc that was traversed by the corresponding global subtransaction.

Definition 5.2 (Correct History) *A complete global history GH is* correct *when every global transaction $GT_i \in S_{GT}$ satisfies both the* pattern criterion *and the* delimiter criterion*; and all global subtransactions and compensating subtransactions initiated by some $GT_i \in S_{GT}$ satisfy the correctness criterion of the local database on which they were executed.*

- Pattern criterion:

  1. *GH has an equivalent history GH_C where all consecutive sequences in the history containing only aborted subtransactions and their compensating subtransactions have the following grammar:*

     $S ::= GST_i\ CST_i \mid$
     $\quad GST_i\ S\ CST_i$

     where the GST_i's are distinct aborted global subtransactions and CST_i is the compensating subtransaction for GST_i.

  2. *Let $T_1 \cdots T_n$ be the projection of history entries for* GT_i *from GH_C, excluding the entries containing the transaction begin and transaction end delimiters. Let GTS_i be the global transaction specification for GT_i. Then*
     - *If the end transaction delimiter for GT_i was a* transaction_commit*, then $T_1 \cdots T_n$ is accepted by GTS_i according to the modified traversal rules presented earlier.*
     - *If the end transaction delimiter for GT_i was a* transaction_abort*, then traversing by GTS_i according to the modified traversal rules presented earlier with input $T_1 \cdots T_n$ results in the current state set being the same as the start state set.*

- Delimiter Criterion:
  The projection of history entries for GT_i from GH begins with an entry containing the transaction begin delimiter and ends with an entry containing the transaction end delimiter.

Basically, this is the same as saying that if you remove balanced nested sequences of aborted and compensating subtransactions from the equivalent history GH_C, the resulting history is correct according to Definition 4.1.

5.3 Dependency Maintenance for Correct Recovery

For recovery to succeed in restoring the database to a correct state, information must be kept about the different global transactions, and how their global subtransactions may interact. These interactions give us clues as to how the different global subtransactions may depend on each other, so we call it *dependency information.*

The first type of dependency that forms is among the global subtransactions associated with a single global transaction. Because the global subtransactions must execute in an order accepted by the global transaction's GTS, if some global subtransaction is later invalidated, the subtransactions that follow it in the same execution thread of the same global transaction may no longer be ordered correctly. Thus, each global subtransaction is dependent on the previous committed global subtransaction in the same execution thread of the same global transaction, because if the previous one is not ordered correctly in the global transaction, then this one isn't either. Because these dependencies are defined by the pattern expressed in the GTS, we call these *pattern dependencies.*

Because we use compensation-based recovery, dependencies must form among compensating subtransactions and global subtransactions. In particular, each compensating subtransaction is dependent on the global subtransaction it compensates for. Technically, other dependencies form among the compensating subtransactions associated with a single global transaction. However, while the existence of these dependencies may constrain *when* compensating subtransactions are issued during recovery, the dependencies themselves do not influence what is invalidated after an abort. Because they compensate for other operations, they are never invalidated themselves. Therefore, they need not be kept for future use.

The Commutation Lemma (Lemma 5.1) indicates another place where dependencies can form. If a *path* exists from one global subtransaction to another because the two subtransactions access the same information in the same local database, then the two global subtransactions cannot commute in the history. This is because the execution of the earlier global subtransaction may affect the execution of the later global subtransaction because it modified the information in the local database. We call these *information dependencies.*

Because of local database autonomy assumptions, we cannot compute the information dependencies in a history exactly. For instance, if a global subtransaction GST_1 modifies data item x, and then a second global subtransaction GST_2 reads x, GST_2 may or may not be information dependent on GST_1, depending on whether or not some independent transaction modified x in between the two global subtransaction accesses. Also, if GST_1 modifies data item x and GST_3 never accesses x, it seems on the surface that no information dependency should exist. However, say some independent transaction reads x and later writes data item y. If GST_3 later reads y, then GST_3 actually *is* information dependent on GST_1.

While pattern dependencies only exist between global subtransactions initiated by the same global transaction, information dependencies may exist between global subtransactions initiated by different global transactions. However, since dependencies are only recorded for recovery purposes, it is not clear that we need to compute or record all information dependencies. In particular, if a global subtransaction GST_1 has an information dependency on $GST2$, and GST_2 was initiated by some global transaction that has already committed, then GST_2 cannot be aborted and therefore cannot cause GST_1 to be invalidated. Therefore, we only maintain dependencies to global subtransactions whose global transactions have not yet committed.

5.4 Other Approaches

One difficult problem in heterogeneous multidatabases that follow the whole transaction model is that, if you assume that the local databases are fully autonomous, then the local databases can allow things to happen that compromise the consistency of the data at the global level. For example, independent transactions on the local database are never seen by the global transaction manager. It is not clear how the global transaction manager could discover if independent transaction has done an operation that compromises the integrity of the global database.

Heterogeneous multidatabases that constrain the global transactions to be serializable do not have this problem, but this approach to transaction management limits concurrency and has some effects that may be undesirable. Say Bob has bought a plane ticket as a part of a large set of travel arrangements done in the context of a single global transaction. We noted in Section 1 that it may be undesirable for this global transaction to commit until Bob has completed his trip. However, we do not want to hold back on committing the global subtransaction for the plane reservation because that constrains the allowable subsequent operations on the database. For example, if the flight was canceled, it may be impossible to remove that reservation because Bob's global subtransaction still holds a lock on the reservation object. Clearly this is unacceptable.

There are a few modifications we can make to our approach to global transaction management to allow the global databases more latitude in constraining the behavior at the local level while not necessarily enforcing that global transactions be serializable. The first is to allow global transactions to delay the commit of a global subtransaction for as long as it requires a guarantee that the global subtransaction's effects on the database should remain. While this approach would prevent independent transactions on the local database from making undesirable modifications, it does not address the concern from the previous paragraph that this type of constraint may be undesirable. However, we are currently working on ways to delay the commit and to characterize the tradeoffs between committing global subtransactions earlier vs. committing them later.

The second approach is to try to define and apply constraints on the global transactions. This approach assumes that there is some way of decomposing the constraints and enforcing them at the local database level. Whether or not this can be done effectively, and how it affects our autonomy assumptions at the local level is a subject of future research.

(a) (b)

Figure 2: States and state sets. (a) Start state and start state set. (b) Final state and final state set.

6 Global Transaction Specification

6.1 A Visual Language

Each global transaction is defined by a single pattern. The representation of these patterns (i.e., the global transaction specification, or GTS) needs to be adequately expressive for the user of the heterogeneous multidatabase. Some of the requirements are:

1. Each global transaction must be representable as a single GTS.
2. Since patterns are active, the GTS must contain enough information to allow the global transaction manager to execute the transaction without requiring any interaction from other processes or transactions. For example, computations internal to the global transaction need to be associated with the GTS at specific points. These computations may be used to tell the global transaction what actions to try next or to specify parameters for a given action.
3. We support function replication, which allows the same task to be performed equivalently by either one of several actions or an action to be accomplished by one of several global subtransactions. A priority order needs to be presented among these alternative actions or global subtransactions. In the whole transaction model, operations are initiated by the global transaction itself, and the patterns specify the form of the global transaction. Thus, the patterns need to encapsulate all of the information necessary for the transaction manager to attempt alternative actions and global subtransactions in the desired order.
4. To facilitate concurrency, a GTS needs to specify the potential for parallel actions or sets of actions whose execution order is nondeterministic.
5. The system must be able to use the GTS in its synchronization algorithms to recognize prefixes of correct histories incrementally.

A GTS is represented as a labeled graph with an associated set of local variables. A node in the graph corresponds to a state, and may be part of some state set. Each node has information associated with it, including an optional entrance computation to run when the state is first entered, and an optional exit computation which is run when the state is exited.

There are a few types of nodes distinguished in the GTS. The start state set indicates the current states at the beginning of the pattern accepted by the GTS. A final state set indicates a potential end of the pattern accepted by the GTS. A global transaction can only terminate when the current state set is identical to some final state set. The pictorial representation for these nodes is shown in Figure 2.

An arc is a directed *edge* from some (set of) state(s) to some (set of) state(s). Each arc is associated with an atomic, independent action. The simplest kind of arc connects one node to another. It is directed from its *tail* node to its *head* node. A simple arc may be traversed when its tail node is in the set of current states. When it is traversed, the node at its tail is removed from the set of current states, and the node at its head is added. Arcs may also *fork*, or have multiple *heads*. When a fork arc is traversed, each state at its head is added to the set of current states. An arc which *joins* has multiple tails. A join arc may not be traversed until each state in its tail is in the set of current states. All states in the tail are removed from the set of current states when the arc is traversed. Arcs may also both *fork* and *join*. Figure 3 shows how these are represented graphically.

Alternative actions may also be described for a global transaction. These differ from the partial order of steps defined within an action in that two alternative actions from the same state have different functions

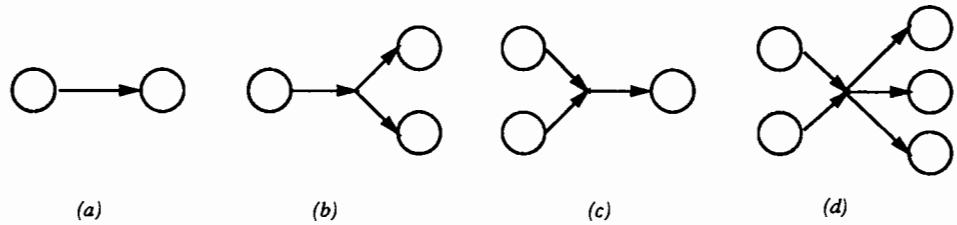

Figure 3: Types of arcs. (a) Simple arc. (b) Fork arc. (c) Join arc. (d) An arc that both forks and joins.

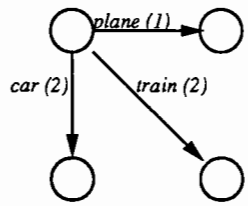

Figure 4: Alternative actions in a global transaction.

and may take the GTS to different state sets. Alternative actions are represented by more than one arc leaving a particular state. Alternatives may be specified with a priority, where the priority is represented by a number on the arc. These numbers are not necessarily unique. Figure 4 shows that there may be more than one way to make the trip. Flying is the alternative of choice, though driving and taking the train are also options to consider.

Local variables associated with a GTS are used to store information from the local databases that is being used internally to this global transaction. For example, a local variable may be used to store the results of some query. These results may be processed to extract information used to direct the computation or to compute values of parameters to be used in queries and updates to other local variables. For instance, once a plane reservation is made, the returned information may be used to update a travel agent's database.

6.2 Nesting of Global Transaction Specifications

In order to facilitate modularity in global transaction definition, a global transaction may specify a *nested global transaction* on an arc in addition to specifying an action. Note that an arc may specify either a nested global transaction or an action, but a nested global transaction may not be an alternative in the partial order of global subtransactions which compose an action. Recursive nesting of global transactions is not allowed. This kind of nesting does not change the power of the transaction that can be expressed with operation machines, since the GTS for any nested global transaction can be substituted directly into the top level GTS.

A nested global transaction is specified as a GTS. Traversing the arc representing a nested global transaction is equivalent to traversing the complete child GTS, starting in the start state set and finishing in some final state set. For the parent GTS to accept an alternative action from the current node, the child GTS must be aborted.

Nested global transactions complete under the same conditions as top-level global transactions. When all the current state tokens are final states in a final state set, the nested global transaction commits and terminates. The completion of a nested global transaction indicates the traversal of the arc associated with the nested transaction in the parent GTS.

Consider the trip reservation example, as sketched in the GTS in Figure 5(a). The process of making a plane reservation may require the travel agent to query a database containing flight information on all airlines as to which flights suit the traveler's needs. This may be based on information provided by the traveler regarding acceptable price ranges and travel times. Once this query finishes the current state of the GTS is actually in the nested global transaction PLANE, as shown in Figure 5(b). The result of this

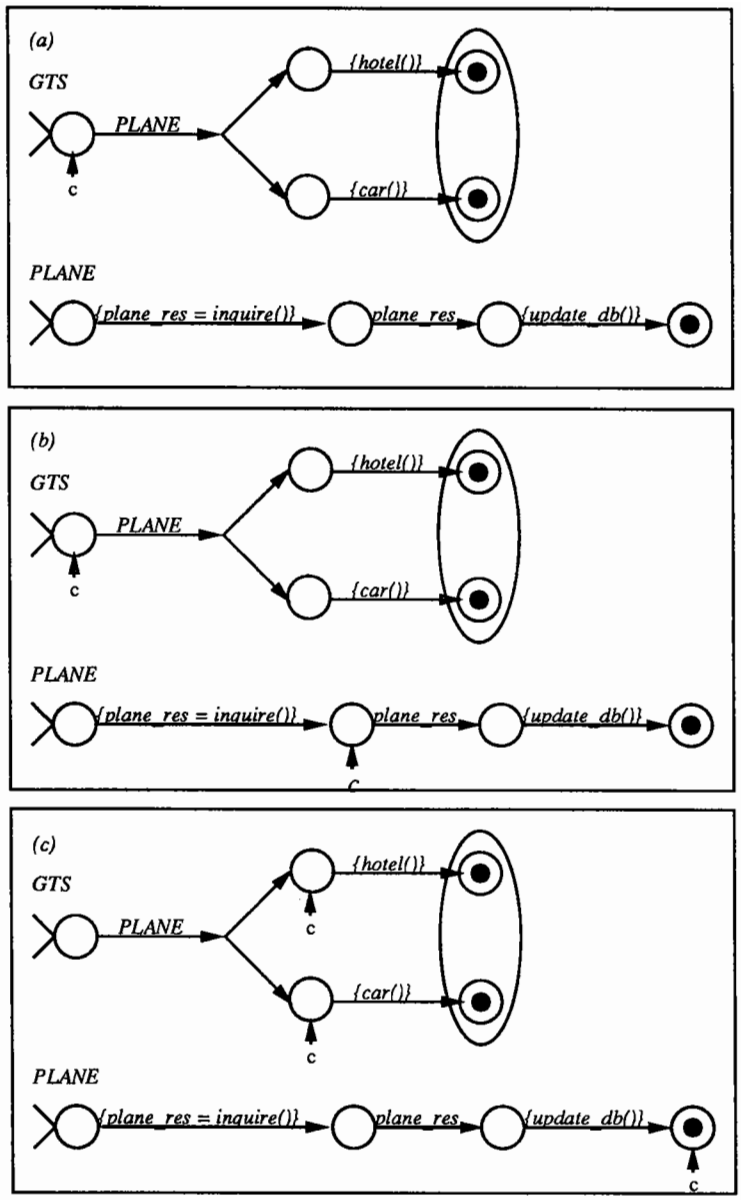

Figure 5: Global Transaction Specification and Execution for the Plane Reservation Example.

inquiry is a partial order of procedure calls for global subtransactions, any one of which would make an airline-specific seat reservation according to the needs of the person taking the trip. This result, which defines an action, is placed in the local variable *plane_res*. Next, the action defined in *plane_res* is invoked, and the seat is reserved by updating one of these databases. Following this procedure the travel agent needs to update the traveler's customer record with the agent transaction to record the specific flight reservation. Upon completion of this update the PLANE nested transaction is complete and the current state tokens are moved to the two middle states in the parent transaction's GTS, as shown in Figure 5(c).

6.3 Equivalence between a GTS and DFSAs

In our work on cooperative transaction hierarchies [NFSZ90], we defined a pattern as an augmented DFSA, and used none of the special features for concurrency and nesting defined in this section. However, for any GTS defined using these extensions, we can show how to generate a DFSA that accepts the same language. This is done working from the innermost nested GTS outward to the one that specifies the entire global transaction, doing the following:

1. For each arc that is labeled with the name of a nested GTS, replace the arc with the nested GTS, as follows:
   (a) Remove the arc in the parent GTS.
   (b) Insert an arc specifying the *NULL* action between the node(s) at the tail of the arc in the parent GTS and the start node of the nested GTS.
   (c) Insert arcs specifying the *NULL* action between each final state set in the nested GTS and the node(s) at the tail of the arc in the parent GTS.
2. For any arcs that both fork and join, create a new state. Remove the arc that both forks and joins, and replace it with a join arc with its head at the new state and its tail states being the same as the removed arc. Label this with the NULL action. Also, insert a fork arc with its tail at the new state, and its head states being the same as the head states of the removed arc. Label this arc with the action from the initial arc.
3. For any join arc with more than two tail nodes, let N be the head of that arc. Replace the arc with a network of nodes and arcs where all arcs have exactly two tail nodes, and all but the one ending at N having the *NULL* action. The one ending at N has the action associated with the initial arc.
4. For any fork arc with more than two heads, let N be the tail of that arc. Replace the arc with a network of nodes and arcs where all arcs have exactly two head nodes, and all but the one with its tail at N having the *NULL* action. The one with its tail at at N has the action associated with the initial arc.
5. We now have a graph where all fork arcs have exactly two heads, and all join arcs have exactly two tails. Furthermore, for each node N at the tail of a fork arc, we can assign a set of nodes S that are "downstream" from that node. Tokens that reach N can also reach any node in S. Let M be the part of the modified GTS that contains all the nodes in S that are reachable until the point where starting from N, the token will have traversed an equal number of fork and join arcs. These submachines nest within the modified GTS. Identify these nested submachines.
6. Working from innermost submachines out, replicate the GTS as follows:
   (a) Make an exact copy of the machine, giving two copies.
   (b) For each machine, eliminate a different head of the fork arc and all nodes and arcs downstream from only it. Also eliminate the dangling tail of the join arc.

   We now have two machines that together represent the concurrent execution specified in the original machine. Continue doing this with each machine created this way from the original modified GTS until no machine remains that has fork or join arcs.

7. In each machine generated by the previous step, check each pair of nodes N_a and N_b at the extreme ends of a path containing only arcs specifying the *NULL* action. Remove all arcs on the path, and merge nodes N_a and N_b.

This process leaves us with a set of GTSs that were descended from the original one, and which specify the same global transaction as was specified in the original visual language. Furthermore, the set of machines has the same concurrency properties as the original machine, and no execution dependencies have been created.

6.4 The Specification Language

A simple specification language is provided for users to define the GTS for a global transaction. This language was defined with the assumption that it will eventually be used as a basis for a graphical language/interface to define global transactions. In our language a GTS is a list of node declarations, followed by start and final state sets, followed by arc declarations, enclosed in the *START_PATTERN* and *END_PATTERN* keywords. The keyword *DEFINE_NODE* precedes each node declaration where as the keyword *DEFINE_ARC* precedes each arc declaration. Declarations of local variables used in the GTS and their types precede all other declarations. Local variables can be any type valid in the C programming language or of type *action*. Type *action* is used for a pointer to a partial order of global subtransactions defining an action for an arc. For convenience, a user may include comments in the specification language by enclosing them in square brackets.

Figure 6 shows a template of the global transaction specification language. Words in upper case letters are keywords in the language where as words in lower case letters are user supplied information. Required fields in the language are underlined in the template. Fields not underlined may be omitted. Figure 7 gives the BNF specification for the language

Following the *ACTION* keyword is either a list of procedure names, the name of a nested global transaction, or a local variable name of type *action*. Note that the three different ways to define an action can not be interspersed within the partial order for the same action.

The specification language provides some validity checking to ensure that correct GTSs are specified. Besides checking for correct syntax, the system does some semantic checks. All final state sets are checked to make sure that a node is a member of at most one final state set. All nodes referenced must be previously defined using the *DEFINE_NODE* keyword. All arcs must have at least one head and one tail node. The GTS specified by the arcs must be connected and acyclic. That is, except for nodes that are part of a start state set all nodes must have at least one incoming arc and except for nodes which part of final state sets all nodes must at least one outgoing arc.

The specification should also have the property that the number of current state tokens remains consistent throughout the graph structure. Since the graph representing the GTS is acyclic this can be accomplished using a depth-first search on the graph from each start state in the start state set. The search procedure checks that all leaves in the tree representing the depth-first search traversal are final states. It also ensures that, for every node N at the tail of a fork arc, all leaves in its depth-first search subtree are in the same final state set. For each node N, we also keep track of the number C_N of current states that must exist if the node we are visiting at the moment is a current state. C is initialized to the cardinality of the start state set for each start state. As each node N is visited, set its value C_N to C, When the path traverses a fork arc, add to C the cardinality of the fork arc minus one. When the path traverses a join arc, subtract from C the cardinality of the join arc plus one. When the node being visited is a leaf node, check that C is the same as the cardinality of the final state set that the leaf belongs to.

6.5 Examples

6.5.1 Global Transaction *PLANE*

Recall the global transaction for making a plane reservation as described in Section 6.2 above. The arc labeled "*plane_res = inquire()*" in Figure 5 represents a simple action which is executed by running the *inquire()* procedure from the transaction library. This procedure takes as arguments the source and destination cities, the dates of arrival and departure from the destination city and the earliest and latest time of departure from both the source and destination cities. The procedure queries the travel agent database for appropriate flights. In its entrance computation, the start state (A) gathers specific information regarding times and

```
START_PATTERN pattern_name [this begin a new pattern]
LOCAL_VARIABLES (type variable_name,
                 type variable_name,
                 .
                 .
                 .)
DEFINE_NODE node_name
    ENTRANCE entrance_computation_name
    EXIT exit_computation_name
DEFINE_NODE node_name
    .
    .
    .
START_STATES {list of nodes in start state sets}
FINAL_STATES {{list of nodes in final state sets}, {list of nodes in final state sets}}
DEFINE_ARC
    ALT_PRI priority
    FROM {list of nodes at tail of arc}
    TO {list of nodes at head of arc}
    ACTION {{procedure_name, order},{procedure_name, order}, ...} |
           {{nested_global_transaction_name, PATTERN}} |
           {{local_variable_name, VAR}} | [variable of type action]
           {{NULL}}
DEFINE_ARC
    .
    .
    .
END_PATTERN
```

Figure 6: Global Transaction Specification Language Template.

```
pattern spec ::= START_PATTERN ident var_list node_def start_state final_state
                                    arc_def END_PATTERN
               | START_PATTERN ident node_def start_state final_state
                                    arc_def END_PATTERN
var_list ::=     LOCAL_VARIABLES open_paren type_var_list close_paren
node_def ::=     node | node_def node
start_state ::=  START_STATES state_set
final_state ::=  FINAL_STATES open_curly state_set_list close_curly
arc_def ::=      arc | arc_def arc
node ::=         DEFINE_NODE ident | DEFINE_NODE ident node_comp
node_comp ::=    exit_comp | entr_comp | entr_comp exit_comp
exit_comp ::=    EXIT comp
entr_comp ::=    ENTRANCE comp
arc ::=          DEFINE_ARC order FROM state_set TO state_set action
                 | DEFINE_ARC FROM state_set TO state_set action
order ::=        ALT_PRI number
action ::=       ACTION open_curly action_spec close_curly
action_spec ::=  action_list | nested | lt_var | null_act
action_list ::=  action_ent | action_list comma action_ent
action_ent ::=   open_curly comp close_curly | open_curly comp comma number close_curly
nested ::=       open_curly ident comma PATTERN close_curly
lt_var ::=       open_curly ident comma VAR close_curly
null_act ::=     open_curly NULL close_curly
comp ::=         ident | expr
expr ::=         proc_name param_list
proc_name ::=    ident equals ident
param_list ::=   open_paren type_var_list close_paren | open_paren close_paren
type_var_list ::= type_var | type_var_list comma type_var
type_var ::=     type var | type star var | num
state_set_list ::= state_set | state_set_list comma state_set
state_set ::=    open_curly list close_curly
list ::=         ident | list comma ident
type ::=         action | int | char | short | long | struct ident
var ::=          ident
open_curly ::=   {
close_curly ::=  }
open_paren ::=   (
close_paren ::=  )
comma ::=        ,
equals ::=       =
star ::=         *
```

Figure 7: BNF format for Global Transaction Specification Language Template.

```
START_PATTERN PLANE
LOCAL_VARIABLES: (ACTION plane_resv,
                  char * specifics)
                  char * possible)
                  char * confirmed)
DEFINE_NODE A
                                                  [get dates and times from traveler]
   ENTRANCE specifics = get_user_info()
DEFINE_NODE B
                                                  [order flights by priority]
   ENTRANCE plane_res = compute_reservation_attempt_order(char *possible)
DEFINE_NODE C
DEFINE_NODE D
START_STATES {A}
FINAL_STATES {{D}}
DEFINE_ARC
   FROM {A} TO {B}
   ACTION{{possible = inquire(specifics)}}        [find appropriate flights]
DEFINE_ARC
   FROM {B} TO {C}
   ACTION{{confirmed = plane_res,VAR}}            [attempt reservation according to order]
DEFINE_ARC
   FROM {C} TO {D}
   ACTION{{update_db(confirmed)}}                 [update agent database]
END_PATTERN
```

Figure 8: Code for Global Transaction *PLANE*.

destinations from the traveler. As an entrance computation node B takes the results of the query and generates a partial order from the least to the most expensive of alternative flights which are available. This partial order is used for the next action. The travel agent then tries to make reservations on each alternative flight according to the partial order resulting from node B's entrance computation in the plane_res action. After making a successful reservation the travel agent updates its own database taking the customer name and flight number on which the seats were finally reserved as input.

Figure 8 shows the code defining the global transaction that makes such a plane reservation. Note that *plane_res* is the name of a local variable which points to the action generated by the entrance computation for node B. Note that all local variables associated with generated actions are initialized to an empty list. If for example the inquire action returned no appropriate flights then the global transaction would fail.

6.5.2 Global Transaction *TRIP_RESERVATION*

The GTS shown in Figure 9 contains nested global transaction *TRIP_RESERVATION*. This transaction starts in state A with several alternative actions. The preferred alternative for travel is by plane, followed by car then by train or bus. In this case, multiple arcs are needed between nodes A and $\{B, C\}$ since although both the plane and train alternatives result in the same next state set they are not part of the same action. For simplicity in drawing GTSs one arc with several labels is used to indicate several alternative actions resulting in the same next state set.

Before preceding to translate this GTS into the specification language, consider the following assumptions about the states and actions in this GTS. All actions except the *local_car()* action between nodes C and F are vital. Non-vital actions are specified by defining a *NULL* arc, as shown. This arc is traversed if the *local_car()* action fails. The exit computation on node A calculates the number of days for the entire trip. Node G has an entrance computation which prints all confirmed reservations to date.

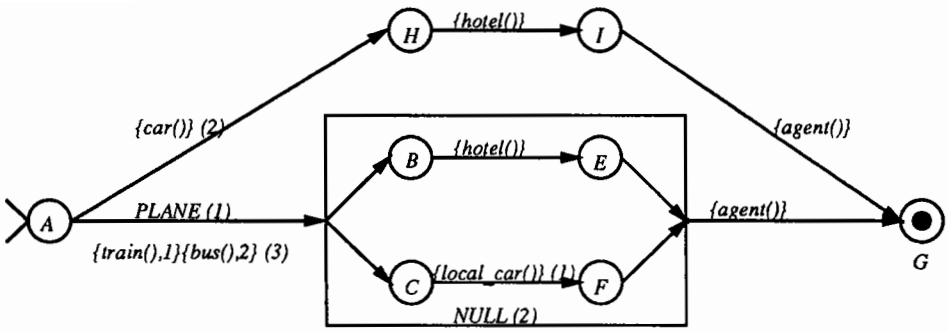

Figure 9: GTS for Global Transaction *TRIP_RESERVATION.*

```
START_PATTERN TRIP_RESERVATION
DEFINE_NODE A
    EXIT calculate_days()
DEFINE_NODE B
DEFINE_NODE C
DEFINE_NODE E
DEFINE_NODE F
DEFINE_NODE G
    EXIT print_details()
DEFINE_NODE H
DEFINE_NODE I
START_STATES {A}
FINAL_STATES {{G}}
DEFINE_ARC
    ALT_PRI 1
    FROM {A} TO {B, C}
    ACTION {{PLANE, PATTERN}}
DEFINE_ARC
    ALT_PRI 2
    FROM {A} TO {B, C}
    ACTION {{train(), 1},{bus(), 2}}
DEFINE_ARC
    ALT_PRI 3
    FROM {A} TO {H}
    ACTION {{car()}}
DEFINE_ARC
    FROM {B} TO {E}
    ACTION {{hotel()}}
DEFINE_ARC
    FROM {H} TO {I}
    ACTION{ {hotel()}}
DEFINE_ARC
    ALT_PRI 1
    FROM {C} TO {F}
    ACTION {{local_car(), PATTERN}}
DEFINE_ARC
    ALT_PRI 2
    FROM {C} TO {F}
    ACTION {{NULL}}
DEFINE_ARC
    FROM {E,F} TO {G}
    ACTION {{agent()}}
DEFINE_ARC
    FROM {I} TO {G}
    ACTION {{agent()}}
END_PATTERN
```

Figure 10: Code for Global Transaction *TRIP_RESERVATION.*

The language for the trip reservation global transaction in Figure 9 is as follows in Figure 10.

7 A Global Transaction Manager

This section describes the basic data structures and algorithms we use in our implementation of the global transaction manager.

7.1 Data Structures

A global transaction specification that defines a pattern is a tuple

$$\mathcal{P} =< NAME, TID, NODES, ARCS, S, F, C >$$

where

NAME is the name of the global transaction specification.
TID is the transaction ID of the global transaction.
NODES is the set of nodes in the global transaction specification.
ARCS is the set of arcs in the global transaction specification.
S is the start state set, where $S \subseteq NODES$.
F is the set of final state sets, where for each $N \in F$, $N \subseteq NODES$ and N is disjoint from all other final state sets in F.
C is the set of current states, where $C \subseteq NODES$.

A node in the *NODES* list is a tuple

$$\mathcal{N} =< NAME, ENTRANCE, EXIT >$$

where

NAME is the name of the node.
ENTRANCE is the procedure call for the entrance computation.
EXIT is the procedure call for the exit computation.

An arc in the *ARCS* list is a tuple

$$\mathcal{A} =< PRI, TAILS, HEADS, ACTION >$$

where

PRI is the priority of this action with respect to other alternative actions whose tails are the same as this action's tails.
TAILS contains the names of the nodes at the tail of this arc.
HEADS contains the names of the nodes at the head of this arc.
ACTION specifies the action associated with this arc. May be a partially-ordered list of global subtransaction calls, a local variable containing a partially-ordered list of global subtransaction calls, a nested pattern, or NULL.

7.2 Executing A Global Transaction

7.2.1 Overview

Assume that there is exactly one pattern for each global transaction. In enforcing that the transactions are executed correctly, an approach similar to the one used for cooperative transactions [NSZ90] is taken. The GTS associated with the patterns for all the active global transactions (i.e., those which have started but have not yet terminated) specify what can be done next.

All actions are assumed to be active. Thus, the database can determine what to do next, find the appropriate executable, execute it, and digest the results without intervention. Therefore the actual history of the actions in the database is generated according to the specifications in the patterns for execution order.

Recall that an action may be specified as a nested global transaction. In this case the global subtransactions of the nested GTSs must be executed in order to traverse the arc in the parent GTS. A nested global transaction may be one of several alternative arcs leaving the current state.

7.2.2 Algorithm

A global transaction is executed according to its GTS. The global transaction manager begins its execution when a user issues an *execute_gt* command. Execution proceeds according to the following algorithm:

1. Set the current state to the nodes which make up the start state set in the GTS structure.
2. Execute any entrance computation associated with the node(s) corresponding to the current state in the GTS.
3. For each node in the current state set, the out_arc points to the head of an ordered list of arcs which represent alternative actions. Choose the highest priority alternative action that has not yet been attempted.
   a. If several actions have the same priority choose one at random.
   b. If no unattempted action remains abort the global transaction.
4. Make sure all of the nodes at the tail of the chosen arc are in the current state set. If not, this action cannot be attempted yet. Wait until this condition holds.
5. If the arc represents a nested global transaction, find the first arc in the nested global transaction using the same procedure as outlined in steps 1-3. Recall that global transactions can nest to several levels, thus the process may need to be applied recursively until an action is found.
6. If the action is NULL, traverse the arc. Otherwise, execute the action as described in Section 7.3.2. If it fails, go back to step 3 to try the next alternative.
7. Execute the exit computation.
8. Remove the nodes at the tail of the arc from the current state list, and add the nodes at the head of the arc to current state list.
9. If all current states are part of a final state set, commit the global transaction. Otherwise, go to step 2.

7.3 Executing an Action

7.3.1 Overview

An action is defined as a partially-ordered set of one or more global subtransactions, exactly one of which must commit for the action to commit. Note that results from earlier actions may determine this partial order, or may determine how this action executes.

There are two ways to attempt an action. The first is to attempt global subtransactions according to the priority defined by the partial order. If more than one global subtransaction is of the same priority, attempt the global subtransactions in parallel. When the first one commits, commit the action. Compensate for any other global subtransactions that are in the set that was attempted and that later commit. This procedure requires compensating subtransactions to be defined for each global subtransaction since the global transaction manager would need to compensate for any global subtransactions that commit after the first.

The second way is to determine a full order consistent with the partial order. Attempt global subtransactions serially in this order until one commits at which point the action itself is committed. This is the alternative of choice if some global subtransactions do not have corresponding compensating subtransactions.

7.3.2 Algorithm

The following is an algorithm for executing actions by determining a full order consistent with the partial order and attempting global subtransactions in this order, one at a time, until one commits.

1. Find the highest priority global subtransaction not already attempted. If there is more than one with the highest priority, choose one of them arbitrarily. If none remain, return failure to the GTM

2. Execute the chosen global subtransaction's procedure in the global subtransaction library.

   a. If the global subtransaction commits, return success to the GTM, along with any return values.

   b. If the global subtransaction aborts, go to step 1.

7.4 Committing or Aborting an Action

7.4.1 Overview

An action commits when some global subtransaction that was attempted by the global transaction manager to fulfill the action commits in the local database. The NULL action always commits. An action aborts when all alternative global subtransactions that could execute that action abort.

Committing an action requires that the global transaction manager update the GTS associated with the global transaction if and only if the global subtransaction commits. The global transaction manager is notified of the success or failure of each global subtransaction, as well as receiving any return values from the subtransaction. The global transaction manager logs the action, the results of the global subtransaction, and the information required to issue a compensating transaction if and when the action must be undone, then forces the log to be written to non-volatile storage. It also updates the appropriate GTS.

If there are no communications or process failures, at the end of this procedure the global subtransaction is committed and the GTS is updated. Because the global subtransactions commit immediately, we are forced to support compensation-based recovery.

7.4.2 Algorithm

Call the global transaction GT and the global subtransaction that executed the action GST.

To commit an action, the following is done after the global transaction manager is notified of GST's commit in the local database:

1. The global transaction manager determines which action the global subtransaction satisfies.

2. The global transaction manager logs the action and the global subtransaction information in the global transaction log and forces the log to be written to non-volatile storage. Global subtransaction information includes the following:

   - The name and arguments of the global subtransaction that executed the action.
   - Any results returned by the global subtransaction.
   - The information required to generate a compensating transaction, if necessary.

3. The global transaction manager updates the GTS associated with GT.

7.5 Committing a Global Transaction

7.5.1 Overview

A global transaction commits only when the current state equals some final state set. At this point all actions are committed in their corresponding local databases. When a global transaction commits, we note in the log that the global transaction has terminated, and remove its associated GTS from the set of active GTSs. At this point we need not keep any information about the transaction including compensating transactions since once the global transaction commits the transaction is permanent.

Committing a global transaction means it cannot later be aborted even though it may have dependencies on other global transactions which may later abort. Because we do not assume that dependencies between transactions are critical, we can allow the user to specify that the commit should take effect immediately, regardless of what may happen to the other global transactions in the future. This may cause inconsistencies if some other global (or independent) transaction that this transaction "read from" later aborts. For example, let us examine the trip reservation transaction again to see how using the second option may effect the traveler. Suppose the traveler could only get a reservation in their last choice of hotels due to prior bookings

at their first choices. If a transaction which reserved a room at one of the more preferred hotels aborts, then a rooms is now available for the traveler, but the transaction making his trip reservations has already committed. In this example, the effect of the aborted transaction does not cause severe problems.

7.5.2 Algorithm

1. Check to be sure that the current state set equals some final state set. If not, do not commit.
2. Log the *global transaction commit*, and force the log to non-volatile storage.
3. Remove the GTS for the global transaction from the set of active GTSs in the global transaction manger.

8 Data Structures and Algorithms for Logging and Recovery

8.1 Logging

The log records the history of the heterogeneous multidatabase, for recovery purposes. In order for recovery to work correctly, the log must record information about when the global transactions began and ended, and information about the sequence and timing of the global subtransactions generated by by each global transaction. Also, information about the dependencies among the global subtransactions must also be recorded.

For each global transaction that was active during the history, the global transaction begin record contains the following information:

$$GT\_BEGIN =< LSN, GTID >$$

where

LSN is the log sequence number for the log entry.
GTID is the transaction identifier for the global transaction.

For each global transaction that completed during the history, the global transaction end record contains the following information:

$$GT\_END =< LSN, GTID, S >$$

where

LSN is the log sequence number for the log entry.
GTID is the transaction identifier for the global transaction.
$S \in \{COMMITTED, ABORTED\}$ is the final state of the global transaction.

Each action that was initiated by some global transaction and whose associated global transaction has committed also has a log entry containing the following information:

$$ACTION =< LSN, GTID, GS\_TID, LDB, SS, P, S, D, CT\_INFO, DEPS >$$

where

LSN is the log sequence number for the log entry.
GTID is the global transaction identifier of the global transaction that initiated the global subtransaction.
GS_TID is the global subtransaction identifier assigned by the local database.
LDB is the local database.
SS is the node name(s) of the current state(s) of the global transaction specification before the global subtransaction completed.
P is the procedure name and parameters that defined the global subtransaction.
$S \in \{COMMITTED, COMPENSATED, INVALID\}$ specifies the state of the action.
D is any data returned by the global subtransaction.
CT_INFO is any additional information that would be required to generate a procedure call for a compensating subtransaction.
DEPS is the set of LSNs for global subtransactions that this global subtransaction is pattern- or information-dependent on. Included also is a dependency on its global transaction's GT_BEGIN node.

Each compensating transaction that completes also has an entry placed in the log at the time it completes, containing the following information:

$$COMP\_ACTION =< LSN, GTID, CTID, LT\_LSN, S >$$

where

LSN is the log sequence number for the log entry.

GTID is the global transaction identifier of the global transaction that initiated the compensating transaction.

CTID is the compensating subtransaction identifier as assigned by the local database.

LT_LSN is the LSN for the log entry for the global subtransaction that this transaction compensated for.

$S \in \{COMMITTED, INVALID\}$ specifies the state.

Information such as the local database and the procedure called to do the compensation can be either directly determined from or derived from the log entry for the global subtransaction.

The LSNs for the log entries reflect the order that the entries were logged.

A correct log is one in which the log entries for the global subtransactions and compensating transactions reflect a correct history, as defined in Definition 5.2.

8.2 Recovery Algorithms

8.2.1 Additional Data Structures

For the global transaction manager to recover, especially after it has failed, it must abort any outstanding global subtransactions or compensating subtransactions. This means that, in addition to maintaining the log, the global transaction manager needs to maintain a list of global subtransactions and compensating subtransactions that are still in-progress. This list, called the *Outstanding Transaction List (OTL)*, contains a tuple

$$< LDB, GS\_TID >$$

for each outstanding global subtransaction or compensating subtransaction, where

LDB is the local database.

GS_TID is the global subtransaction identifier.

Elements are added to this list as an atomic part of the code that issues a global subtransaction or compensating transaction begin, and are removed as an atomic part of the code that processes a global subtransaction or compensating transaction end.

Recovery after an abort requires an additional list called the *Invalid LSN List (ILL)*. This list is generated and used during the recovery process itself, and contains a tuple for each of the following log entries:

1. The GT_BEGIN record for any global transaction that is in the process of being aborted.
2. The ACTION record for any global subtransaction that has committed, but is now in the process of being undone.

This tuple is of the form:

$$< LSN, ND, STATE >$$

where

LSN is the log sequence number of the invalid global subtransaction's log entry,

ND is the number of other global subtransactions that depend on this one, and

$STATE \in \{WAITING, COMPENSATING\}$ is the state of the invalid entry, where *WAITING* is the state the entry starts out in, and *COMPENSATING* means that the compensating subtransaction has been requested but has not yet committed.

The *ILL* is generated during the processing of the abort, and used to guide recovery. Also, if the global transaction manager fails during recovery, it uses the *ILL* in continuing the original recovery process when it comes back up.

8.2.2 Recovery after a Global Subtransaction Abort

This section discusses what happens when some global subtransaction decides to abort an action that has already committed, without aborting the entire global transaction.

1. Starting from the log entry for the aborted global subtransaction, traverse the log to its end, doing the following:

   - If the record is an ACTION record for a global subtransaction that has committed, invalidate the entry if any one of the following conditions holds:
     - The global subtransaction is on the one being aborted.
     - The global subtransaction is dependent on one or more other global subtransactions that have been marked as invalid.

     Invalidating a log entry means setting the state S in the log entry to *INVALID* and placing an entry in the *ILL* containing the LSN of the log entry, and with *ND* set to zero and *STATE* set to waiting.
   - If the log entry was invalidated, find the *ILL* entries corresponding to each invalid log entry that this global subtransaction depends on, and increment its *ND* counter.

2. Using the *OTL*, abort all global subtransactions and compensating transactions that depend on any global subtransaction in the *INVALID state* and have not completed. Wait until all these aborts have completed.

3. Traverse the *ILL* in inverse LSN order, issuing compensating transactions for each invalid global subtransaction once the *ND* field in the *ILL* becomes zero. This ensures that the database remains in some semblance of a consistent state in case some compensating subtransaction fails.

4. Once the ILL is empty, recovery is complete.

Note that this process may fail if some compensating subtransaction fails. This could occur, for instance, if some independent subtransaction accesses the local database in the interim and updates relevant information so the compensating subtransaction no longer has the resources it needs. At this point, manual intervention is required to complete recovery.

When compensating transactions complete, the global transaction manager needs to do some bookkeeping to ensure that recovery proceeds and terminates correctly. We have two cases, depending on whether the compensating transaction commits or aborts.

If the compensating transaction commits, the following must be done:

1. The log entry *LE* for the corresponding global subtransaction needs to be found, and its state S set to *COMPENSATED*.

2. The LSNs for the global subtransactions that were depended on by the corresponding global subtransaction are found in *LE* as well, and should be used to decrement *ND*, the dependency count, for each of these global subtransactions in the *ILL*.

3. The COMP_ACTION entry for the compensating transaction should be logged.

Most discussions on compensating transactions assume that they do not ever abort. This is unrealistic for a heterogeneous multidatabase, in that the local databases themselves can fail. We assume instead that compensating transactions are written to be idempotent, and thus we can retry them. Thus, when a compensating transaction aborts, we have the option of retrying it later. If the compensating transaction continues to fail, recovery eventually fails. At this point, a database administrator is required to rectify the database.

8.2.3 Recovery after a Global Transaction Abort

Recovery after a global transaction abort is very similar to recovery after a global subtransaction abort. The only differences are the following:

1. The first log entry invalidated is the *GT_BEGIN* entry for the aborted global transaction.
2. A *GT_END* log entry for the global transaction must be logged once recovery is complete, indicating that the global transaction was aborted.

8.2.4 Recovery from Global Transaction Manager Failure

Since global transactions in a heterogeneous multidatabase tend to be long, we do not necessarily want a failure in the global transaction manager to result in the abort of all in-progress global transactions. Rather, we want to use the log as well as other additional information kept in non-volatile storage to restore the global transaction manager to a known (correct) state consistent with the state of the local databases. The global transaction can then continue executing each ongoing global transaction from this consistent state.

During the recovery process, the global transaction manager does the following for each element in the *OTL*:

1. Attempt to abort the subtransaction. If the abort succeeds, remove the entry from the *OTL*.
2. If the abort fails, the subtransaction has already completed. However, the global transaction manager does not know whether subtransaction committed or aborted. In this case, a compensating subtransaction must be issued to remove the effects of the step from the database (in case it committed). If we assume that the compensating step is idempotent, then it will not matter if the original subtransaction in fact aborted.
3. Once the compensating subtransaction completes, remove the entry for the original subtransaction from the *OTL* as well as the entry for the compensating subtransaction. The GTS for the global transaction does not need to be updated, because it still correctly reflects the old state.

Now that there are no outstanding transactions, we need to worry about the case where the global transaction manager failed during recovery. This involves the following steps:

1. Re-traverse the log in chronological order starting from the LSN in the first entry of the *ILL*, ensuring that all invalid log entries are correctly identified and represented in the *ILL*.
2. Continue as if the abort is being processed, starting from the inverse traversal of the *ILL* in the section on global subtransaction abort.

8.2.5 Recovery from Communication or Local Database Failure

Communication and process failures during the execution or commitment of some global subtransaction may cause problems. In this section, we are assuming that the state of the GTS is flushed out to disk every time some action commits. We also assume that a local commit ensures that the information is permanent in the local database. Failures of the local database or the communication links can cause the following problems:

1. The global transaction manager starts global subtransaction T, but the commands to run T are lost because of a communication failure before it reaches the machine the local database runs on.
2. The global transaction manager initiates global subtransaction T, but never receives a response because T is deadlocked, livelocked, or stuck in some infinite execution state.
3. The global transaction manager initiates global subtransaction T, but the machine on which T is running fails before T terminates (either commits or aborts).
4. The global transaction manager initiates global subtransaction T, and T runs to completion, but the reply is lost because of a communication failure.

Cases 1-4 look the same to the global transaction manager, and can thus be handled in the same way. The global transaction manager has the option of either waiting for a reply or attempting to get the transaction back into a known state using the algorithm presented below.

Note also that the global transaction manager can continue to try alternative actions even if some global subtransaction T is stuck. Once the global transaction manager has gotten the transaction to abort or compensated for it, it has the option of either leaving things as they are, or resubmitting T and aborting any alternative that succeeded.

The following procedure can be used to recover from communication or process failure during the execution or commit of an action. Again, let GT be the global subtransaction, and let GST be the global subtransaction that is currently attempting to execute the action.

1. Try to abort. If this succeeds, the global transaction manager knows for sure that GST failed, and that the GTS for GT is in a correct state.

2. If the global transaction manager receives a message that the abort failed, then GST has run to completion, but we do not know if it committed or aborted. Assume that it committed, and run the transaction that compensates for GST until the compensation succeeds. Note that if GST actually aborted, the compensation should do nothing because the compensation procedure is idempotent. Once the compensation succeeds, the global transaction manager knows that the GTS for GT is in a correct state.

3. If the global transaction manager receives no response, either the communication link or the machine running GST is down. If the machine running GST is down, then the local database recovery should abort GST, because we assume that all local databases support atomic transactions. However, the local database may not know where to send the response.

4. If the communication link is down, the global transaction manager will eventually resume contact with the local database. At this point, the global transaction manager can try to abort or compensate for GST again. This time, it should succeed.

9 Conclusions

In this paper, we presented a model for global transactions on heterogeneous multidatabases called the *whole transaction model.* This model is characterized by the fact that each step in the global transaction is executed as a complete transaction on the local database.

In this paper, each global transaction is defined as a pattern of actions/steps. There is no notion of conflict, except in that each step is assumed to be atomic. We explored correctness constraints for global transactions in the whole transaction model. If no aborts or failure occur, correctness really just means assuming each global transaction executes its steps in an appropriate order.

Because global transactions commit their work early, the whole transaction model defines a notion of recovery based primarily on compensation. We define a notion of correctness in the presence of recovery, based on the ability for compensating subtransactions to commute backwards in the history in certain conditions. A correct global history is one that is equivalent to one in which each global subtransaction that is invalidated is followed directly by a successful compensating subtransaction, according to the type of commutation defined in the commutation lemma (Lemma 5.1), and the projection of the valid entries from the log is correct according to the non-recovery correctness criteria.

Based on these notions, we present a way to specify global transactions. We also specify algorithms for executing those global transactions and recovering from aborts or failures. We have implemented and tested a simple global transaction manager that uses the execution algorithms, though it does not yet support recovery.

This work explores the least restrictive end of a spectrum of correctness criteria that are defined in our current research on InterActions[1]. It has some specific problems, the most serious of which are caused by the ability of the independent transactions to interfere with the state of the database while the global transaction is running. This means, for instance, that a compensating subtransaction may not succeed because no sound history [KLS90] can be generated by extending the current history of some local database. The work on InterActions addresses these issues explicitly.

[1] This work will appear in a later Technical Report.

References

[Bar90] Kenneth Barker. Transaction management on multidatabase systems. Technical Report TR 90-23, Department of Computing Science, The University of Alberta, 1990.

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

[BST90] Yuri Breitbart, Avi Silberschatz, and Glenn R. Thompson. Reliable transaction management in a multidatabase system. In *SIGMOD Proceedings*, pages 215–224, 1990.

[DE89] Weimin Du and Ahmed K. Elmagarmid. Quasi-serializability: A correctness criterion for global concurrency control in interbase. In *Proceedings of the 15th VLDB*, pages 347–355, 1989.

[DE90] Weimin Du and Ahmed K. Elmagarmid. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of the 6th Int'l Conference on Data Engineering*, February 1990.

[EH88] A. Elmagarmid and A. Helal. Supporting updates in heterogeneous distributed database systems. In *Proceedings of the 4th Int'l Conference on Data Engineering*, 1988.

[ELLR90] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for interbase. In *VLDB Proceedings*, pages 507–518, 1990.

[GGK+90] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Coordinating multi-transaction activities. Technical Report CS-TR-247-90, Princeton University Department of Computer Science, February 1990.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *VLDB Proceedings*, pages 144–154, 1981.

[GS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *ACM SIGMOD Proceedings*, pages 249–259. ACM, 1987.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In *VLDB Proceedings*, pages 95–106, 1990.

[NFSZ90] Marian H. Nodine, Mary F. Fernandez, Andrea H. Skarra, and Stanley B. Zdonik. Cooperative transaction hierarchies. Technical Report CS-90-03, Department of Computer Science, Brown University, February 1990. Revised February, 1991.

[NSZ90] Marian H. Nodine, Andrea H. Skarra, and Stanley B. Zdonik. Synchronization and recovery in cooperative transactions. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice*, pages 329–342. Morgan Kaufmann, 1990. also Proceedings of the 4th International Workshop on Persistent Object Systems.

[Pu88] Calton Pu. Superdatabases for composition of heterogeneous databases. In *Proceedings of the 4th International Conference on Data Engineering*, pages 548–555, 1988.

[SBD+81] J. M. Smith, P. A. Bernstein, U. Dayal, N. Goodman, T. Landers, K. W. T. Lin, and E. Wong. Multibase – integrating heterogeneous destributed systems. In *Proceedings of the National Computer Conference*, pages 487–499, 1981.