BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M4

# Pattern Specification and Global Transaction Management in Heterogeneous Multidatabases

by
Patrice Tegan

# Pattern Specification and Global Transaction Management in Heterogeneous Multidatabases

by
Patrice A. Tegan

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science
in the Department of Computer Science at Brown University
February, 1991

This submission by Patrice Ann Tegan is accepted in its present form
by the Department of Computer Science in partial fulfillment of the
requirements for the Degree of Master of Science.

Date 2/27/91

Stanley B. Zdonik
Advisor

# Pattern Specification and Global Transaction Management in Heterogeneous Multidatabases

Patrice A. Tegan
Brown University
Providence, RI 02906

February 12, 1991

**Abstract**

A heterogenous multidatabase allows a user to manipulate data contained in a collection of individual, autonomous databases managed by different database management systems. This paper discusses an implementation of the *whole transaction 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 according to the whole transaction model. The whole transaction model is distinguished from other heterogeneous multidatabase transaction models in that it does not require the transactions on the multidatabase to be atomic. Rather, the user can specify how the different multidatabase transactions are allowed to interleave.

# 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. The *whole transaction model* is distinguished from other heterogeneous multidatabase transaction models in that it does not require the transactions on the multidatabase to be atomic. Rather, the user can specify how the different multidatabase transactions are allowed to interleave.

This paper gives a summary of the whole transaction model followed by a discussion of 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 according to the whole transaction model.

A heterogeneous database consists of two logical levels. The *local* level is comprised of autonomous databases, hereby referred to as local databases, which access data through *local transactions*. The model assumes 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 contained in more than one local database. The *global transaction manager* enforces correctness and persistence guarantees at the global level. Note that such guarantees can be no stronger than those made in the local databases.

The following travel agent example, first proposed by Gray [Gra81], is used 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 sees the travel plans as one (global) transaction of long duration. This global transaction consists of a series of local transactions which cooperate 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 local transaction. Thus, the car rental is accomplished by a complete transaction on the car rental database. The global transaction coordinates the local transactions. The global transaction also maintains an internal context which is updated based on the responses from the local transactions. Within a global transaction, a local transactions is often dependent on prior local transactions; for example the travel agent can not make an airline reservation without first getting the travel dates and destinations from the traveler. Other local transactions 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 global transactions are not necessarily atomic, but are viewed as collections of local transactions that execute and commit in some constrained order. This order is specified using a variant of the notion of *patterns and conflicts* as defined in [NSZ90].

In the whole transaction model, the global transaction manager initiates a series of local transactions, where each local transaction executes as a single piece of code which resides in a transaction library. The global transaction manager receives a response from the local transaction manager indicating whether the transaction committed or aborted.

Local database management systems are not modified to facilitate the operation of the global transaction manager. Because of the atomicity restriction on the local transactions, once a transaction commits its effects 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 any time prior, the global transaction may be aborted. For example, suppose the traveler before receiving confirmation of the trip arrangements decides to cancel the trip. At this point the local transactions have already updated the airline, car and hotel databases. These transactions 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 local transaction 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. It is assumed that the procedures in the transaction libraries deal with such issues. Instead higher-level transaction issues like *synchronization*, *replication*, and *transaction specification techniques* are stressed in the model and implementation.

The rest of the paper is organized as follows: In Section 2 a summary of the related research is given. Section 3 summarizes the global transaction model as originally proposed by Marian Nodine and gives an overview of her definition of patterns and conflicts explaining how these patterns are used to specify global transaction execution and correctness criteria. A graphical language and its programming equivalent for transaction specification is discussed in Section 4. Section 5 outlines the actual implementation of a global transaction manager in a heterogeneous multidatabase server. Section 6 concludes the paper. Data structures and the actual code for the implementation of the global transaction manager are provided in Appendix A. Appendix B describes the specifics of how to use a prototype of a database server for language specification and global transaction management and briefly discusses problems with the current prototype and suggested future enhancements.

# 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. in the 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.

Federated databases differ from heterogeneous multidatabases in that they do not assume the same level of autonomy in the individual databases. In a federation of databases each knows about other members of the federation and can decide with whom to share information. Heimbigner and MacLeod [HM85] view a federated database as a dynamic set of local databases that may cooperate on behalf of the users of the local database to retrieve information from different databases in the federation. This differs from the whole transaction model in that we view the local databases as cooperating through a global interface rather than through each other. In the whole transaction model full communication autonomity is assumed. Thus individual local databases do not have any information regarding even the existence of the other local databases.

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. These include [EH88], [BST90] and [AGMS87]. Atomicity at the global level often restrictions 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 atomity and serializability is 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. Quasiserializability requires global transactions to execute serially, but does not restrict local transaction execution. Their claim is that subtransactions of the same global transaction contain no global integrity constraints nor are there value dependencies among subtransactions for the same global transaction.

Elmagarmid, Leu, Litwin and Rusinkiewicz [ELLR90] offer a model which supports a relaxed approach for non-atomic transactions. The flow of the global transaction is controlled by a predicate petri net. Participating database systems are assumed to be autonomous, as in the whole transaction model. Function replication is also addressed here. This model although similar to the whole transaction model does not deal with conflicts between concurrently executing global transactions.

Garcia-Molina et.al. [GMS87, GMGK+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. However, this work does not address how to restrict the interleaving of sagas and nested sagas where necessary.

# 3 The Whole Transaction Model

This section describes in summary the work of Marian Nodine on the whole transaction model. The intent of this section is to familiarize the reader with the model enough to understand the language and implementation described in the remainder of the paper. For a more detailed description of the model and architecture see [NT91]. The work also contains a more rigorous analysis of global transaction correctness and recovery.

## 3.1 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 made 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. In the whole transaction model all local transactions are atomic, serializable, and recoverable. No assumptions are made about the underlying data models, allowing *heterogeneous* data models to be supported. Local databases need not communicate with each other or even know about each others' existence. Thus, the whole transaction model gives full *communication autonomy*. *Execution autonomy* relates to how the local databases execute their transactions. Local databases can execute any transactions, including both local transactions generated by the global transaction manager and *independent transactions* initiated by other agents.

A global database is seen 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 local databases forming the heterogeneous multidatabase.

Global transactions are not necessarily atomic. Rather, a global transaction is viewed as defining how a set of local transactions can *cooperate* to do some task that spans multiple local databases. This is similar to earlier work on *cooperative transactions* by [NSZ90]. In the whole transaction model, global transactions are defined by sequences of operations that share state and work together to accomplish a specific task.

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, these interactions are considered non-critical, and will be ignored during normal database operation. [NT91] discusses this assumption and its ramifications in more detail.

## 3.2 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 *local transactions*.

The *local transaction library* has a set of parametrized procedures that can be called to initiate specific transactions in the local database. The library defines all possible local transactions 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 local transaction committed or aborted.

The *local data managers* are the servers responsible for executing the local transactions. All accesses and updates in the heterogeneous multidatabase are ultimately executed by the

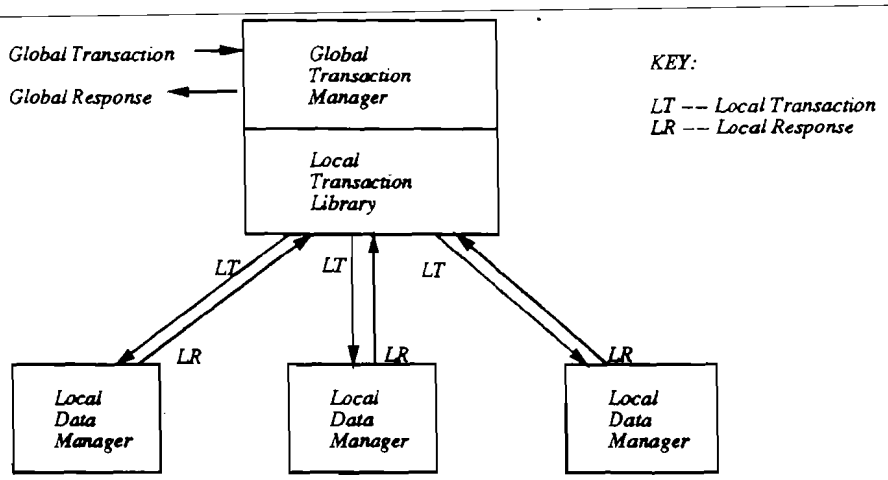


Figure 1: A Multidatabase Architecture Using the Whole Transaction Model.

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.

## 3.3 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. 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 the 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.

## 3.4 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 local transaction on a single database. Since both function and data replication are allowed in the model, there may be more than one local transaction that can accomplish a specific action. A *pattern* defines the actions that are necessary to accomplish the global transaction's 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.

In addition to defining the global transaction, patterns also serve as a correctness specification for the global transaction. Because of this, actions that may not occur at a specific point in the execution of the global transaction (*conflicts*) may also be defined within the context of patterns.

This definition is similar to the patterns and conflicts defined for cooperative transactions in [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 local transactions and do all computations necessary for completion of the transaction without additional input from the definer of the global transaction.

## 3.5 Transaction Definition

Global transactions may 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.

*Conflicts* may also be specified by a global transaction. A conflict specified at some point in a global transaction $GT$ indicates actions (actually, local transactions) that cannot be initiated at that point in the execution of $GT$. That is, conflicts restrict how global transactions can interleave their actions.

In the pattern specification techniques, described in detail in Section 4, one of the goals was to give the user certain flexibility in specifying global transactions. *Function replication* and *non-vital actions* facilitate in this aim.

*Function replication* means that the same objective can 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 local transactions. For instance, if the action is making a plane reservation, alternative local transactions might be to use United airlines or to use Delta airlines. Another way is to use alternative actions. For instance, you may want to fly to another city and rent a car, or you may want to 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, using public transportation is a viable alternative. Non-vital actions are desirable, but not necessary. Therefore, a non-vital action will always be attempted, usually as the last alternative , but if it fails, the global transaction will not fail.

### 3.5.1 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. Arcs in the GTS represent actions that need to complete for the global transaction to change state.

The start state of the GTS represents the beginning of the pattern for the global transaction, and final states represent possible ends of the pattern.

A conflict is an action, possibly initiated by some other global transaction, that cannot occur when this global transaction is in some specific state. A conflict is represented in the GTS as an arc to a dead state (conveniently named DEAD). Thus, if a conflicting action is taken, this global transaction cannot successfully complete its pattern.

We see that each global transaction is a regular expression in some language, where the terminal symbols in the language are actions.

A GTS is formally defined as follows:

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

$N$ is the name of the GTS,

$K$ is a finite set of states, including the DEAD state,

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

$\Delta$ is a transition function from $K^i \times \Sigma \rightarrow K^j$, where $i \geq 1$ and $j \geq 1$,

$s \in K$ is the initial or start state set,

$c \in K$ is the set of current state, and

$F \subseteq K$ is the set of final state sets.

### 3.5.2 Action Specifications

An action is a set of local transactions, 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 the local transactions should be tried, the set of local transactions in an action is partially ordered.

A local transaction 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 used for non-vital transactions. The arcs labeled with the NULL action are automatically traversed.

# 4 Global Transaction Specification

## 4.1 A Visual Language

Patterns are used for global transactions specifications in heterogeneous databases. Each global transaction is defined by a single pattern. In this section the representation of patterns or GTSs are discussed. Changes need to be made for the following reasons:

1. 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. Thus computations internal to the global transaction need to be associated with the GTS at specific points. These computations may be used, for example, to tell the global transaction what actions to try next, or to specify parameters for a given action.


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

2. Function replication allows the same task to be performed equivalently by either one of several actions or an action to be accomplished by one of several local transactions. A priority order needs to be presented among these alternative actions or local transactions. 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 initiate operations in the desired order.

3. A GTS needs to specify the potential for parallel actions or sets of actions whose execution order is nondeterministic.

A GTS is represented as a graph. A node in the graph corresponds to a state (or a part of a state). 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 GTS. A set of final states indicates the end of the GTS. These final states are partitioned into final state sets. A global transaction can only terminate when the current state set is identical with 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. 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.

Conflicts defined in a GTS for a global transaction indicate when other global transactions cannot access the same data. This occurs at the level where the local transactions

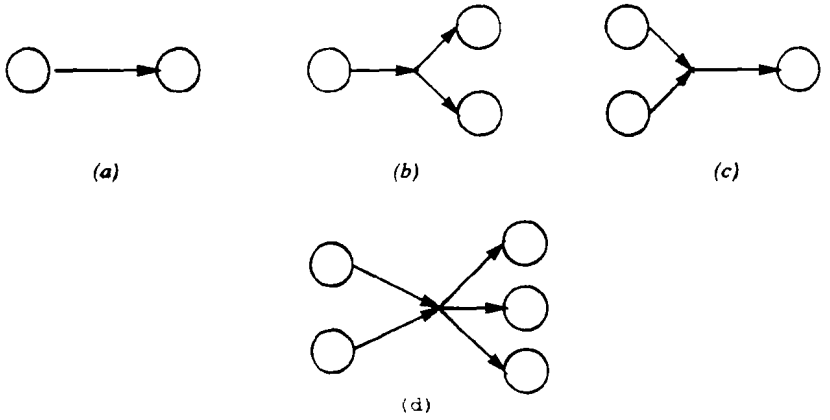


Figure 3: Types of arcs. (a) Simple arc. (b) Fork arc. (c) Join arc. (d)An arc that both forks and joins.

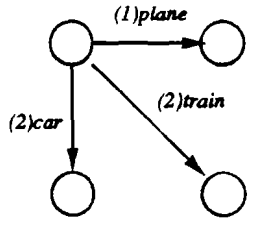


Figure 4: Alternative actions in a global transaction.

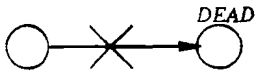


Figure 5: A Conflict Arc

are manipulating data in the local databases. Thus, a conflict occurs between some local transaction which may be used to accomplish some action, and some other global transaction that does not want that particular local transaction to be executed.

A conflict arc specifies the set of local transactions that conflict at this point. Given a GTS that describes a transaction, a conflict arc specified within that GTS defines one or more local transactions that one or more other global transactions cannot do at a specific point. For example, a conflict might state that once an inquiry has been made about a particular flight, no other global transaction can inquire about the flight or change a reservation until this global transaction decides whether or not to make a reservation on the flight. Barring interference by independent transactions in the local database, this conflict ensures that no other global transaction will do anything to interfere with the information returned by the inquiry.

Conflicts are indicated in a GTS by an X on the arc representing the action, as shown in Figure 5. The head of a conflict arc is always the DEAD state.

## 4.2 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 a nested global transaction or an action is specified on an arc, but a nested global transaction may not be an alternative in the partial order of local transactions which compose an action. Recursive nesting of global transactions is not allowed, thus nesting does not change the power of the transaction which 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 and its associated conflicts. Traversing the arc representing a nested global transaction is equivalent to traversing the complete child GTS, starting as some start state and finishing in some final state set. All conflicts in the parent GTS are active while executing the child GTS. Thus once the child GTS is initiated, all operations are checked first against the parent GTS for conflict or queue requests and then against the child GTS for conflict, queue or accept. A nested GTS is can be one of several alternative actions. If the child GTS aborts, the parent GTS may attempt an alternative action from the current node.

Nested global transactions complete under the same conditions as a top-level global transaction. When all the current state tokens are final states in a final state set, the nested global transaction can commit and terminate. The completion of a nested global transaction does not indicate a commit of the parent GTS, but the traversal of the arc associated with the nested transaction.

Let us consider the trip reservation example. The GTS for this example is shown in Figure 6. The process of making a plane reservation may require the travel agent to query

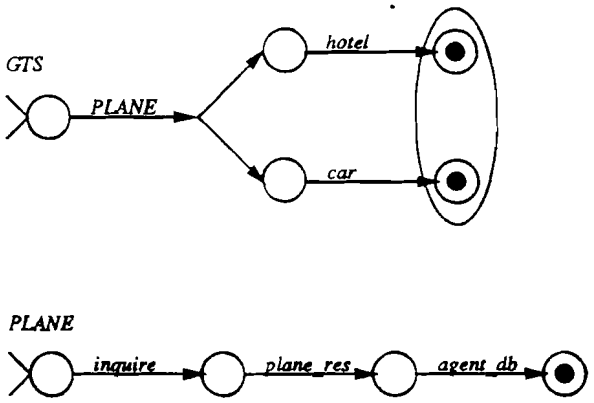


Figure 6: Global Transaction Specification for the Plane Reservation Example.

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. Plane_res is a partial order of airline specific seat reservations. 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 tPLANE nested transaction is complete and the current state tokens are moved to the two middle states in the parent transaction's GTS.

## 4.3 The Specification Language

A simple specification language is provided for users to define the GTS and conflicts associated with 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 local transactions defining an action for an arc. For convenience, a user may include comments in the specification language by enclosing them in square brackets.

Figure 7 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 8 gives the BNF specification for the language

The *ENTRANCE* and *EXIT* fields may be omitted if a node does not have any such computations. The *CONFLICT* keyword is specified if a particular arc represents a conflict. If the *CONFLICT* keyword is omitted then the arc is an action arc. The head state of all conflict arcs is the system specified *DEAD* state.

Following the *ACTION* keyword is either a list of procedure names, the name of a nested

```
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, if any
     HEAD_STATES {list of nodes at head of arc}
    TAIL_STATES {list of nodes at tail of arc}
    CONFLICT
    ACTION {{procedure name, order},{procedure name, order}, ...} |
           {{name of nested global transaction, PATTERN}} |
           {{name of local variable of type action, VAR}} |
           {{NULL}}
DEFINE_ARC
.
.
.
END_PATTERN
```

Figure 7: Global Transaction Specification Language Template.

```
pattern spec → pat_name
               node_def
               start_state
               final_state
               arc_def
               END_PATTERN
               | pat_name
               var_list
               node_def
               start_state
               final_state
               arc_def
               END_PATTERN
pat_name     → START_PATTERN ident
var_list     → LOCAL_VARIABLES (type_var)
node_def     → node
               | node comp
               | node_def node
               | node_def node comp
arc_def      → act_arc_def
               | con_arc_def
               | arc_def act_arc_def
               | arc_def con_arc_def
con_arc_def  →conflict
               | arc_def conflict
act_arc_def  → act_arc
               | arc_def act_arc
act_arc      → arc_name head_state
               tail_state action
               | arc_name order head_state
               tail_state action
conflict     → arc_name head_state
               tail_state CONFLICT action
order        → ALT_PRI number
comp         → exit_comp
               | entr_comp
               | entr_comp exit_comp
start_state  → START_STATES list }
final_state  → FINAL_STATES {state_list}

action_list  → action_ent
               | action_list , action_ent
action_ent   → { ident COMMA number }
               | { ident }
               | { expr , number }
               | { expr }
action       → ACTION { action_list }
               |ACTION { nested }
               |ACTION { lt_var }
state_list   → list }
               | state_list COMMA list }
node         → DEFINE_NODE ident
exit_comp    → EXIT ident
               | EXIT expr
entr_comp    → ENTRANCE ident
               | ENTRANCE expr
param_list   → ( type_var )
expr         → proc_name param_list
               | proc_name ()
type_var     → type var
               | type var var
               | type * var
               | type_var , type var
               | type_var , type var var
               | type_var , type * var
proc_name    → ident = ident
arc_name     → DEFINE_ARC
head_state   → HEAD_STATES list }
tail_state   → TAIL_STATES list }
list         → {ident
               | list ,ident
nested       → { ident , PATTERN }
lt_var       → {ident , VAR R}
type         → TYPES
var          → num
               |ident
```

Figure 8: BNF format for Global Transaction Specification Language Template.

global transaction, or a local variable name of type *ACTION*. Note that the three different *types of actions* can not be interspersed within the partial order for same action.

The specification language provides some validity checking to ensure that correct GTSs are specified. Besides checking for correct syntax, the system performs 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 with the *DEFINE_NODE* keyword. All arcs must have at least one head and one tail node. All nodes except those that are part of a start state set 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 GTS specified by the arcs must be connected and acyclic, these checks are performed using well known algorithms for cycle detection and connectivity [AHU74].

The specification language should also make sure that the number of current state tokens remains consistent through out the graph structure. New current state tokens are only generated by traversing fork arcs. Current state tokens are only removed from the set by the traversal of a join arc. Since the graph representing the GTS is acyclic this can be accomplished by performing a modified *depth first search* of the non-conflict arcs and checking that all leaves are final states. By definition, depth first search visits every node and thus every path, starting with nodes which make up the start state and ending with the nodes which compose final state sets. A fork arc essentially If all leaf nodes are final states, then each path from the root ends in a final state. If there were no fork or join arcs then the total number of tokens would remain consistent through out the graph created by the GTS. For each new path created by a fork arc, this search guarentees that either there are a consistent number of final state tokens as there are start states and tokens created by fork arcs or that some join arc caused the path created by the fork to rejoin in an already found path to the final states set. At this point, all paths created by the fork arc are marked as part of the same set. When the end of the path or the final state is reached this node is added to a working final state set. When all paths stemming from the fork arc have been traversed this working final state set is checked against specified final state sets and the working final state set is reset to NULL.

When a join arc is traversed, a token is removed from the current state set, as dictated by the return to an already traversed path the a final state.

## 4.4 Examples

### 4.4.1 Global Transaction PLANE

Recall the global transaction for making a plane reservation as described in Section 4.2 above. The arc labeled inquire in Figure 6 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. The start state, node A, gathers specific information regarding times and destinations from the traveller. 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 9 shows an example of what the language would looks like to make 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.

### 4.4.2 Global Transaction TRIP RESERVATION

The GTS shown in Figure 10 contains nested global transaction PLANE. 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 alteratives 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 that indicated by the arc between nodes C and F are vital. Non-vital actions are specified by defining a NULL arc, as shown between nodes C and F. This arc is traversed if all local transactions in the partial order associated with the car action fail. Assume that car is actually a nested GTS similar to the plane GTS as shown in Figure 6. 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. Note the conflict arc between nodes C and DEAD, the system specified dead state. This conflict indicates that no other global transactions may update the agent database until the car arc has been traversed. Conflicts indicate actions that can not be taken at the global transaction level, but do not govern any local level transactions. Also note that when an action has only one local transaction, the priority of this action may be omitted.

The language for the trip reservation global transaction in Figure 10 is as follows in Figure 11.

# 5 A Global Transaction Manager

## 5.1 Executing A Global Transaction

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) together specify what can be accepted next, as opposed to what operations should be rejected or queued.

```
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
   HEAD_STATES {B}
   TAIL_STATES {A}
                          [find appropriate flights]
   ACTION{{possible = inquire(specifics)
DEFINE_ARC
   HEAD_STATES{C}
   TAIL_STATES{B}
                          [attempt reservation according to order]
   ACTION{{confirmed=plane_res,VAR}}
DEFINE_ARC
   HEAD_STATES{D}
   TAIL_STATES{C}
                          [update agent database]
   ACTION{{update_db(confirmed)}}
END_PATTERN
```

Figure 9: Code for Plane Reservation Example.

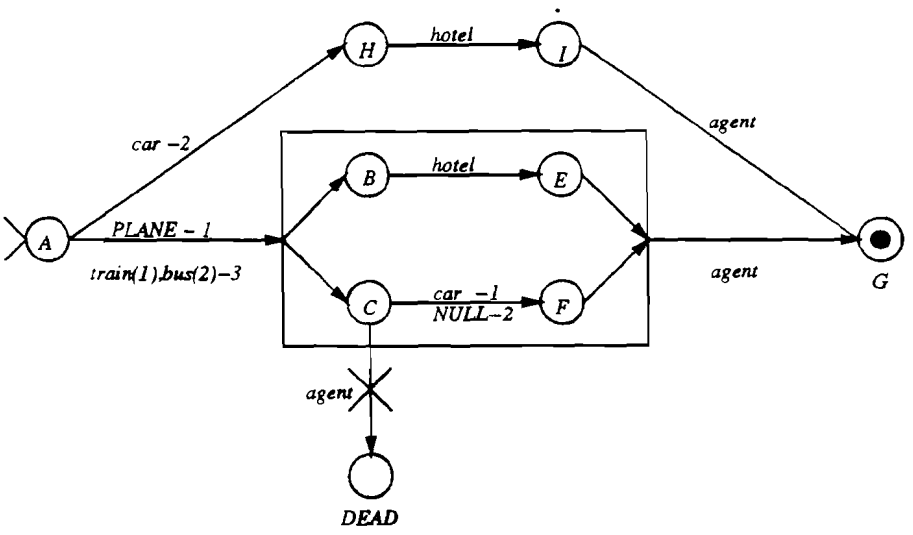


Figure 10: Nested Global Transaction Specification Example.

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 deterministically 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 local transactions 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 alternatives arcs leaving the current state.

A global transaction is executed according to its specified pattern or GTS. A global transaction begins execution when a user issues an *execute_gt* command from the client. The client and server communicate through an *RPC-type* interface. The *execute_gt* command takes the name of the GTS which represents the global transaction as an argument. The global transaction manager looks for the structure by name. If such a GTS has not already been defined then the global transaction manager automatically aborts the global transaction. If the GTS is found then the global transaction manager executes the global transaction using the steps outlined below. Each global transaction runs in a separate thread in the global transaction manager.

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 and add any conflict arcs to the global conflict list.

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.

```
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
   HEAD_STATES {B,C}
   TAIL_STATES {A}
   ACTION {{PLANE,PATTERN}}
DEFINE_ARC
   ALT_PRI 2
   HEAD_STATES {B,C}
   TAIL_STATES {A}
   ACTION {{train,1},{bus,2}}
DEFINE_ARC
   ALT_PRI 3
   HEAD_STATES {H}
   TAIL_STATES {A}
   ACTION {{car,PATTERN}}
DEFINE_ARC
   HEAD_STATES {E}
   TAIL_STATES {B}
   ACTION {{hotel}}
DEFINE_ARC
   HEAD_STATES{I}
   TAIL_STATES {H}
   ACTION{ {hotel}}
DEFINE_ARC
   ALT_PRI 1
   HEAD_STATES {F}
   TAIL_STATES {C}
   ACTION {{car,PATTERN}}
DEFINE_ARC
   ALT_PRI 2
   HEAD_STATES{F}
   TAIL_STATES {C}
   ACTION {{NULL}}
DEFINE_ARC
   HEAD_STATES {G}
   TAIL_STATES {E,F}
   ACTION {{agent}}
DEFINE_ARC
   HEAD_STATES {G}
   TAIL_STATES {I}
   ACTION {{agent}}
DEFINE_ARC
   HEAD_STATE {DEAD}
   TAIL_STATE {C}
   CONFLICT
   ACTION {{agent}}
END_PATTERN
```

Figure 11: Code for Language for Global Transaction trip reservation

b. If no unattempted action remains abort the global transaction.

c. If arc is NULL, than traverse it and go to step 7.

4. 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.

5. Make sure all nodes at tail of arc are in the current state set.

6. Execute action. (See Section 5.3)

7. Execute exit computation.

8. Remove node from current state list and conflicts from global conflict list

9. Follow head_node pointers to set of nodes at head of arc representing action just executed and add to current state list.

10. Check if all current states are part of a final state set. If not go to step 2 else commit global transaction.

For the global transaction to commit, all current states should be final states in the same final state set, with one current state token per state in the final state set.

## 5.2 Checking for Conflicts

When a local transaction is executed, the GTM checks to see if the local transaction to be attempted is present in the global conflict list. If not then no conflict exists and the local transaction is attempted. If the local transaction that is being attempted is present, then a conflict exists. There are two options. The global transaction that initiated the local transaction as a part of its GTS specifies at start up which choice it prefers by giving *QUEUE* or *REFUSE* as parameters to the *execute_gt* command.

1. *QUEUE:* The global transaction manager requests that all local transactions are queued, which means the global transaction manager should wait until the conflict no longer exists, then try the local transaction again. (Note: this option introduces some potential for deadlocks).

2. *REFUSE:* The global transaction requests that it all conflicts be refused, which means that when the conflict occurs, the global transaction manager should treat the conflicting operation in the same way as if the local transaction failed. The global transaction manager continues in its attempts to find some other local transaction that succeeds for the current action.

The prototype linearly searchs the global conflict list which contains the conflict arcs for all active global transactions. If the procedure name is found then if *QUEUE* is specified queue the local transaction and wait for the conflict to be resolved. If *REFUSE* is specified a

conflict exists and the global transaction is notified of the conflict. The next highest priority alternative local transaction is attempted as discussed in the previous section. If the search of global conflict list is exhausted without finding the local transaction then there is no conflict. Note that the global conflict list may not be modified from the time a search is started to the time that the actual local transaction has completed.

## 5.3 Executing an Action

An action is defined as a partially-ordered set of one or more local transactions, 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 involves determining a full order consistent with the partial order and attempting local transactions in this order, one at a time, until one commits. At this point the action itself is committed. This is the alternative used in the prototype. A detailed algorithm follows:

1. Find the highest priority local transaction not already attempted. If none, return failure to GTM
2. Check for conflicts. Accept, queue or refuse request based on user input.
   a. If there are no conflicts go to step 3
   b. If the local transaction conflicts and is refused go to step 1 with next highest priority alternative local transaction.
   c. If local transaction is to be queued due to a conflict, wait for resolution then if accepted to go step 3 , if rejected go to step 1.
3. Execute the procedure in the transaction library which corresponds to this local transaction.
   a. If action succeeds commit local transaction and return success to GTM.
   b. If action fails go to 1 with next local transaction in partial order.

The second way to execute actions is to attempt local transactions according to the partial order. If more than one local transaction is of the same priority, attempt to execute them in parallel, each in a separate thread. When the first one commits, commit the action. This procedure requires compensatory transactions for each local transaction since the GTM would need to compensate for any local transactions that commit after the first.

## 5.4 Committing an Action

Call the global transaction GT and the process that initiates the local transaction T. The global transaction manager is GTM. The semantics of the commit is that the GTM updates the GTS associated with GT if and only if T commits. The procedure for running T in the global database is as follows:

1. GTM initiates T to start the transaction in the local database. If the local transaction commits, the following is done:

    a. T commits and the GTM is notified. The GTM determines which action the local transaction satisfies.

    b. GTM updates the GTS associated with GT. GTM then logs the action and the local transaction information in the global transaction log and forces the log to be written to non-volatile storage. (Local transaction information includes the name of the procedure that runs the local transaction, the arguments that were originally specified, the results, and the name of the procedure that runs the compensating transaction.)

2. If the local transaction aborts, the action in the GTS is not done.

    a. If no alternative local transactions are being attempted in parallel, GTM should try the next alternative local transaction if one exists.

    b. If there are alternative local transactions being attempted in parallel then wait for outcome of such alternatives.

    c. If there are no further alternative local transactions, repeat step a. and b. above with any alternative actions. If no alternative actions exist abort global transaction.

Obviously if there are no communications or process failures, at the end of this procedure the local transaction is committed and the GTS is updated. This means that if the action is later invalidated because of some abort, the recovery process must be based on issuing compensatory transactions.

The implementation of the prototype does not handle any failure or recovery and thus does not log actions as discussed in this section. However, note that communication and process failures may cause the GTS not to know the status of the local transaction. Assume that the state of the GTS is flushed out to disk every time some action commits. Also assume that a local commit ensures that the information is permanent in the local database. Failures could result in one of the following situations:

1. GTM sends a message to start local transaction T, but the message is lost because of a communication failure before it reaches the machine the local database runs on.

2. GTM initiates the local transaction T, but never receives a response because T is deadlocked, livelocked, or in some infinite execution state.

3. GTM initiates the local transaction T, but the machine on which T is running fails before T terminates (either commits or aborts).

4. GTM initiates the local transaction T, and T runs to completion, but the reply is lost because of a communication failure.

5. The local transaction T is run to completion, but the GTM fails in the meantime, and T cannot respond. The resolution here is entirely dependent on the policies implemented in the local database.

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

1. Try to abort T. If this succeeds, the GTM knows what happened.

2. If the GTM receives a message that the abort failed, then T has run to completion, but we do not know if it committed or aborted. Run the transaction that compensates for T until the compensation succeeds. Note that if T actually aborted, the compensation should do nothing because the compensation procedure is idempotent. Once the compensation succeeds, the GT is in a known state.

3. If the GTM receives no response, either the communication link is down or the machine the local database is running on is dead. If the machine is dead, it should eventually come up. At this point the local database recovery should abort T, because we assume that all local databases support atomic transactions. However, the local database may not know where to send the response.

4. If either the communication link is dead or the machine is down, the GTM will eventually resume contact with the local database. At this point, the GTM can try to abort or compensate for T again. This time, it should succeed.

Note also that the GTM can continue to try alternative actions even if some local transaction T is in an inconsistent state due to failure. Once the GTM 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. Because the local transactions commit immediately, compensation-based recovery must be supported. This is true both for the case when an individual local transaction is invalidated, and when the global transaction is aborted.

## 5.5 Committing a Global Transaction

A global transaction commits only when all current state tokens are in the same final state set. Furthermore, there should be one token per final state in the final state set, and no state in the final state set should not be a current state. At this point, all actions have previously committed in their corresponding local databases. When a global transaction commits, the log is updated to reflect the termination of the global transaction and its associated GTS is removed from the set of active GTSs . At this point no information about the transaction including compensating transactions needs to be kept since once the global transaction commits the transaction in 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. Since these dependencies are seemed to be unimportant in this implementation, the commit takes effect immediately, regardless of what may happen to the other global transactions in the future. This may cause inconsistent or incorrect information in the database some transaction on which the committed global transaction is dependent on later aborts.

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. The effect of the aborted transaction does not cause severe problems in this case.

### 5.6 Aborting Global Transactions and Actions

An abort of an entire global transaction is effected by aborting all of the actions associated with that transaction. A similar approach to this kind of abort is taken in nested sagas [GGKKS90]. Refer to [NT91] for details on failure and recovery.

## 6 Conclusions

This work has addressed the problem of transaction specification and synchronization in heterogeneous multidatabases according to the whole transaction model. We believe that the whole transaction model accesses and manipulates data in much the same why as humans do in everyday life. The whole transaction model differentiates itself from other transaction models in the heterogeneous multidatabase domain in that it does not assume that global transactions are atomic or serializable. Rather, global transactions are series of transactions on a set of local databases which conform to the correctness criteria specified by the collection of GTSs for all concurrently executing global transactions.

Transaction specification is accomplished by defining a GTS for each global transaction. Together the global transaction specifications determine how local transactions can work together to accomplish tasks which span the local databases that make up the heterogeneous multidatabase. A GTS may contain conflict arcs governing which transactions can not occur simultaneously.

Both a visual and written transaction specification language has been developed to facilitate the specification and definition of global transactions. The latter has been implemented and has proved to be an efficient tool with which to define global transactions specifications. Future work in creating a graphical interface to this tool consistent with the visual language discussed in Section 4.1 should be considered.

One of the goals of the whole transaction model is to maximize the autonomy of the the local database, thus no changes are made to the local transaction management systems to facilitate synchronization. Instead the global transaction manager submits local transactions to the local transaction manager and receives a notification of whether the transaction has successed or failed. A global transaction manager has been implemented to perform according to this model. Shortcomings and suggested improvements to the implementation are discussed in more detail in Appendix B.

# A Data Sturctures, Algorithms and Pseudocode

## A.1 Data Structures

This section contains the data structures used in the prototype. These structures are initially allocated and set up when the GTS is parsed with the language specification. They are maintained by global transaction manager.

```
struct arc_list {
    struct arc_list *next;
    int order;
    struct arc *arc_ptr;
};
struct pattern {
    struct pattern *next;
    char name[32];
    int *global_tid;
    struct node_list *start_state;
    struct node_list *final_state[10];
    struct node *nodes;
    struct arc *arcs;
} ;
struct arc {
    struct arc *next;
    int alt_pri;
    struct node_list *head_states;
    struct node_list *tail_states;
    int action_type;
    union {
        struct local *lt;
        char pattern[32];
        char local_var[32];
    } action;
};
```

```
struct arg_list {
    struct arg_list *next;
    char type[32];
    char value[32];
};
struct node_list {
    struct node_list *next;
    struct node *node_ptr;
};
struct local {
    struct local *next;
    int order;
    char proc[32];
    struct arg_list *arguments;
};
struct node {
    struct node *next;
    char name[32];
    int current;
    int start;
    int final;
    struct arc_list *out_arc;
    struct arc_list *conflict;
    char entrance[32];
    struct arg_list *entrance_args;
    char exit[32];
    struct arg_list *exit_args;
    char *constraints;
};
```

## A.2 Code for Database Server

Following is the actual code for the communications portion of the server. The server and clients communicate using an RPC style interface. Ideally the server will listen for two types of requests, either *specify_pattern* which parses a new GTS and stores the associated structures in the system or *execute_gt* which actually executes the local transction.

```
/*************************************************************************/
/* Main:  This just starts the server up        */
/*************************************************************************/

main() {

   specify_pattern("pat.temp");                /* specify intial patterns */
   THREADgo (1,0,setup_connect,0,0,30*1024,8);
}

/*************************************************************************/
/* This routine sets up a TCP connection and gets a name and a port so
        connections can be made to the server.  The server name and port
        are written to a file so that clients can connect to server.
        (we assume all clients know the location of this file).

        The socket listens for connect requests and child thread processes
        are forked to receive the messages                                */
/*************************************************************************/
void setup_connect()
{  int sock,msgsock;
   struct hostent *h;
   int len,rval;
   char h_name[32];
   FILE *fp;
   THREAD child_proc;

   /* establish an internet socket */

   if ((sock = socket(AF_INET,SOCK_STREAM,0))== -1) {
      perror("opening tcp socket");
      exit(1);
   };

   /* get a name so we can accept messages */

   name.sin_family=AF_INET;
   name.sin_addr.s_addr=htonl(INADDR_ANY);
   name.sin_port=htons(0);        /* convert to network byte order*/
```

```
  if (bind(sock,&name,sizeof(name))== -1){
     perror("bind");
     exit(1);
  };

  /* what is our name, get the host name and the port number which
     we were assigned  */

  gethostname(h_name,sizeof(h_name));
  printf("host %s is running\n",h_name);
  len = sizeof(name);
  if (getsockname(sock,&name,&len) == -1) {
     perror("getsockname");
     exit(1);
  }
  /* write this information to file (or database)  so client knows
        how to get to server */

  fp=fopen("server_db","w");
  fprintf(fp,"%d %d %s %s\n",htons(name.sin_port),1,"gtm",h_name);
  fclose(fp);


  /* set up monitors for threads */

 io_mon=THREADmonitorinit(0,NULL);
  conflict_mon=THREADmonitorinit(0,NULL);

 /* start accepting connections, fork thread to handle connections */

  listen(sock,5);
  do{
     msgsock = accept(sock,(struct sockaddr *)0, (int *)0);
     if (msgsock == -1) perror("accept");
     else THREADcreate(receive_data,msgsock,0,0,30*1024,16);
  }while (1);

  /* when all  are finished we synch up and close socket */

  while (child_proc=THREADwaitforchild())
      THREADeliminatechild(child_proc);
  close(sock);
}
\newpage
/*************************************************************************/
```

```
/* This routine reads from the accepted socket, unpacks the message which
   contains either:
            0 - indicating a request to add patterns to system
           in which case the remainder of the message is the
        file name containing the pattern specifications

                      or

        1 - indicating a request to run a global transaction
            in which case the remainder of the message is the
         name of the pattern which represents the global
         transaction to be run by the system

    Upon return from this call the return value if any is ready to be sent
    back to client and the socket is closed                                  */
/*************************************************************************/
int receive_data(sock)
int sock;

{
   char buffer[1028],string[1024],message[1024];
   short service,ret;
   int rval;
   extern struct pattern *first_pattern;

   bzero(buffer,sizeof(buffer));
   if ((rval = read(sock,buffer,1028)) < 0)
      perror("reading message");
   else {
      memcpy(&service,buffer,2);
      service=ntohs(service);
      memcpy(string,&buffer[2],1024);
   switch (service) {
      case 0:                                  /*  specify pattern  */
         first_pattern=specify_pattern(string);
            ret=htons(1);
         memcpy(&ret,message,2);
         break;
      case 1:                                 /* execute global transaction*/
            ret=execute_gt(string,0);
            ret=htons(ret);
            printf("status = %d\n",ret);
            memcpy(&message[0],&ret,2);
            break;
    }
```

```
    write(sock,message,1024);
   };
   close(sock);
}
```

## A.3 Code for Client

Each machine accesses the GTM through a client procedure using an RPC type interface. This procedure is outlined below. A user of the client will just need to make a procedure call to either *execute_gt* to run a global transaction or *specify_pattern* to add a GTS to the system.

These two procedure calls are given below:

```
int execute_gt(pattern_name,args,queue)
   char *pattern_name;
   struct arg_list *args;
   int queue;

struct pattern *specify_pattern(spec)
   char *spec;   /*name of ascii file containing GTS */
```

The following is the code for the client stubs for the *specify_pattern* and *execute_gt* remote procedures. The the code for the communication portion of the client is also provided below.

```
/*************************************************************************/
/* client stub for specify pattern procedure                             */
/*************************************************************************/
specify_pattern(s)
char s[32] ;
{
   int c,i,stat;
   short port;
   char mess[1032],ret_buf[1024],temp[100];
   char host[32];

   /* find server, package arguments to send, call client side of RPC,
      unpackage arguments upon return from RPC */

   port=find_server("gtm",host);
   if (port == 0){
      printf("server_not_available\n");
      return;
   }

   stat=rpc$client(host,port,s,ret_buf,0,1);
```

```
   if (stat > 0)
      printf("pattern specified --%s\n",s);

}

/*********************************************************************/
/* client stub for executing a global transaction                    */
/*********************************************************************/
execute_gt(s)
char *s ;
{
   int c,i,stat;
   short port;
   char mess[1032],ret_buf[1024],temp[100];
   char host[32];

   /* find server, package arguments to send, call client side of RPC,
       unpackage arguments upon return from RPC */

   port=find_server("gtm",host);
   if (port == 0){
      printf("server_not_available\n");
      return;
   }

   stat=rpc$client(host,port,s,ret_buf,1,1);
   if (stat > 0)
      printf("--%s , global transaction executed\n",s);
}


/*********************************************************************/
/* rpc$client performs the client side RPC function.  This includes setting
        up a TCP socket, creating the connection and packaging the
        service number and return flag  associated with the call at the
        beginning of message passed to the server.  This procedure waits
        a return value if one is expected.                            */
/*********************************************************************/
rpc$client(host,port,message,return_b,service,ret)
char *host;
short port;
char *message,*return_b;
short service;
short ret;
{
```

```
    int sock;
    struct sockaddr_in name;
    struct hostent *h;
    int x;
    int rval;
    char buffer[1024];
    char temp[1028];
    fd_set mask;
    struct timeval to;
    short stat;

/* establish an internet socket */

    if ((sock = socket(AF_INET,SOCK_STREAM,0))== -1) {
       perror("opening tcp socket");
       exit(1);
    };

    /* set up to  connect to server */

    h=gethostbyname(host);
    if (h == NULL)  printf("host unknown\n");
    bcopy(h->h_addr,&name.sin_addr,h->h_length);
    name.sin_family=AF_INET;
    name.sin_port=port;

    /* make the connection */

    if (connect(sock,(struct sockaddr *)&name,sizeof(name)) < 0) {
         perror("establishing connect");
         exit(1);
    }

    /* send the message by writing to socket, append number of program
       and return flag at the beginning of the buffer */

    service=htons(service);
    memcpy(temp,&service,2);
    memcpy(&temp[2],message,1024);
    message=temp;

    if (write(sock,message,1032) < 0) perror("writing on stream socket");

    /* setup for select and timeout */
```

```
   FD_ZERO  (&mask);
   FD_SET (sock,&mask);
   to.tv_sec = 600;

   bzero(buffer,sizeof(buffer));
   while (stat= select (sock+1,&mask,NULL,NULL, &to) > 0){
      if ((rval = read(sock,buffer,1024)) < 0)
         perror("reading message");
      else {
         memcpy(&ret,&buffer[0],2);
         printf("status = %d\n",ret);
         return(ret);

      }
   }

   if (stat== 0)
   printf("time expired, no longer awaiting reponse\n");

   close(sock);


}
```

## A.4 Code for parsing GTS

This section contains interface to a compiler which will parse a GTS and create the appropriate data stuctures. The compiler itself is written using *YACC* and *LEX*. Code is not provided here for the compiler, but can be found in *u/pat/thesis/proto*. The parsing code is simliar to the BNF in Figure 8.

```
struct pattern *specify_pattern(spec)
char *spec;    /* name of ascii file containing GTS */

{
   extern int error_flag;
   extern struct pattern *first_pattern;
   int sts;
   THREAD_MANAGER_BLOCK manager;

   THREADmonitorentry(io_mon,&manager);
   freopen(spec,"r",stdin);
   error_flag = FALSE;
   sts=yyparse();
   THREADmonitorexit(io_mon,&manager);
```

```
   if ( sts == TRUE )
     printf("error\n");       /* this error indicates error getting into
  parser routine */
   else if (error_flag == FALSE){  /* if we get here and error_flag = TRUE */
     /* a more comprehensive error will have
      already been returned by parser */
   printf("finished parsing pattern\n");
   return(first_pattern);
   }

}
```

## A.5 Code for Global Transaction Manager

This section contains the code associated with the global transaction manager used in the prototype. This includes all syncrhonization and conflict checking code.

```
/*************************************************************************/
/* execute_gt: This routine handles the execution of a global transaction.
               In this implementation conflicts are refused, but we
            provide for the queue or refuse option for future
            implementations                              */
/*************************************************************************/

int execute_gt(pattern_name,gtid,args,queue)
char *pattern_name;
int gtid;
struct arg_list *args;
int queue;

{  struct arc *arc_temp;
   struct arc_list *temp_arc,*last_arc,*checked,*new_arc;
   struct node_list *node_temp,*temp,*tempa,*c_states,*new,*last;
   struct pattern *c_pattern;
   int finished,status;
   THREAD_MANAGER_BLOCK manager;

   /* find pattern in known pattern list */

   c_states = NULL;
   c_pattern=find_pattern(pattern_name);
   if (c_pattern == NULL)
      return(ABORT);
```

```
/* assign Global transaction id, if not a nested transaction */

 if (gtid==0) {
    THREADmonitorentry(gtid_mon,&manager);
    c_pattern->global_tid=tid_ct++;
    gtid=c_pattern->global_tid;
    THREADmonitorexit(gtid_mon);
 }

 /* for each node in the start state set set add to current */

 temp=c_pattern->start_state;
 while (temp != NULL) {
    c_states=add_to_current(temp,c_states);
    temp=temp->next;
 };

/* while current state does not equal any final state sets traverse
   arcs in graph structure, serially */

while (finished=check_states(c_states,NULL,c_pattern->final_state,TRUE)
        ==FALSE){
    status = FALSE;
    while (status == FALSE){
    arc_temp = find_highest_arc(c_states,checked);
       if (arc_temp == NULL) return (ABORT);
       new_arc = (struct arc_list *) calloc (1,sizeof(struct arc_list));
    new_arc->arc_ptr=arc_temp;
    if (checked == NULL)  checked=new_arc;
    else {
       temp_arc=checked;
        while (temp_arc!= NULL){
           last_arc=temp_arc;
           temp_arc=temp_arc->next;
        };
        last_arc->next=new_arc;
    };

       /* find out what type of arc we and try to traverse it */

     if (arc_temp->action_type == PAT)
       status = execute_gt(arc_temp->action.pattern,gtid);
    else
       status = execute_action(arc_temp,c_states,gtid);
       if (status) {
```

```
            checked = NULL;
          tempa=arc_temp->tail_states;
              while (tempa != NULL ) {
                 c_states=remove_from_current(tempa->node_ptr->name,c_states);
                 tempa=tempa->next;
           }
              tempa=arc_temp->head_states;
              while (tempa != NULL) {        /* add head nodes to current*/
                 c_states=add_to_current(tempa,c_states);
                 tempa=tempa->next;
              };
           };
       };
    };

    return (status);
}

/*************************************************************************/
/* execute_action: searchs a list of local transactions and tries to find
                   the highest priority non-conflicting action to execute.
              When a conflict occurs the GTM refuses to execute
              request and search for next conflicting action.  This
                   may
              change in future implementations by setting queue flag
              to true when calling the check_conflict routine.    */
/*************************************************************************/
execute_action(arc_temp,c_states,gtid)
struct arc *arc_temp;
struct node_list *c_states;
int gtid;   tempa=c_states;

{

    struct local *temp;
    int num,status,stat=FALSE;
    int getid,id;
    char string[32];
    int (*comp)();
    struct timeval time_out;
    THREAD_MANAGER_BLOCK manager;

    temp=arc_temp->action.lt;

    while (temp != NULL && stat == FALSE) {
```

```
      if (check_conflict(gtid,temp->proc) == FALSE ) {
         system(temp->proc);
         time_out.tv_sec=2;
         time_out.tv_usec=0;
         select(NULL,NULL,NULL,NULL,&time_out);
       stat = TRUE;
      }
      temp=temp->next;
   };
   return (stat);
}

/*************************************************************************/
/* add_to_current: add node to current state state set, add conflict to
                   global conflcit list & execute entrance computation */
/*************************************************************************/

struct node_list *add_to_current(np,c_states)
struct node_list *np,*c_states;
{
   struct node_list *new,*last,*temp;
   struct arc_list *last_conflict,*temp_conflict;
   THREAD_MANAGER_BLOCK manager;

   /* set current state to true and add node to current state list */

   np->node_ptr->current = TRUE;
   new = (struct node_list *) calloc (1,sizeof(struct node_list ) );
   new->node_ptr= np->node_ptr;
   if (c_states == NULL)
      c_states = new;
   else {
      temp = c_states;
      while ( temp!= NULL) {
       last=temp;
       temp=temp->next;
      };
   last->next=new;

   if (np->node_ptr->conflict != NULL) {

      /* set gtid flag for conflicts in list */

      temp_conflict = np->node_ptr->conflict;
      while (temp_conflict != NULL){
```

```
            np->node_ptr->conflict->arc_ptr->gtid=gtid;
            temp_conflict=temp_conflict->next;
         };

         /* conflicts to end of global conflict list */

         THREADmonitorentry(conflict_mon,&manager);
         temp_conflict=global_conflict;
         last_conflict=global_conflict;
         while (temp_conflict != NULL) {
            temp_conflict=temp_conflict->next;
            last_conflict=temp_conflict;
         }
         if (last_conflict == NULL)
            global_conflict=np->node_ptr->conflict;
         else
            last_conflict->next=np->node_ptr->conflict;
         THREADmonitorexit(conflict_mon);
      };

};

/*************************************************************************/
/* remove_from_current: remove node from current  state set and execute
                        exit computations     */
/*************************************************************************/
struct node_list *remove_from_current(string,c_states)
struct node_list *c_states;
char *string;
{

   struct node_list *temp,*last,*next;
   struct arc_list *last_conflict,*temp_conflict;
   THREAD_MANAGER_BLOCK manager;
   int found = FALSE;

   /* find node in current state list */

   temp=c_states;
   last=NULL;
   while (temp != NULL && found == FALSE) {
      if (strcmp(temp->node_ptr->name,string) == 0) {
         found == TRUE;
         temp->node_ptr->current = FALSE;
         if (last == NULL)
```

```
            c_states = temp->next;
         else {
            last->next=temp->next;
            free (temp);
            return (c_states);
         }
      else {
         last=temp;
         temp=temp->next;
      };

      /* remove conflicts from global conflict list */

      if (temp->node_ptr->conflict != NULL) {
         THREADmonitorentry(conflict_mon,&manager);
         temp_conflict=global_conflict;
         last_conflict=global_conflict;
         while (temp_conflict != NULL) {
            if (gtid == temp_conflict->arc_ptr->gtid)
                if (last_conflict != global_conflict)
                   last_conflict->next=temp_conflict->next;
                else
                   global_conflict=NULL;
            temp_conflict=temp_conflict->next;
            last_conflict=temp_conflict;
            };
         THREADmonitorexit(conflict_mon);
      };
   };
   if (found == FALSE)
      printf("ERROR: should never not be able to remove from current\n");
   return (c_states);
}
/*************************************************************************/
/* find_pattern: find requested global transaction in list of
                 patterns     */
/*************************************************************************/

struct pattern *find_pattern(pattern_name)
char *pattern_name;

{
   extern struct pattern *first_pattern;
   struct pattern *tp;
   int found=FALSE;
```

```
   tp=first_pattern;
   while (tp != NULL && found == FALSE ){
      if (strcmp(tp->name,pattern_name) == 0) {
         found = TRUE;
         return (tp);
      }
      else
         tp=tp->next;
   };
   printf ("ERROR: Global Transaction %s not specified\n",pattern_name);
   return (NULL);

}

/*************************************************************************/
/* check_conflict: this routine checks to see there are conflicts with
                        execution of this local transaction     */
/*************************************************************************/

int check_conflict(gtid,string)
int gtid;
char *string;

{
   struct local *temp2;
   struct arc_list *temp_conflict;
   THREAD_MANAGER_BLOCK manager;

   THREADmonitorentry(conflict_mon,&manager);
   temp_conflict=global_conflict;
   while (temp_conflict != NULL) {
      temp2=temp_conflict->arc_ptr->action.lt;
      while (temp2!=NULL) {
         if (gtid!=temp_conflict->arc_ptr->gtid &&
             strcmp(string,temp2->proc)==0){
               printf("gt %d conflicts with gt %d on lt %s\n",
                      gtid,temp_conflict->arc_ptr->gtid,string);
               THREADmonitorexit(conflict_mon);
               return(TRUE);
         };
      temp2=temp2->next;
      };
      temp_conflict=temp_conflict->next;
    };
```

```
    THREADmonitorexit(conflict_mon);


    return (FALSE);
}

/*************************************************************************/
/* check_states: this routine checks  that all head_states are
                     part of current state set if the flag is set to
               FALSE otherwise check if all the current states
               are part of a final state set     */
/*************************************************************************/

int check_states(c_states,q_states,f_states,flag)
struct node_list *c_states,*q_states;
struct node_list *f_states[];
int flag;  /* if flag is true we are looking for final state sets */
{

   struct node_list *temp, *temp_current;
   int found,array = 0;

   if (flag == 0)
      temp = q_states;
   else
      temp = f_states[0];
   while (temp != NULL) {
      found=FALSE;
      temp_current = c_states;
      while (temp_current != NULL &&  found == FALSE) {
         if (strcmp(temp_current->node_ptr->name,temp->node_ptr->name)==0)
            found=TRUE;
         else
            temp_current=temp_current->next;
      };
      if (found == FALSE) {
         if (flag == 0)
            return (FALSE);
         else {
            temp=f_states[++array];
            if (temp == NULL) return (FALSE);
 }
      }
      else
         temp=temp->next;
```

```
   };
   return (TRUE);
}

/*************************************************************************/
/* find_highest_arc: this routine finds the highest priority arc in the
                     current state set which is ready for traversal.
                     An arc
                     must have all nodes in the head_state set in the
               current state set to be ready for traversal. If
               more then one highest priority arc use the first one
               found.  Once a priority one arc is found return it
/*************************************************************************/

struct arc *find_highest_arc(c_states,checked)

struct node_list *c_states;
struct arc_list *checked;
{
   struct node_list *tp;
   struct arc *tp_arc,*highest;
   int found=FALSE;

   tp=c_states;
   highest = NULL;
   while (tp != NULL) {  /* for each node in current state set*/
      tp_arc=tp->node_ptr->out_arc->arc_ptr;
      while (tp_arc != NULL) {
         if (check_states(c_states,tp_arc->tail_states,NULL,FALSE)== TRUE)  {
            if (notchecked (checked,tp_arc)) {
               if (highest == NULL ||highest->alt_pri > tp_arc->alt_pri)
                  highest=tp_arc;
               if (highest->alt_pri== 1)
                   return (highest);
            }
         }
         tp_arc=tp_arc->next;
      }
      tp=tp->next; /* look at next node */
   };
   if (highest == NULL) /* return highest alternative or null*/
      printf ("Can currently traverse no arcs\n");
   return (highest);

}
```

# B The Prototype

## B.1 Tips on Using the Prototype

The code associated with the prototype resides on the Sun system in *u/pat/thesis/proto* on February 10th, 1991. This directory contains the source code, makefiles and all other necessary files. The executable image *server* can be run on any system. This will start up the server and write a file to in the default directory to announce its availability and give the port of the server. The server can handle several clients concurrently.

Client software can be executed from any machine in the system and from remote machines which can access the server's port file. A user writes a C routine which contains calls to the remote procedures *specify_pattern* and *execute_gt*. These calls are expected to be in the following form:

```
int execute_gt(pattern_name,args,queue)
   char *pattern_name;
   struct arg_list *args;
   int queue;

struct pattern *specify_pattern(spec)
   char *spec;   /* name of ascii file containing GTS */
```

The user must then compile their client code with the client.c code which resides in the directory */u/pat/thesis/proto.*

*specify_pattern* takes the path name of an ascii file containing the GTS as input. Note, standard input is redirected to this file while the pattern is being specified.

In the prototype, local transactions are not stored in a transaction library as specified in the model. Instead, local transactions are executable files which are expected to be in the same directory as the server executable. When a user writes a local transaction, this transaction must be compiled and linked with any relevant librarys. The executable image is then place in the same directory as the server, this directory was chosen for convenience in testing this prototype. If one wants to have all the executables in a different directory then just substitute the name of the new path in the *gt.c* procedure. The same is true of entrance and exit procedures. One might want to have a specified directory for local transactions and one for entrance and exit computations.

## B.2 Example

In this section, several examples of global transaction execute as performed by the prototype are given. These examples use the *plane* and *trip* GTSs as defined in Section 4.

### B.2.1 Two concurrent global transactions, no conflicts

Global transaction *plane* and *trip* executing concurrently. Both global transactions commit.

```
accepted connect from client1
client1 requesting plane
starting on GTS plane gtid = 11
about to call inquire gtid = 11 from nodes Aplane
  ##executing local transaction inquire
accepted connect from client2
client2 requesting trip
starting on GTS trip gtid = 12
starting on GTS plane gtid = 12
about to call inquire gtid = 12 from nodes Aplane
  ##executing local transaction inquire
about to call planeres gtid = 11 from nodes Bplane
  ##executing local transaction plane
about to call planeres gtid = 12 from nodes Bplane
  ##executing local transaction plane
about to call agent gtid = 11 from nodes Cplane
  ##executing local transaction agent
about to call agent gtid = 12 from nodes Cplane
  ##executing local transaction agent
final state Dplane gtid = 11
COMMITTED global transaction plane gtid = 11
final state Dplane gtid = 12
COMMITTED global transaction plane gtid = 12
about to call hotel gtid = 12 from nodes Btrip
  ##executing local transaction hotel
about to call car gtid = 12 from nodes Ctrip
  ##executing local transaction car
about to call agent gtid = 12 from nodes Etrip,Ftrip
  ##executing local transaction agent
final state Gtrip gtid = 12
COMMITTED global transaction trip gtid = 12
```

### B.2.2 Two concurrent global transactions, non-fatal conflict

Global transaction *test* and *trip* executing concurrently. *test* is similar to *plane*, but contains a conflict from node Atest with the inquire action. This forces the *trip* GTS to use alternative action car instead of the nested GTS *plane*. Both global transactions commit.

```
accepted connect from client1
client1 requesting test
starting on GTS test gtid = 1
```

```
about to call car gtid = 1 from nodes Atest
accepted connect from client2
client2 requesting trip
starting on GTS trip gtid = 2
starting on GTS plane gtid = 2
**->gt 2 conflicts with gt 1 on local transaction inquire
gtid 2 can currently traverse no arcs in GTS plane
about to call car gtid = 2 from nodes Atrip
  ##executing local transaction car
about to call planeres gtid = 1 from nodes Btest
  ##executing local transaction plane
  ##executing local transaction car
about to call agent gtid = 1 from nodes Ctest
  ##executing local transaction agent
about to call hotel gtid = 2 from nodes Htrip
  ##executing local transaction hotel
final state Dtest
COMMITTED global transaction test gtid = 1
about to call agent gtid = 2 from nodes Itrip
  ##executing local transaction agent
final state Gtrip
COMMITTED global transaction trip gtid = 2
```

### B.2.3 Two concurrent global transactions, fatal conflict

Global transaction *plane* and *trip* executing concurrently. A conflict occurs on local transaction agent. This causes *plane* to be aborted since all conflicts are refused and there is no alternative actions or local transactions for the vital action *inquire*.

```
accepted connect from client1
client 1 requesting trip gtid = 3
starting on GTS trip gtid = 3
starting on GTS plane gtid = 3
about to call inquire gtid = 3 from nodes Aplane
  ##executing local transaction inquire
about to call planeres gtid = 3 from nodes Bplane
  ##executing local transaction plane
about to call agent gtid = 3 from nodes Cplane
  ##executing local transaction agent
final state Dplane
COMMITTED global transaction plane gtid = 3
about to call hotel gtid = 3 from nodes Btrip
  ##executing local transaction hotel
about to call car gtid = 3 from nodes Ctrip
accepted connect from client2
```

```
client2 requesting plane
starting on GTS plane gtid = 4
about to call inquire gtid = 4 from nodes Aplane
  ##executing local transaction inquire
about to call planeres gtid = 4 from nodes Bplane
  ##executing local transaction plane
  ##executing local transaction car
**->gt 4 conflicts with gt 3 on local transaction agent
gtid 4 Can currently traverse no arcs in GTS plane
ABORTED global transaction plane gtid = 4
about to call agent gtid = 3 from nodes Etrip,Ftrip
  ##executing local transaction agent
final state G
COMMITTED global transaction trip gtid = 3
```

## B.3 Unresolved Problems and Suggested Enhancements

The prototype does not support interactive specification of patterns as originally planned. All the code to support this is available, but there is a problem with using the *LEX* and *YACC* code with Brown threads. Brown threads requires the use of *"/pro/threads/thread.h"* library instead of the *stdio.h*, the standard c library. *LEX* somehow imports the standard library. Attempts to override this have thus far failed.

Local transactions currently only return information regarding whether the transaction committed or aborted. The *system()* call is used to fork a process which executes the local transaction. This mechanism does not allow for return values. Therefore two alternatives exist to deal handle this return information. We could either use a message passing system or return it to some global return area. The message passing system is the preferred alternative, but this requires that each local transaction send a message to the GTM with the designated return parameters. This could be done by have a standard piece of code which is appended to the local transaction code to set up the communications with the GTM and return the appropriate information. The server would also have to set up a socket to listen for responses. Future implementations should allow for more comprehensive information to be returned. This is due in part to the way local transactions are executed.

The prototype automatically refuses all conflicting local transaction. The next version should allow for queueing of conflicting actions. Note that this may result in deadlock, therefore a deadlock dectection algorithm will be needed.

Further concurrency could be added to the GTM, by running the actions associated with the fork arcs in separate threads. Additionally if several alternatives were to be attempted in parallel, these would also be executed from separate threads.

# References


[AGMS87] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency control and recovery for global procedures in federated database systems. *IEEE Data Engineering Bulletin*, 1987.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[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 *ACM Sigmod Proceedings*, pages 215–224, 1990.

[DE89] Weimin Du and Ahmed K. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency in interbase. In *VLDB Proceedings*, pages 347–335, 1989.

[DE90] Weimin Du and Ahmed K. Elmagarmid. A paradigm for concurrency control in heterogenous distributed database systems. In *Proceedings of the Sixth International Conference on Data Engineering*, February 1990.

[Doe87] Tom Doeppner. A threads tutorial. Technical Report CS-87-06, Brown University, March 1987.

[EH88] A. Elmagarmid and A. Helal. Supporting updates in heterogeneous distributed database systems. In *Proceedings of the International 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.

[GMGK+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.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *ACM SIGMOD Proceedings*, pages 249–259, 1987.

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

[HM85] Dennis Heimbingner and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Office Automation Systems*, 3(3):253–278, July 1985.

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

[KS88] Henry F. Korth and Gregory D. Speegle. Formal model of correctness without serializability. In *ACM SIGMOD Proceedings*, pages 379–386, 1988.

[Lyn83] Nancy A. Lynch. Multilevel atomicty - a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4), December 1983.

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

[NSZ90] Marian H. Nodine, Andrea H. Skarra, and Stanley B. Zdonik. Synchronization and recovery in cooperative transactions. In *Proceedings of the 4th International Workshop on Persisitent Object Systems Design, Implementation, and Use*, 1990.

[NT91] Marian H. Nodine and Patrice A. Tegan. Coordinating multi-transaction activities. Technical Report In preparation, Brown University, Department of Computer Science, 1991.

[SBD+81] John Miles Smith, Philip A. Bernstein, Umeshwar Dayal, Nathan Goodman, Terry Landers, Ken W. T. Lin, and Eugene Wong. Multibase – integrating heterogeneous distributed database systems. In *Proceedings of the National Computer Conference*, pages 487–499, 1981.