Interactions: Multidatabase Support for Planning Applications

Marian H. Nodine

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-17
May 1993

Interactions: Multidatabase Support for Planning Applications

Marian H. Nodine
Brown University
Providence, RI 02912

May 13, 1993

Abstract

A planning task's purpose is to determine, given information about the world, how to reach some particular goal in that world. In this thesis, we examine the ways that a planning task interacts with information in different databases. These databases are grouped together into a single logical entity, a *multidatabase*.

A planning task lasts from the time the goals are initially decided on to the time the execution of the plan is complete. During this time, which may be very long, the state of the world as represented in the multidatabase may evolve. If this evolution affects the validity of the plan (as developed so far), then the plan itself may need to be modified so that it is valid given the new state of the multidatabase.

We propose a new transaction model, called *Interactions*, and an architecture to support them. This model and architecture provide the following capabilities to the planner.

1. Define different possibilities for how to accomplish the goal and the relative acceptability of each possibility.
2. Define conditions that are set by specific subtasks and which must be maintained for the plan to remain valid.
3. Detect when a condition is violated, indicating the plan is no longer valid.
4. React to the condition violation by modifying the current plan into one that is valid given the current multidatabase state.

.

This thesis has two major themes. The first is theoretical: we provide a formal definition for Interactions. Basically, an Interaction is an open nested transaction, i.e., a partial order of atomic multidatabase transactions. It is a flexible model in that different plans cause different sets of global transactions to be executed. It is reactive in that it can automatically replan when some change in the multidatabase invalidates the current plan. We give an underlying theory about correctness for atomic multidatabase transactions (which are the basic building blocks for Interactions), and for Interactions both independently and in situations where it must react to some change in the multidatabase. We provide scheduling and synchronization algorithms that ensure both that Interactions get executed correctly and that changes as little completed work done by the Interaction as possible given the different changes in the multidatabase.

The second theme of this thesis is a system design and implementation of a multidatabase that supports Interactions. We define a language, TaSL, that can be used to specify Interactions. We also describe the basic system design of our multidatabase system, Mongrel. Mongrel has been used to test the feasibility and practicality of Interactions as a multidatabase transaction model. Novel functions of the Mongrel system include an ability to check the multidatabase for specific changes that impact running Interactions. It also provides for the ability of each local database in the multidatabase to define its interface to the multidatabase, and to use this interface to automatically generate semantic undo steps in the local database when needed (e.g. in backing out of some invalid plan).

Chapter 1

Introduction

1.1 What is a Multidatabase?

A *multidatabase* is a time-varying collection of autonomous *local databases* containing information that may be queried and/or updated together. Multidatabases are created when it becomes useful to access related information that is already stored in different databases via some *uniform interface*. It may be impractical to copy all of the data into a single database. For example, if the data in the databases were managed by different enterprises, copying it into a single database would be out of the question.

A multidatabase has two-levels. The *local level* consists of a set of autonomous local databases that actually hold the information in the multidatabase. It is the local level that ensures that the information in the multidatabase is *persistent*. The *global level* contains the modules that allow a user to access the information in the collection of local databases *as if they were using a single database*. Because the local databases are autonomous, the global level needs to operate without the explicit cooperation of the local level.

At the global level, a multidatabase needs to provide its users with a coherent means for defining and executing specific tasks on the information in its collection of local databases. In single databases, tasks are executed as atomic transactions. Multidatabases may need to support different mechanisms for task specification, and for insuring that the information in the multidatabase remains consistent.

Consider as an example of a multidatabase the set of databases used by a travel agent. The travel agent has his own private database that stores information about his specific customers and the trips he is currently working on. In addition to accessing the information in his own database, the travel agent also has access to airline databases such as United, TWA, and PanAm for making plane reservations. Similarly, he has access to hotel reservation databases, rental car databases, cruise databases, and many other repositories of information relating to the travel industry. These different databases are accessed as if they were a single data repository, the multidatabase, that has a global interface the travel agent can use in booking trips for his customers.

1.2 What is a Planning Task?

Long-lived applications that access one or more databases are difficult to support using traditional transaction models. Planning applications are used when people (or intelligent processes) are deciding the best way to accomplish some goal and taking steps to ensure that that way succeeds. An example of a planning application is that of a travel agent. A task for a travel agent might be making plans for a business trip. At the outset, the traveler knows that he must be in a particular city for some number of days. He also needs a hotel and some way to get around to the various meetings during his stay. He has no idea what the best and possibly the cheapest way to do all this is. In some sense, that is the point of the planning task.

Planning tasks tend to be long duration activities. Typically, a planning task begins when the ultimate goal is first envisioned and the plan begins to take place. As the plan is developed, reservations are made to ensure that the plan succeeds. Finally, the plan is executed. Since plans may change based on different

circumstances, a planning task typically does not end until the plan has actually been executed to completion. Furthermore, if we look at the structure of a planning task, we find that it typically consists of short activities separated by long periods of idle time. For instance, a traveler doesn't always make decisions about what hotel to stay in at the same time he makes his plane reservations.

In a planning task, we view the underlying database as a (hopefully accurate) representation of the state of the world. We expect the database to evolve as conditions change in the world. Because of the length of the planning task, we cannot expect the database to remain static from the point of view of the task. Rather, we need a model for interaction with the multidatabase that allows the planning task to *react* to the changes in the world and evolve its work so that it remains consistent with the evolving information in the database.

When changes happen in the outside world that affect a planning task, the reservations made by the task need to be updated to take those changes into account (*reactivity*). As they proceed, planning tasks accomplish subtasks, such as making different reservations in the databases. Later subtasks rely on some of the effects of earlier subtasks. A planning task can explicitly specify conditions on these effects that should continue to hold for its work to remain consistent, along with a specification of what could be invalid if those conditions no longer hold.

Suppose that we make an airline reservation as a part of our trip to San Francisco. We assume the flight information as we try to make a car reservation. Thus, the condition on the flight reservation is that the plane will fly at that specific time. If the flight is canceled, the reactive task determines what additionally is affected by the change, namely the car reservation. Thus, the car reservation is also reconsidered. Thus, our assumption is that a reactive task must protect itself from changes in its environment, but without explicitly preventing these changes from occurring by maintaining control over the data.

In this thesis, we assume that planning tasks are executed in a multidatabase using a two-level open nested transaction model (e.g., Sagas [GS87]). That is, a task executes a related set of atomic transactions, and commits when the plan has executed to completion. An open nested transaction differs from a traditional (closed) nested transaction (see [Mos85]) in that when a child transaction commits it releases its resources rather than passing them on to the parent transaction. Thus, the parent transaction is not atomic.

In classic transaction processing theory, transactions are kept isolated so that dependencies cannot form. We allow such dependencies to form because preventing them may not be desirable in the long-lived applications we support. Thus, there must be additional database support for dealing with work that depends on other work that has become invalid, and for examining alternative ways to accomplish any work that has become invalid. Note that these alternatives are already built in in the flexible task specification.

In this way, each task is able specify how to fix itself in the face of autonomous events. Then when a conflict occurs, recovering the consistency of the task can be automatic. Therefore, planning tasks may use different methods for maintaining their own consistency in the presence of conflict, in contrast to classic transactions where isolation (atomicity) is the only method that the database supports for ensuring consistency.

In this thesis, our approach to regaining consistency is primarily through semantic undoing inconsistent subtasks and then working forward. By semantic undoing (compensation) we mean running a different transaction that executes steps that reverse the effects of the transaction that executed the original subtask. By working forward we mean continuing the planning task from some recovered consistent internal state.

1.3 Planning Task Examples

Examples of planning applications abound in the real world, including the travel agent example (which is widely discussed in multidatabase transaction research), as well as Artificial Intelligence planning applications for military and other operations. This section expands on the travel agent example by defining several scenarios for booking travel plans. We hope to give a flavor for the different expectations people have with respect to a planning application. These illustrations are used to help us to understand the requirements that a planning application places on an underlying multidatabase.

1.3.1 Standard Trip Plan

In the first example, a travel agent is booking a business trip for a customer. This requires booking a round-trip plane flight, a hotel and a rental car. When first planning the trip, the travel agent does the

following:

1. Gets the constraints on the times of the flight and the return flight from the customer, and places them in the travel agent database.
2. Using these constraints, books the flights. If several flights are available, the customer's preferences determine which airlines to try first.
3. Notes the flight information in the travel agent database.
4. Books a hotel once the arrival time is known. The customer may have preferences concerning the type of hotel to book, and these preferences should be honored.
5. Notes the hotel information in the travel agent database.
6. Books the rental car once the arrival time is known. This also notifies the car rental agency when the car should be available. If no car is available, this step is optional. Note that this can be done concurrently with the step that books the hotel.
7. Notes the car rental information in the travel agent database.
8. Sends the itinerary to the customer.

Later, the customer pays for the tickets and the travel agent issues them to the customer. The customer can then take his trip and everything goes smoothly.

1.3.2 Trip Plan with Budget

A second example is very similar to the standard trip plan except that the traveler has a budget for the trip, and the cost of the whole trip is required to be within the budget. In this case, the budget is kept internally to the trip, and the amount of the budget is decremented appropriately when each reservation is made. Thus, the steps in the previous example that booked the flights, the hotel and the rental car would both require access to this single, internal variable. Also, since the potentially concurrent steps of booking the hotel and the car both also decrement the budget, these two steps will be forced into a specific nonconcurrent execution order during the planning of the trip. This is because of concurrency control restrictions on the internal variables, which need to be managed as if they are data items in some local database.

1.3.3 Trip Plan with Flight Cancellation

The third example is just like the trip plan, with the exception that the flight is canceled after the itinerary is sent to the customer, but before the day of the trip. The plans for the trip are now invalid because some external condition that the plans depended on (the existence of the flight) has changed. Since these changes occur outside the scope of the trip plan, and possibly are done by some transaction running independently of the multidatabase, it is the multidatabase itself rather than the planning task that is in the best position to notice the change. The multidatabase must then prod the travel agent application so that it can *react* by replanning the invalid part of the task and making the plan consistent again.

In this case, the traveler needs to modify the existing trip plan to work around the flight cancellation rather than aborting the trip plan and starting over. This means doing the following:

1. Making new, less desirable flight reservations.
2. Noting the flight reservation changes in the travel agent database.
3. Checking to ensure that the booking for the rental car is still OK. If not, adjusting the booking.
4. Noting any changes to the car rental arrangements in the travel agent database.
5. Concurrently with checking the car reservation, checking to ensure that the booking for the hotel is still OK. If not, adjusting the booking.

6. Noting any changes to the hotel arrangements in the travel agent database.

7. Notifying the customer of the changes.

8. Getting a refund for the canceled flight.

9. Paying for the new flight.

Note here that even though the car reservation and the hotel reservation logically require information from the flight reservation, they may not need to be adjusted (depending on how close the new flight times are to the times of the invalid flight).

If all the adjustments to the trip are possible, then the traveler can continue the trip as replanned. However, different things may go wrong during the replanning process; for instance, no alternative flight may be available. In this case, other alternatives must be tried. If no viable alternatives remain, the planning task must be aborted.

1.3.4 Trip Plan with Bankrupt Airline

This last example is similar to the previous one with the flight cancellation, except that the traveler is operating on a budget. The budget is managed as in the example in Section 1.3.2. Also, when the flight is canceled, the customer is unable to get a refund because the airline went bankrupt.

In this situation, two problems exist. First, the travel agent cannot get a refund on the ticket, but instead may have to take alternative actions such as filing a claim against the bankrupt airline to be considered when the case goes to court. This causes a second problem, that the trip may no longer fit in the customer's budget, because the budget has to cover both the invalid and the new flights.

1.4 What Makes Executing Planning Tasks on Multidatabases Difficult?

There are four basic characteristics of planning tasks on multidatabases that require special consideration. First, the local databases in the multidatabase are *autonomous*, and should not need to be modified to participate in the multidatabase. Second, planning tasks are *long-lived*. Third, since the optimal plan may not be known at the time the task is specified, the task specification and execution may need to be *flexible* and depend on the underlying state of the multidatabase. Last, since the information in the multidatabase may *evolve* as real-world situations change, the task needs to be able to *react* to changes that impact the validity of the plan.

The following sections examine each of these requirements in more detail, giving details as to what the problem is, why it exists, and what its impact is on an underlying transaction model.

1.4.1 Local Database Autonomy

In this work (as in most multidatabase work), we assume that the local databases are *autonomous* from the multidatabase. There are several standard types of autonomy that are recognized in the literature.

We assume that the local databases are *heterogeneous*, and therefore we support *design autonomy*. Design autonomy means that the different local databases may support different concurrency control protocols, data models and data manipulation languages, and that the multidatabase cannot require modification of the local database's code or functionality before it can participate in the multidatabase. This is particularly a requirement because multidatabases are usually formed when an organization requires uniform access to information scattered in a set of already existing databases.

One exception we do make to the design autonomy requirement is that we only use local databases that support ACID (atomic, consistent, isolated, durable) transactions and basic recovery from failure. This is because in our work we try to guarantee atomic global transactions at some level, and this is not possible without some atomicity guarantees in the local databases.

Execution autonomy means that the local databases can execute their transactions in any order. Otherwise, local database functions such as the concurrency control protocols would need to be modified. One

of the major problems this causes is that the local databases can execute other transactions besides those naturally generated by the multidatabase. These other transactions, which we call *independent transactions*, impact the multidatabase execution quite a bit. For instance, even if the multidatabase generates serial executions of its global transactions, the independent transactions may cause the local database to serialize the local parts of the different global subtransactions inconsistently; therefore the global execution may not, in fact, even be serializable. Another consequence of execution autonomy that must be considered is that the local databases cannot be expected to participate actively in any distributed agreement protocol such as the two-phase commit protocol used in distributed databases.

Control autonomy refers to the amount that the multidatabase needs to control the order and timing of the messages on the input stream to the local databases. Violating control autonomy does not require modification of the local databases themselves, but rather requires controlling the input to the local database. [SKS91a] notes that it is impossible to make certain guarantees about the atomicity of global transactions on a multidatabase without violating either control autonomy or execution autonomy.

The *communication autonomy* requirement states that the local databases need not be explicitly aware of the multidatabase or of the other local databases in the multidatabase. Thus, the multidatabase itself cannot assume that the local databases can communicate with each other during tasks such as coordinating a global commit action.

One less-discussed effect of the autonomy of the different enterprises that manage the different databases in the multidatabase is that an enterprise's access to its own data should probably take priority over the access by the planning applications. This is because it is the enterprise itself that is responsible for the validity and consistency of its own data. For example, in an airline database, it is the airline's responsibility to ensure that if a flight is canceled, the database reflects that fact. The cancellation itself should take priority over the different planning tasks that have made reservations on that flight, simply because the airline has the ultimate say on whether or not the flight will actually take place.

1.4.2 Task Longevity

Planning tasks are long-lived, as they are active from the time the plan begins to be specified until the planned activities actually complete. Yet it is the responsibility of the planning application to disturb the operation of the enterprise actually responsible for the database as little as possible. This means that the planning task itself should not block the enterprise from accessing its own data, at least for any major length of time. Thus, the planning applications should not hold locks on any information in the multidatabase for more than a brief period. Certainly they should not hold locks for the entire duration of the planning application.

Because (as a consequence of their long life) planning tasks are not atomic, their effects on the local databases in the multidatabase become visible before the planning task completes. This impacts the concurrency control and recovery protocols in the multidatabase. In particular, in the case in which a planned activity is canceled, the multidatabase must be able to *semantically undo* effects on the local database that are already committed and visible. That is, the multidatabase must be able to run new transactions on the local databases that back out the changes made by the planning tasks. This should leave the local databases in a state that is semantically equivalent, but not necessarily identical, to the state they were in before the planning task ran.

1.4.3 Task Flexibility

A planning task does not know the exact plans that it will make *a priori*. Rather, different acceptable alternatives are examined, and the "best" plan (by some definition) is chosen among the different possible alternatives.

We view making a plan as a searching process. The search for the best available plan begins with an attempt to do the job in the best possible way. As different parts of that plan can be successfully put into place, different local databases are modified and their results committed. If the best way does not work, the planning application will try the second best way, and so on. For example, a traveler may try to book his flight on United Airlines. If that fails, he may try American. Once the flight is booked, he may try a

hotel. If his first choice hotel does not work, another hotel might be tried (without affecting the existing plane reservation).

Defining a planning task flexibly to the database allows a certain amount of this searching to be done automatically by the planning application. Different alternatives are specified and prioritized. The priorities can then be used to guide a more automated search. However, when a particular plan is partially realized and then some critical step fails, it may be necessary to backtrack over some set of the completed (and committed) subtasks. This backtracking may involve removing the effects that that partial plan had committed in the multidatabase. Because of this, flexibility also implies a need for semantically undoing committed work during the search for the best feasible plan.

1.4.4 Task Reactivity and Evolution

A planning task really operates on promises. A flight reservation is a promise that the traveler can actually fly to his destination on a specific flight at a specific time. However, because of the autonomy of the local databases, the multidatabase itself really can make no guarantees that the promise will be kept. We say that the multidatabase can *evolve* into a state that is inconsistent with the completed work of the task.

There are really two problems that occur when the multidatabase evolves like this. The first is that the task itself is no longer consistent with the state of the multidatabase. That is, the task would no longer get the same results if executed after the multidatabase evolved. For example, if a flight were canceled, and some task had made a reservation on that flight, the task is now inconsistent with the database. If the task were executed from the current state, the results would be different.

Because of this, we need to determine some way of changing the task (including its committed effects on the multidatabase) such that it is consistent with the current state of the multidatabase. This can be done by monitoring key data items to ensure that they remain consistent with the expectations of the task, and *reacting* when the conditions are violated by backtracking over inconsistent subtasks and redoing that work to reach a new consistent state. Since we already have a flexible task specification, finding a new way to accomplish the same task can be done without explicitly invoking the application problem. Backtracking over completed but inconsistent or invalid subtasks, however, requires the same semantic undo capability that was used during the original planning search.

Booking a new flight to replace a flight that was canceled is an example of a case in which a task must react to a change in the multidatabase. However, once this type of reaction begins, the task becomes internally inconsistent. Thus, regaining consistency when a multidatabase evolves in a way that impacts some task also requires that the task regain its internal consistency.

The real problem here is determining exactly what needs to be done to regain internal consistency. Because we have existing promises to the traveler in the form of reservations, we want to disturb the original plan as little as possible during this process.

For example, if a hotel was booked based on the arrival date of the flight, the task could just assume that the hotel booking is now inconsistent. This may not be true however, as the new flight may be on the same day as the old flight was. In this case, even though the hotel reservation was dependent on the plane reservation, the critical results from the plane reservation (from the point of view of the hotel) did not change. The rebooking of the hotel is simply not necessary. In complex plans in which the user wants to minimize the impact of the plan change on the task as a whole, it becomes very important to deal with this type of situation efficiently. In fact, we consider this so important that we place an additional requirement on reactivity: it must not only bring the task back into a state which is both consistent with the multidatabase and internally consistent, but also it must do this by making as few a number of changes to the plan as is possible.

We assume that the different local databases in the multidatabase are autonomous. Therefore, we also insist that the monitoring of the local databases for changes, the tasks' reactivity to changes, and the backtracking/redoing required to bring the tasks to new consistent states must be done without the explicit cooperation of the different local databases.

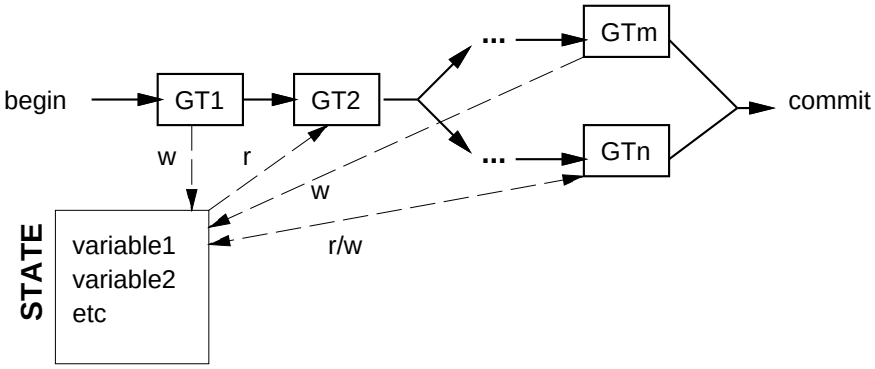


Figure 1.1: Basic structure of an Interaction.

1.5 The Interaction Approach

Interactions are flexible, open nested database transactions. By open nested, we mean that Interactions are defined as a related set of global transactions, each of which executes in its own *sphere of control* (as defined by Davies in [Dav78]). As each global transaction completes it fully commits and releases its resources. By flexible, we mean that alternative executions may be specified, though only one will actually succeed.

The Interaction model is meant to support applications whose tasks execute for a long time and/or may be defined flexibly. A key new feature of Interactions is its support for backtracking, both in searching for and executing the best feasible alternative from the flexible definition, and in backing out when some previously committed subtask has become invalid.

A task is represented in the multidatabase by an Interaction execution. An Interaction execution on a multidatabase is a partially-ordered set of atomic global transactions that share some state. The partial order of the global transactions reflects the execution dependencies among the global transactions (i.e. places where one global transaction must commit before a second can begin) as well as the order in which the global transactions of the Interaction read information in the Interaction variables from each other. The state of an Interaction is kept in its *variables*. We assume that the variables are maintained in a "state database", which is a local database used exclusively for multidatabase management information.

1.5.1 Interaction Structure

The structure of an Interaction is shown in Figure 1.1. In this figure, the boxes labeled $GT1, \cdots, GTn$ represent global transactions on the multidatabase, each of which is assumed to execute atomically. The arrows between the boxes represent the execution dependencies among the global transactions; thus, $GT1$ must commit before $GT2$ begins, but the executions of GTm and GTn can overlap. Each of the different global transactions also accesses the Interaction state in some way; these accesses are represented by labeled dashed arrows indicating whether the information is read from or written to the Interaction state, or both. In this Interaction, if GTm writes some information read later by GTn, then GTm would have to precede GTn in the partial order.

We make certain assumptions about the global transactions on the multidatabase in our work. Each global transaction consists of a set of *global subtransactions*, each of which executes on a single local database. Thus, the global transactions span multiple local databases. However, to help ensure that the global transactions are atomic, no two global subtransactions of the same transaction execute on the same local database. Each global subtransaction begins at the time the global transaction starts executing on the local database, and commits atomically with its global transaction commit. This is necessary to ensure the atomicity of the global transactions, as other transactions on the local database could see partial results if they serialized between two subtransactions of the same global transaction.

Each global subtransaction is executed as a sequence of *steps*. Each step accesses information on a single local database, and derives and logs the information required for replanning and recovery, in addition

to executing its subtask on the local database. The Interaction is defined in terms of these steps. The Interaction definition also specifies how the steps are grouped into global transactions.

1.5.2 Flexibility

Interactions are a *flexible* transaction model. Flexibility means that different parts of the Interaction's task may be accomplished in alternative ways. The Interaction specifies how the different alternatives can be combined to accomplish the same task. Interactions are flexible at the level of global transactions.

We recognize a couple of different types of flexibility. *Alternatives* provide different global transactions that can be run to accomplish the same part of the task. For instance, a plane reservation could be made on one of several different airlines. A second type of flexibility is to allow parts of the task to be optional (*non-vital*). For example, the global transaction to reserve the car might be non-vital, so that if no car is available the whole trip will not be aborted.

1.5.3 Atomicity, Visibility and Conflict

Each global transaction in an Interaction commits atomically once all of its steps have executed (open nesting). In this sense, Interactions are like Sagas [GS87], in that this commitment makes the effects of the global transaction on the database visible, to other global transactions and Interactions as well as to independent transactions on the local databases. As with Sagas, this means that if the Interaction fully or partially aborts, the committed global transactions must be semantically undone using compensating transactions.

Clearly, once a global transaction of some Interaction commits, any effects it has on the multidatabase not only become visible to other Interactions and independent transactions, but also can be overwritten. This overwriting may violate the (still running) Interaction's notion of internal consistency. For instance, in the travel agent example one global transaction may do the work involved in setting up a plane trip. The whole Interaction assumes that the plane reservations persist in the database, at least until the plane flights are over. Thus, a second global transaction of the same Interaction may later make hotel reservations based on the times and dates of the airplane flights, even though the reservation may now be canceled by some other transaction.

In order to deal with these issues, we introduce a notion of *weak conflicts* associated with an Interaction. In our terminology, *strong conflict* denotes the conflict between global transactions that is necessary to preserve their isolation. Strong conflict is enforced by the correct definition of the global transaction boundaries in the Interaction. Weak conflicts are data stability conditions that should hold for some portion of the Interaction execution. When a weak conflict is violated an *event* is generated (this is similar to raising a signal in an operating system and having it caught). This causes the Interaction to semantically undo relevant parts of its work, and then try to re-execute the Interaction from the resulting state, possibly choosing different alternatives from the flexible specification.

Weak conflicts are used to specify when a database should notify the planning application so that it can react to a specific change in the underlying data. This is particularly necessary in a planning task because of its inherent structure. Typically a planning task generates one or more bursts of activity on the multidatabase as the reservations are made and their effects are reflected in the data itself. Once the plans are complete, the multidatabase has effectively promised that certain conditions will hold. If later some change in the data disturbs these plans (such as the airline canceling the flight), the planning task needs to replan around the problem. Thus, even though the plans are complete, the planning task hangs around "just in case something happens". Once the planned activities are complete, the planning task and its associated conflicts can go away.

1.5.4 Recovery and Replanning

Most of the work on open nested transactions and recovery deals with recovery due to process, communication, or storage failure (*failure recovery*). There is work on Sagas [GS87] that applies to Interaction abort. All those methods apply directly to Interactions. However, we expect Interactions to be *reactive*. Thus, when an event occurs, we need to support *user-initiated* requests to undo work that has previously committed.

When the work done by some global transaction in an Interaction becomes invalid, for instance if some weak conflict that it set is violated, then all committed work done either by that global transaction or by some other global transaction in the Interaction that depends on it must be *semantically undone*. This can be done by generating compensating transactions for each such global transaction, and executing them in a way that correctly reverses their collective effects. Once all of the invalid work has been semantically undone, the Interaction can then try to re-execute from the resulting state. Note that, as with other flexible transaction models, the re-execution may diverge radically from the original execution.

One problem with using compensation for replanning is that effects of the invalid global transactions are visible for some time. Other Interactions or independent transactions may read and/or modify the affected information, thus their execution *depends on* the execution of the invalid global transaction. This problem is described in detail in [KLS90]. During replanning it is important to detect these dependencies, as the invalidation may or may not need to cascade to the dependent transactions.

With Interactions, this cascading occurs via the weak conflicts. Weak conflicts indicate to the multidatabase what constraints must hold on specific data items for each running Interaction to remain internally consistent. We have seen that weak conflicts can be violated by another executing transaction; but they can also be violated by some compensating transaction during replanning. Thus, if the semantic undoing of some portion of some Interaction violates the internal consistency of some other Interaction, that Interaction will be notified of the event and enter into its own replanning process.

In this work, we differ radically from earlier work on compensation in that we do not necessarily compensate for the invalid global transactions of an Interaction in the inverse of the original execution order. Rather, we attempt to *replace* invalid work while at the same time leaving intact successive work that is not affected by the change. We call this *replacement*.

In our work, we assume that all global transactions that are executed are compensatable. If some step cannot be compensated for, then the remainder of the Interaction must be run as a single atomic global transaction. However, unlike other work in open nested transactions, we allow weaker compensating transactions to be attempted if an exact semantic undo of some global transaction fails.

1.6 A Layered Architecture

The different layers of function required to implement a multidatabase that supports Interactions are shown in Figure 1.6. The multidatabase application runs in the application layer.

Just below the application layer is a layer that persistently stores the Interaction itself and its execution state. This layer is responsible for examining the flexible Interaction specification and guiding the search to find the most acceptable plan. It notes which alternatives were tried and failed, which alternatives succeeded, and which alternatives were never tried. If a weak conflict is violated, it determines which new alternatives to retry, once the invalid work is identified.

The Interaction execution layer and the global transaction layer schedule and execute concurrent sets of Interactions and global transactions, respectively. The Interaction execution layer is responsible for ensuring that the Interactions are scheduled correctly. The global transaction execution layer is responsible for ensuring that the global transactions are executed atomically.

Each local database has its own functions for dealing with global subtransactions and steps that run on itself. The global subtransaction execution layer is responsible for representing the global transaction on the local database and for taking the place of the local database in any global agreement protocols such as two-phase commit.

Global subtransactions execute queries and updates on the local databases via steps. Steps are defined, and their executables persistently stored, in a library associated with the step definition and execution layer. Each step definition contains commands in the local database's DML to run the functions on the database, and also commands to extract and store information required if the step ever needs to be compensated for.

The bottom layer is the local database. It is assumed to be autonomous from the layers above it.

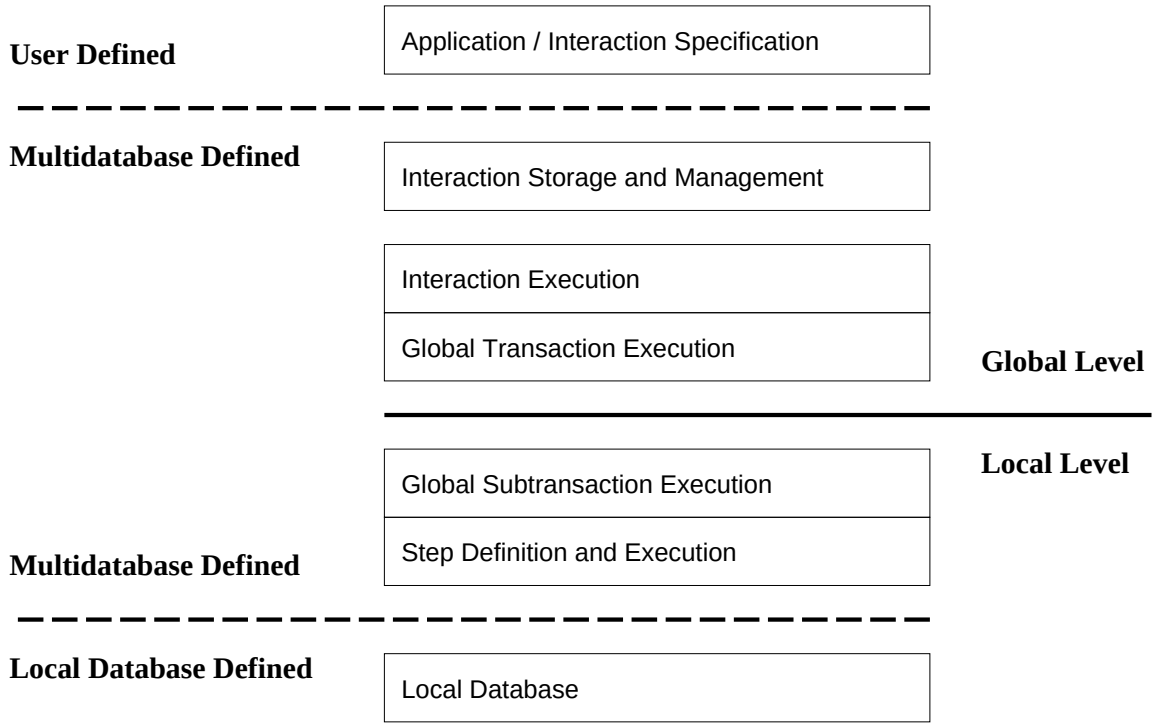


Figure 1.2: Layers in the multidatabase architecture

1.7 Contributions

This thesis contains two categories of contributions. The first category is theoretical, including formal definitions for Interaction correctness. The second category includes contributions from the design and implementation of our prototype multidatabase, Mongrel.

The Interaction transaction model differs from other transaction models in the following ways:

1. Reactivity - the ability to perceive changes in the multidatabase that conflict with the Interaction's work, and to modify the Interaction's work to be consistent with the change.
2. Flexibility implemented over open nesting - different executions of Interaction could cause different sets of global transactions to be executed.

Theoretical contributions that relate to our work on Interactions include the following:

1. Definition of correctness for a reactive open nested transaction execution.
2. Scheduling and reactivity algorithms to ensure correct execution.

In addition to the above work Interaction correctness also requires the multidatabase to enforce global transaction serializability. Theoretical contributions in this area include:

1. A characterization of protocols to maintain a consistent global serialization order across all local databases.
2. A complete theory of global transaction commitability.
3. Two new global transaction commit algorithms.

The second set of contributions of this thesis comes from my design and implementation of a prototype multidatabase, Mongrel, that uses Interactions as its basic transaction model. Contributions related to Mongrel include the following:

1. A procedural approach for accessing information in the local database. The procedures are called *steps*.

2. A defined association between steps and compensating steps that allows automatic generation of compensating (semantic undo) transactions when needed.

The Mongrel multidatabase has a data manipulation language, TaSL, that is used to specify Interactions. TaSL borrows features from block-structured languages, other data manipulation languages, and planning languages. TaSL also assumes that it is running on a multidatabase that is accessed using steps, like Mongrel. The combination of features unique to TaSL includes:

1. Steps as the basic building block, with control structures defining how the steps are executed.

2. Database primitives for combining steps into global transactions.

3. Flexibility specified in terms of the global transactions, with prioritized alternatives.

4. Primitives to specify potential weak conflicts.

5. Reactivity primitives to guide backtracking when a task's work is invalidated.

1.8 Road Map

In this thesis, we define the basic Interaction model, using the travel agent application as the basic example. We define a language for specifying Interactions in Chapter 3. In Chapter 4, we specify correctness criteria for Interactions in which no weak conflicts are violated. We also define basic synchronization algorithms for enforcing correctness. In Chapter 6 we introduce the basic tools that are used to make Interactions reactive. This includes expanding the correctness definition to include the correctness of Interactions that have had weak conflicts violated. We also define the events that can indicate weak conflict violation to the Interaction, and give replanning algorithms that conform to the specified notion of correctness. In Chapter 5 we discuss new techniques for enforcing global transaction atomicity, which is a prerequisite for correct Interaction synchronization. In Chapter 7, we describe the basic design of the system that we built to manage Interactions.

Chapter 2

Related Research

In 1986, Gligor and Popescu-Zeletin [GP86] wrote a paper on concurrency control issues in multidatabase systems. They outlined a set of requirements on local databases in a multidatabase so that atomic transactions on the multidatabase could be supported. Since then, many interesting developments have occurred in the area of transaction models for multidatabases. Many of these are summarized in [BGS92].

In this chapter, we discuss other research that is related to our work on Interactions. We divide the work on histories and correctness into three sections: one to discuss serializability in multidatabase transactions, one to discuss issues of relaxing serializability slightly so that it is easier to enforce in a multidatabase, and one to discuss multilevel and open nested models for multidatabase transactions. Also, since weak conflicts are related to active databases, we give a brief summary of the active database research. We also summarize work on compensation and recovery. Finally, we summarize some relevant work done on Artificial Intelligence planners.

2.1 Serializable Multidatabase Transactions

The serializability of global transactions in the multidatabase is a prerequisite for correct Interaction execution. Our definition for Interaction correctness relies not only on the atomicity of the different Interactions' global transactions, but also on being able to enforce consistency between the order in which each Interaction defines its global transactions and the order in which those global transactions are serialized in the multidatabase.

Given that each global transaction only has at most one global subtransaction per local database in a multidatabase, two conditions must be maintained to ensure global transaction serializability. One is that the local databases all serialize their global subtransactions in an order consistent with some general global transaction serialization order. The second is that an atomic commit protocol such as a two-phase commit is used when committing each global transaction.

Early efforts to provide a global transaction model for multidatabases generally limited the global (and possibly local) transactions' access to the information in the database. For instance, Multibase [SBD+81] only allowed the global transactions read access to the information in the multidatabase.

In [BS88], Breitbart and Silberschatz examined issues in allowing multidatabase updates. They proposed an algorithm based on *site graphs* that ensures that no two concurrent global transactions access more than one database in common. However, this method allows a very low degree of concurrency among the global transactions. In [EH88], Elmagarmid and Helal restricted the global transactions so that the set of databases read from and the set of databases written to by a single global transaction had an intersection of at most one local database. Later schemes [BST90] partitioned the information into locally- and globally-updateable information, and restricted access accordingly. This idea also plays a part in [MRKS91] and [MRB+92a].

More recent approaches rely on two protocols; one to ensure that the global transactions' subtransactions serialize consistently on the local databases, and the other to simulate a two-phase commit. This approach is also the one we take. We survey this work because any of these approaches can be used when building a system that uses Interactions as its transaction model. The work surveyed here is all recent, and for the most part was done in parallel with the global transaction serializability work presented in this thesis.

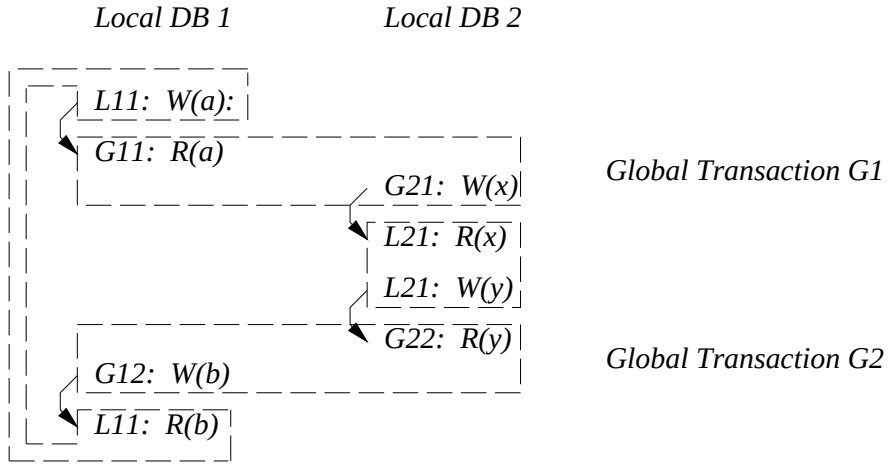


Figure 2.1: Cyclic information flow problem. Global transaction G1 has subtransactions G11 and G21. Global transaction G2 has subtransactions G22 and G12. Time increases as you read down.

2.1.1 Enforcing a Global Serialization Order

In a multidatabase, any two global transactions are serializable only if they do not read information from each other. However, because the local databases are autonomous, and can run transactions independently of the multidatabase, determining whether or not a global transaction reads from another can become very difficult. Consider the example in Figure 2.1. In this figure, G11 and G12 do not conflict at all if they are executed with no interfering local transactions. However, the concurrent execution of local transactions L11 and L21 means the multidatabase history is not serializable. In fact, it can be shown by induction that, given a sufficient number of data items in the database, two global transactions whose executions are separated by an arbitrary amount of time can still have their serialization orders opposite to their execution orders.

The trick with enforcing a global serialization order is to ensure that the local databases all serialize global subtransactions in an order consistent with some global transaction serialization order. In other words, each local serialization order must be consistent with the global serialization order. The global transaction manager must either prevent inconsistencies from occurring, or abort global transactions when inconsistencies have occurred. There are a variety of schemes that have been proposed for this, each of which has a rough analogue in the realm of concurrency control in a single local database. These schemes are described below, and are summarized in Table 2.1.

2.1.1.1 Pessimistic Schemes

Breitbart *et al.* [BGRS91] define a desirable property for local databases called *analogous execution and serialization order*, which basically means that the serialization order of a database can be derived from the order in which the transactions execute. This is not always true, though it is true for the local databases that enforce *strong recoverability*. A strongly recoverable schedule is one in which for any pair of transactions T_i and T_j with conflicting operations $op_i(x)$ and $op_j(x)$, the commit of T_i precedes the commit of T_j. In [Raz92a, Raz92b], Raz refers to this property as *commitment ordering*.

In a multidatabase, if each local database guarantees strong recoverability (or, equivalently, commitment ordering), then the global transaction manager can derive and/or control the serialization order based on the order in which the global transactions execute their subtransactions on the local databases. If the global transaction commit is atomic, then the resulting multidatabase schedule will also be strongly recoverable.

A commonly-used property of local databases that generates a subset of the strongly recoverable schedules is *rigorousness* (also called strong-strict in [Raz92a]). A rigorous execution has the property that no local data item can be read or written until the previous transaction that read or wrote that data item has committed. Rigorousness can be guaranteed within a two-phase lock framework by not allowing a transaction to release any lock (read or write) until it commits.

If all of the local databases in a multidatabase use rigorous concurrency control schemes, then enforcing the atomicity of global transactions can be done simply by ensuring that they commit atomically. The global serialization order is then the global transaction commit order. The atomicity of the commit process ensures that the order is consistent across the different local databases. This observation was also made by Soparkar *et al.* [SKS91a]. However, if we relax the requirement on the local databases to *strictness* (which allows read locks to be released before transaction commit), then they no longer have analogous execution and serialization order. An example non-serializable multidatabase history where the local histories are strict is given in [BGRS91].

Pessimistic schemes require the local databases in the multidatabase to all be strongly recoverable. This is a subset of the types of local databases we allow when using the serialization order approaches in our prototype implementation, which are merely required to be serializable. However, the pessimistic schemes are easier to implement than our schemes.

2.1.1.2 Conservative Schemes

Conservative schemes require that all local transactions and global subtransactions in a multidatabase declare what information in the database they will access at the time they begin. Using this information, both the ROLL scheme used in Hydro [PRR91] and Mehrotra *et al.*'s O-schemes [MRB+92b] generate a global serialization order that allows for maximal concurrency in the multidatabase, i.e. one that accepts all serializable multidatabase histories.

Hydro [PRR91] uses request-ordered linked lists (ROLLs) to determine when a global transaction can execute. Each entry in the ROLL consists of a bit vector, with each bit representing some resource in the multidatabase. Each bit vector represents a resource request for a specific local or global transaction, and is placed at the end of the ROLL at the time the transaction begins. Periodically the transaction's bit vector is compared to the ones that precede it in the ROLL, and the transaction is allowed to proceed once no such bit vector indicates a request for any resource that it wants. Once the transaction runs to completion, its entry can be removed from the ROLL.

Mehrotra *et al.* [MRB+92b] provide a framework for defining conservative global concurrency control schemes for multidatabases. In this context, they also define several different schemes that generate global serialization orders consistent with the global transaction begin order. They call these *BT-schemes*. They also define an instance of an *O-scheme* that allows all global serializable multidatabase schedules. Their O-scheme works by processing not only the begin operations of each transaction, but also their serialization point operations. Because of this, they can minimize the restrictions added to the schedule at each point, and allow for more concurrency among the global transactions.

While conservative schemes seem to be able to provide a lot of concurrency, and have the additional advantage that they do not ever abort global transactions, they do have the disadvantage that each global and local transaction must be able to determine and report its resource requirements at the time it begins. This may not be feasible in environments where the global transactions are open-ended. This situation may occur in our environment, if we allow plans to be specified interactively. Also, if the local databases do not normally require the predeclaration of resource requirements, this may place an undue burden on the programmers of the transactions that run only on the local databases. We believe that the conservative schemes are impractical for the class of planning applications that we examine in this thesis. Therefore, we have not really considered the conservative schemes.

2.1.1.3 Conflict-Based Schemes

All of the multidatabase work to date assumes that the local databases enforce conflict serializability, Because of this assumption, we know that the serialization order generated by a local database is determined mainly by the order in which conflicts are resolved. In the conflict- based schemes, the multidatabase uses this knowledge to push all global conflicts down into the local databases, and to allow the local databases to resolve those conflicts as needed. If the local databases do their job, the global serialization order is maintained.

The simplest conflict-based scheme is *forced local conflict* [GRS91]. With forced local conflict, each local database maintains a separate data item, called the "ticket", for use by the multidatabase. The multidatabase ensures that the tickets in the local database for the different global transactions all be written in the global

serialization order. If some local database has resolved another conflict between two local databases in some different order, this shows up in the local database as a non-serializable execution, and is detected or prevented by the local concurrency control manager. For example, if the local database uses two-phase locking, either the taking of the ticket or the execution of the conflicting write operation results in a deadlock, which can be detected locally by the local deadlock detection scheme.

[GRS93] describes various schemes for using forced local conflict. One is based on an optimistic approach of taking tickets when needed and validating at global transaction commit time. They do assume that the local databases support a visible prepared state for two-phase commit, as well as producing a locally serializable schedule. Their conservative method controls the order in which the global transactions take a ticket. This paper also defines the best times to take tickets based on different concurrency control schemes, and when tickets do not need to be taken (they are *implicit*). Much of the work in this paper is similar to our (parallel) results on global transaction serialization points from [Nod91] described later in this thesis.

Forced local conflict is really an elegant solution in and of itself to the multidatabase concurrency control problem. Not only does it use the local database concurrency control mechanisms to resolve any global serialization order problems, but it also allows the multidatabase to enforce different global serialization orders by *when* during the global transaction execution its subtransactions get their tickets. For example, if the multidatabase orders the global transactions in their commit order, then the tickets are taken during the global transaction commit process, just before the subtransactions commit in the local database.

Forced local conflict can also be used to fill in weaknesses in some of the other schemes. For example, Mehrotra *et al.* [MRB+92b] use forced local conflict in their schemes with local databases where the serialization point is not externally detectable (e.g. in local databases that use serialization graph testing as their concurrency control mechanism). The forced local conflict restricts the histories accepted by the local database to those that serialize the global subtransactions correctly. We also take this approach in our work on enforcing global serialization order.

A different set of conflict-based schemes can be developed based on the correctness criteria described in [ZE93a, ZE93b]. Unlike forced local conflict, the conflict-serializable, sharing-serializable, and hybrid-serializable schedules generated by these schemes do not require the modification of the local database schemas to add a ticket. Thus, with respect to maintaining a global serialization order, these schemes allow the local databases to remain fully autonomous.

In this work, two transactions "share" if one transaction accesses a superset of the data items the second transaction accesses. A set of transactions that share can be forced to serialize in order of increasing (or decreasing) number of variables they access. Two transactions "conflict" if they access the same data item and one of them is a write operation. Conflicting transactions serialize in the execution order of their conflicting operations. Hybrid serializability combines the two notions, and defines the most general scheme to generate a global serialization order that also preserves the local database autonomy. It does, however, require that the read- and write- sets for each global transaction be deduced. Also, a scheduler to generate hybrid serializable global transactions may generate extra read and/or write operations on the local databases in certain situations where natural sharing or conflicting relationships do not exist between two global transactions.

Zhang also presents a strictly conflict-based scheme where each global subtransaction reads one data item that the previous global subtransaction in the global serialization order wrote. This scheme does not require modifying the local database schemas to insert a ticket for the multidatabase. It also eliminates the problem of high contention for the ticket.

Conflict-based schemes are the most generally useful of the schemes we have discussed so far. In our work, we use conflicts to enforce global serialization order on certain problematic types of local databases, such as those that use serialization graph testing as their basic concurrency control mechanism.

2.1.1.4 Optimistic Schemes with Certifiers

In the optimistic schemes, the multidatabase places no control over the order in which the local databases are allowed to execute their global subtransactions. At the time a global transaction commits, it requests from the local database a specification of the order in which the global subtransactions were serialized. The commit process checks these orders for consistency, and aborts any global transaction in which the global subtransactions were ordered inconsistently.

In his Superdatabases work [Pu88], Pu takes an optimistic approach to providing serializable global transactions. He requires each local database to provide the serialization order of its transactions to the global transaction manager. The global transaction manager uses these orders *(O-vectors)* to compose the local transactions into global transactions, and at the same time determine if any of the global transactions do not serialize. This is noted by an inconsistency in the orders of the global subtransactions in the different O-vectors. Since Pu commits a global transaction only if its subtransactions follow those of older global transactions, his scheme certifies the global transactions in an order consistent with their begin order.

It is also possible to have a certification scheme that certifies global transactions in an order consistent with their commit order. In this scheme, again the O-vectors are checked at transaction commit time. The global transaction commit succeeds only if its global subtransactions do not precede any committed global subtransaction in any O-vector.

Certification schemes require the ability to generate the O-vectors for every local database in the multidatabase. The O-vectors can be deduced for many different local database concurrency control methods, including the two-phase-locking (2PL) databases and the timestamp ordering (TO) databases. However, some local database concurrency control schemes, notably those that use serialization graph testing (SGT), provide no convenient way of deducing the O-vectors. In this case, the local databases must be modified to make that information available to the multidatabase. An alternative in this case would be to use forced local conflicts in this type of local database, and make the O-vector be ordered in the order the conflicting operations are issued.

In [GRS93], Georgakopoulos *et al.* provide a scheme that does not deduce the serialization order at the local database, but rather forces conflicts and reports the serialization order as the order of the forced conflict. This they call the *Optimistic Ticket Method*. When a global transaction is ready to enter the prepared state, the global transaction manager compares the different local serialization orders to ensure that it is serialized correctly. If not, the global transaction is aborted.

[SKS92] provides a distributed optimistic scheme to ensure that a consistent global serialization order based on a potentially distributed global transaction manager. In this scheme, the piece of the global transaction manager that is responsible for coordinating a specific global transaction synchronizes the local databases via a basic agreement protocol. When the global transaction reaches its synchronization point, its global transaction manager enters into an agreement protocol. It coordinates with the other global transaction managers that ran parts of the global transactions. The protocol concludes once all of those global transactions agree that the global transaction was serialized in the correct order. Global serializability is proven based on the fact that no two global transactions that conflict locally can be in the synchronization process at the same time, and also based on the (Lamport-style) happens-before ordering of the synchronization messages.

[SKS92] also characterizes the set of local databases that can be effectively synchronized as those that have a natural synchronization interval (serialization point) that occurs during the active execution of the global transaction. Here, the natural synchronization interval would encompass the operations that determine the synchronization order in the local databases, e.g. getting the timestamp in a timestamp-ordered local database.

Certification schemes for enforcing a consistent global serialization order have similar advantages and disadvantages to certification schemes used in the local databases themselves. The major disadvantage is that global transactions will abort more of the multidatabase uses a certification scheme. The extra aborts occur because the multidatabase uses global transaction aborts as the basic tool for eliminating inconsistencies. We believe that in a multidatabase environment, it is better to delay a global transaction than to abort it. Therefore, we have not thus far used certification schemes in our prototype multidatabase.

2.1.1.5 Summary

The strategies for enforcing a consistent serialization order for global transactions in a multidatabase parallel the different concurrency control strategies that are used in regular databases. This is not surprising, given that the concurrency control in a regular database tries to order conflicting operations on data items such that the transactions access each such data item in a consistent order. This ultimately makes the transactions serializable. A multidatabase similarly tries to order conflicting global subtransactions on local databases such that the global transactions access the local databases in a consistent order. This helps ensure that the

Concurrency Control Mechanism	Serialization Point	Corresponding Multidatabase Mechanism	Restrictions
Serialization Graph Testing	Data Conflict	Site Graph Testing [BS88]	
Two-Phase Lock	Last Lock to Commit	Forced Local Conflict [GRS91] (Get ticket after last lock.)	I
Timestamp Order	Begin	Forced Local Conflict [GRS91] (Get ticket with transaction begin.)	I
Rigorous 2PL	Commit	Strict 2PL/2PC Schemes [SKS91b] (Rigorous 2PL local databases only)	R
		Rigorous multidatabases [BGRS91]	R
Commitment Order	Commit	Multiple Resource CO [Raz92b] (CO local databases only)	R
Conservative SGT	Data Conflict	Request-Order Linked List [PRR91]	C
		O-schemes [MRB+92b]	C
Conservative 2PL	Commit		
Conservative TO	Begin	BT-schemes [MRB+92b]	C
SGT Certifier	Data Conflict	Optimistic Ticket Method [GRS93]	I
2PL Certifier	Commit		
TO Certifier	Begin	Superdatabases [Pu88] (Committed GT follows older GTs.)	A

Table 2.1: Comparison of single database concurrency control schemes to multidatabase concurrency control schemes. In the restrictions, I means the scheme requires additional information in the local databases. A means the scheme requires that control or execution autonomy be violated in the local databases. R restricts the local database concurrency control mechanism further than just supporting ACID transactions. C implies that all global transactions must predeclare their resource usage.

global transactions are serializable in the multidatabase.

A multidatabase can serialize global transactions in the order they begin, in the order they end, or in the order their conflicting operations execute. Similarly, local database concurrency control schemes can serialize the local database transactions in begin, commit, or conflicting operation execution order. Because of the autonomy of local databases, however, it is necessary to avoid schemes analogous to serialization graph testing, where the serialization point can occur outside of the lifetime of the transaction.

Table 2.1 provides a summary of the different multidatabase concurrency control strategies and their serialization points. The first column lists different single database concurrency control schemes. Their respective serialization points are listed in the second column. In the third column we list analogous multidatabase concurrency control schemes. For instance, site graph testing [BS88] uses a graph approach similar to a serialization graph, and delays submission if a cycle is found in the site graph. Column four summarizes the restrictions imposed by the multidatabase concurrency control scheme on the local databases and the transactions they run.

2.1.2 Multidatabase Atomic Commit Algorithms

Supporting atomic commitment in multidatabases is difficult because of the necessity to maintain local database autonomy. This means, for instance, that the multidatabase cannot ensure that the local databases support a prepared state for a two-phase commit, or any of the functions required to implement other agreement protocols for commit. In the general case, this means that atomic commitment of certain types of global transactions is not possible [MKS92]. In the global transaction commit work in this thesis, we give a simple characterization of when global transactions can and cannot atomically commit. We also give a general framework for atomic and semantically atomic commitment. All of the algorithms described in this section fit into our general commit framework.

Most of the work on supporting atomic commitment in multidatabases is done by providing a simulated

Scheme	Local Commit Before Global Decision (Requires Undo)	Global Decision Before Local Commit (Requires Redo)	Paradigm
2PC Agent [WV90]		√	LDB blocking rigorous LDBs full atomicity
Hydro [PRR91]	√		LDB Blocking full atomicity
Optimistic Commit Protocol [LKS91a]	√		stratification semantic atomicity
Mehrotra *et al.* [MRB+92a]	read-only subtransactions	update subtransactions	data partitioning rigorous MDB full atomicity
Non-blocking commit [MRKS92]	compensatable subtransactions	retriable subtransactions	allows a pivot semantic atomicity

Table 2.2: Summary of multidatabase global commit strategies.

prepared state for a two-phase commit. Muth and Rakow [MR91] analyzed in detail the requirements for providing a simulated prepared state. They divided up the schemes for doing this into two categories: (1) local commitment after global decision and (2) local commitment before global decision. In both of these schemes, Muth and Rakow assume that each local database has some type of agent to act on its behalf in the two-phase commit. It is this agent that participates in the voting process. The local commitment after global decision schemes do not commit the global subtransactions in the local database until after the multidatabase has taken a vote and made a decision to commit or abort the global transaction. Here, the danger is that some global subtransaction may abort, so there is a requirement to be able to *redo* any such global subtransaction *without causing a violation of the global serialization order*. The local commitment before global decision schemes commit the subtransactions in the local database as the local databases vote in the two-phase commit (2PC) protocol. With this scheme, the danger is that the decision may be to abort, in which case the global subtransaction needs to be *semantically undone*. In this situation, the local database must serialize the transaction that does the semantic undo immediately after the global subtransaction to guarantee full atomicity. This is because any transaction that must serialize between the two sees the effects of the aborted global transaction.

Several different commit strategies for global transactions have been proposed. They are summarized in Table 2.2.

Systems which use the global decision before local commit strategy include the 2PC Agent method [WV90], Hydro [PRR91], and the optimistic commit protocol [LKS91a]. In the 2PC Agent Method [WV90] of Wolski and Viejalainen, a 2PC Agent associated with each local database acts for the local database as a participant in the two-phase commit protocol. The 2PC Agent keeps a log so that it can redo any global subtransaction that was in the prepared state at the time of a failure or database-induced abort. This method assumes that local transactions do not update objects read or written by global subtransactions (they call this restriction DLU for "Denied Local Updates"). They also assume the local databases use strict 2PL, so their scheduler produces strict global schedules. [GRS93] gives a counterexample to show that local strictness may not be sufficient to ensure global serializability, and that the local databases in fact must produce rigorous schedules. This counterexample is based on the fact that strictness allows read-locks to be released early. However, if the local databases are further restricted to be *rigorous*, the 2PC Agent Method is correct.

Hydro [PRR91] enforces the property that the semantic undo must serialize immediately after the original transaction by preventing any transaction on the local database from committing during the time between when the global subtransaction commits and the time the decision for the global transaction is reached. If the global transaction aborts, then one or more of the transactions that has been active during the decision phase may need to abort in order for the semantic undo to be correct. This effectively does the same thing as the denied local updates in [WV90] in that it gets rid of conflicts with the original (and compensating)

global transaction. However, it does this by aborting local transactions rather than denying their updates. In other words, this scheme violates control autonomy rather than execution autonomy.

Levy *et al.* [LKS91a] also use a local commit before global decision strategy in their optimistic commit protocol. This protocol enforces a weaker notion of atomicity, *semantic atomicity*, where a transaction may see the effects of an aborted global transaction before it is undone. It also assumes *persistence of compensation* [KLS90], which states that a compensating transaction is never allowed to abort.

Optimistic two-phase commit allows global subtransactions to release their locks (i.e., commit) once they have received the "prepare" message and decided to reply with a "yes". The "prepare" message is never sent until all global subtransactions have completed their processing. However, since compensating subtransactions are initiated automatically and run independently, they need not coordinate their commit with a two-phase commit protocol. Rather, they commit as soon as their work is complete. Because the resulting compensating transactions may not necessarily be two-phase locked, the global scheduler is augmented to ensure that the global subtransactions and compensating subtransactions all conform to a *stratification property*. That is, the global scheduler guarantees that no global transaction accesses both a site where some other global transaction T_a is locally committed and a different site where T_a is locally undone. Optimistic two-phase commit was a restriction based on a generalized theory of atomic commitment defined in [LKS91b].

Mehrotra *et al.* [MRB+92a] propose an interesting variant on the above global decision before/after local commit protocols. They observe that the local commit point can vary with respect to the global commit point even within a single invocation of a global transaction commit. If a global subtransaction is read-only, it can be committed/aborted at the time the local database votes in the two-phase commit. Otherwise, it is not committed until the global decision is made. For a read-only global subtransaction, the semantic undo is a no-op, so it is very easy to satisfy its requirements for the local commit before global commit. For the global subtransactions that have write operations, the redo consists only of the write operations of the original global subtransaction. To ensure that a consistent global serialization order is maintained, each local database schedule is restricted in which types of transactions can access which data, and the order in which those operations can be scheduled. Specifically, if a global subtransaction accesses any data item that can be read or written by any local transaction, then it can only write to data items that can be neither read nor written by other local transactions. Also, the local schedule must be strongly recoverable (or, almost equivalently, commitment-ordered) and the projection of the global subtransactions from the local database schedule must commit write operations only after any other, earlier read operations by other transactions, and can only schedule read operations after the operations that previously wrote those variables have all committed.

Mehrotra *et al.* also propose a weaker commit protocol in [MRKS92]. This protocol guarantees only semantic atomicity. It assumes that subtransactions of a global transaction are independent. The subtransactions are divided into three categories: those that can *always* be compensated for *(compensatable)*, those that will *always* eventually succeed if retried enough times *(retriable)*, and at most one that falls in neither category *(the pivot)*. Commitment proceeds by first committing all of the compensatable transactions. If one aborts, these all can then be compensated for. The final, irrevocable decision is really made by the pivot. It is committed next. If the pivot commits, then all of the retriable transactions commit as well. If a retriable commit fails, it is retried until it eventually succeeds. This commit protocol has the advantage that it requires few restrictions on the local database (it has to acknowledge when the commit or abort completes) and it is non-blocking. Some of these ideas form a basis for our work on global transaction commit in Section 5.2.4.

Finally, a general taxonomy and unified approach to atomic commit done by Elmagarmid and Mullen is found in [EM93]. This work has some similarity to our work on atomic and semantically atomic global transaction commits. It draws similar conclusions. However, this work uses a different method for classifying the different subtransactions in a global transaction, based on the subtransaction's semantics. Our classifications of subtransactions are based on a static syntactic analysis. Also, our classifications are much simpler (4 classes in our work as opposed to 9 in Mullen's work). We also find that our graph-theoretic approach is simpler and cleaner than his work.

2.2 Nearly Serializable Multidatabase Transactions

Two approaches have been proposed that relax serializability a bit in ways that are easier to enforce in a multidatabase. Both models assume that the local database executions are serializable. Quasi-serializability [DE89, DEK91], also called multidatabase serializability [Bar90], states that the global transactions must be run in such a way that no two transactions executing on the same local database run concurrently. This means that the projection of the global transactions from the multidatabase's history is serializable assuming that any two transactions running on the same local database conflict at all times. Quasi-serializability preserves database consistency if the executions of each global transaction on the different sites are completely independent, and if replication is the only integrity constraint allowed [MRKS91].

Two-level serializability [MRKS91] differs from quasi-serializability in that its conflict model at the global level is that two transactions conflict on specific data items. Thus, two-level serializability is more general than quasi-serializability. In a two-level serializable multidatabase, data consistency is enforced using different combinations of three paradigms:

1. Partitioning the multidatabase into local and global data items, and controlling which types of transactions can access items in which partition, and how.
2. Constraining the types of integrity constraints that are enforced between the different data items.
3. Not allowing value dependencies between the executions of different parts of a global transaction on different local databases.
4. Controlling the structure of the global transactions, for instance disallowing loops and conditional statements.

The Interaction model described in this thesis also is a non-serializable model, but it uses a different tactic to relax serializability from that used in quasi-serializability and two-level serializability. Interactions use global transaction serializability as a fundamental building block. Quasi-serializability and two-level serializability make the assumption that global transaction serializability cannot easily be enforced, and therefore weaken the correctness criterion to something that is easy to enforce in a multidatabase.

2.3 Advanced Transaction Models

Advanced transaction models are being developed that violate one or more of the ACID properties[1], usually by making their effects visible before they commit. Usually this relaxation is coupled with other techniques that allow transactions to maintain some relaxed notion of correctness. This section describes some different advanced transaction models that relate to the work presented in this thesis.

2.3.1 Sagas

Garcia-Molina *et al.* propose Sagas [GS87] as a long-lived transaction model. Sagas are long (non-atomic) transactions that are broken up into a sequence of shorter, atomic transactions. Sagas are basically a two-level restriction of multilevel transactions (see section 2.3.5). The lower (transaction) level defines atomicity as the correctness criterion for transactions. The conflict relationship enforced there is serializability. At the upper (Saga) level the conflict relation is empty, meaning that concurrent Sagas can interleave their transactions arbitrarily.

Nested Sagas are defined as well, [GGK+90, GGK+91] but since multidatabases are inherently two-level, this work does not necessarily apply to Interactions. Nested sagas also extend the transaction model to provide for more flexible transaction definitions, including adding provisions for alternative ways of accomplishing some operations, and allowing non-vital operations.

Sagas relate to this work in that we use the same approach of dividing up a long task into a related set of shorter, atomic subtasks (open nesting). Interactions, however, assume that more complex relationships can exist between the atomic subtasks. Also, Sagas have no weaker notion of conflict as Interactions have. Nonetheless, we consider Sagas to be an important predecessor of our work.

[1] Atomicity, Consistency, Isolation, Durability

2.3.2 ACTA

ACTA [CR90, CR91b] is not a transaction model, but rather it is a framework for characterizing the structure and behavior of existing advanced transaction models, and for allowing a relative comparison between them with respect to concurrency, concurrency control, and recovery. The ACTA framework characterizes different transaction models by defining how subtransactions depend on each other, and how subtransactions are allowed to affect objects in a database.

In ACTA, how subtransactions can affect each other given a specific advanced transaction model is characterized by a notion of *dependencies*. For example, a *commit dependency* between two subtransactions T_a and T_b means that the transaction model requires T_a to commit before T_b. A commit dependency would occur in a Commit Ordered database [Raz92a] if T_a conflicts with T_b, and T_a's conflicting operation preceded that of T_b. Similarly, an *abort dependency* between two subtransactions T_c and T_d means that in this transaction model, if T_c aborts then T_d must abort as well. ACTA defines several other different types of dependencies that can develop between different subtransactions in different transaction models.

ACTA also can characterize how subtransactions are allowed to affect objects in a database, and more specifically when a subtransaction's effects become visible and/or overwriteable to other subtransactions in that database.

ACTA has been used to characterize many of the different transaction models. For example, Sagas are characterized in the ACTA framework in [CR91a]. While we have not attempted to use ACTA to characterize Interactions, we suspect that ACTA is not yet comprehensive enough for this characterization to succeed. For example, the dependency system defined in ACTA cannot be used to represent weak conflicts and reactivity easily.

2.3.3 Flex Transactions

Flex transactions [ELLR90, Leu91, LEB92] were developed with the premise that often in advanced applications, parts of a given transaction could be accomplished in one or more functionally equivalent ways. Thus, if we allow *functional replication*, or the ability to specify more than one way to accomplish some part of a transaction's task, then we can have a more failure-resilient transaction model. In our example, getting a United flight or a TWA flight for some trip on some day would be two functionally equivalent ways of making a plane reservation. The model is resilient to failures in that, if a United reservation cannot be made, then a TWA reservation will suffice in its place. A rigid transaction model would have to abort the overall transaction at that point.

A Flex transaction is defined as a set of subtransactions, some of which may be functionally equivalent, and a set of ACTA-like dependencies among those subtransactions. The dependencies occur among both the normal subtransactions and the compensating subtransactions. These dependencies define which subtransactions are actually executed in a Flex transaction, and in what order. Note also that, since a subtransaction may fail to accomplish its goal and a functionally equivalent subtransaction may be tried as a consequence, the execution of a Flex transaction also depends on the state of the underlying databases.

The Flex transaction work includes some interesting observations as well. For example, having a flexible transaction model implies the need to have some form of compensation to be used when two functionally equivalent steps executed concurrently both succeed. If steps cannot be compensated for, they vary how they execute the overall transaction.

Flex transactions differ from Interactions in that they support flexibility within the context of a single global transaction. This global transaction is viewed as being atomic, although their work also supports the notion of using a semantically atomic commit protocol.

Flex transactions were implemented within the InterBase system [BCEP93]. They also have been implemented and used to support multi-system telecommunication applications [ANRS92].

2.3.4 ConTracts

In [WR91], Waechter and Reuter describe a database transaction model called the *ConTract Model*. ConTracts are very similar in structure to Interactions. ConTracts are scripts of steps. The steps are manually mapped into transactions on a database (as in our atomic blocks).

ConTracts use exit invariants, which are similar to weak conflicts. Exit invariants are associated with steps in the ConTract, and are concerned that certain conditions be maintained in the database for the success of later steps in the ConTract. Thus, a step *revalidates* the invariance of a piece of information according to a previous step's exit invariant. Weak conflicts differ from exit invariants in that they concern themselves with conditions that must be maintained in the multidatabase until an Interaction commits *regardless of whether or not that information is later used by that Interaction.* Thus, weak conflicts are detected as they occur. Also, ConTracts assume that an exit invariant is enforced either by preventing the conflicting operation (the one that causes the exit invariant to be violated), or by adjusting the data. Weak conflicts cause the Interaction that set up the conflict to be rolled back.

Although the ConTract work is similar to Interactions on the surface, it is proposed for use as the transaction manager in a single, possibly distributed, database. They require several special primitives in the underlying database and operating system to support the correct execution of the ConTracts on the database. For instance, they use special "existence locks" on objects to ensure that an object required by a compensating transaction is not deleted before a ConTract commits.

2.3.5 Multilevel (and Open Nested) Transactions

In a multilevel system, operations are nested, and each operation is defined as a sequence of atomic operations at the next level down. For example, a banking transaction initiated at level 2 (the top level) may be defined as some level 1 sequence of atomic inquire, deposit, and withdraw operations. The inquire, deposit, and withdraw operations are themselves defined in terms of level 0 atomic operations read and write. In a multilevel transaction model, the level structure is enforced rigidly by the internal call structure of the operations at the different levels – in the example level 2 operations *always* call level 1 operations, and level 1 operations *always* call level 0 operations. Conflict and correctness is defined between each pair of adjacent levels.

Weikum and Schek [Wei91, WS91a, WS91b] propose a formal definition for multilevel transactions, though not applied to multidatabases. They address issues of concurrency control and recovery. For correctness of multilevel transactions, they define *multilevel serializability.* In a correct multilevel execution, an operation at some level i is isolated at that level because it executes an atomic sequence of level (i-1) operations. A multilevel execution that is serializable has each set of the level i operations executed serializably by level (i-1). It is easy to see that standard transactions are a one-level restriction of multilevel transactions, where a transaction can only specify read and write operations. With this restriction, multilevel serializability is equivalent to conflict serializability [BHG87].

Multilevel transactions have been implemented as a part of the DASDBS prototype [Wei91, SPSW90].

Weikum and Schek do briefly explore modeling multidatabase transactions as two-level multilevel transactions. Their solution requires that the global concurrency control manager in the multidatabase maintain a notion of conflict among all of the transactions on the local databases, including both the global subtransactions and the independent transactions. However, this is not feasible in a multidatabase environment that tries to guarantee the autonomy of the local databases. This is because the restriction to a level-oriented transaction structure is incompatible with allowing local databases to run transactions independent from the multidatabase.

In [WS91a, WS91b], Weikum and Schek also define *open nested* transactions as an extension of multilevel transactions, though they do not attempt themselves to formally characterize correctness in open nested transactions. Figure 2.2 gives an example to illustrate the difference between an open nested transaction structure and a multilevel transaction structure. Figure 2.2(a) shows a multilevel transaction system. Figure 2.2(b) shows an equivalent execution that is not multilevel, but is open nested. T3 calls the read operation at level 0 directly. Weikum and Schek note that open nested transactions still maintain the nested structure, but do not enforce the level structure rigidly; thus, operations at a specific level may call operations at any level below it. By similar reasoning, a user can call operations at any level. Thus, this structure is more useful in the context of multidatabases with autonomous local databases. They conclude that conflict and correctness are harder to define in open nested transactions because they do not adhere to the level restriction, thus you cannot localize the conflict specifications to adjacent levels.

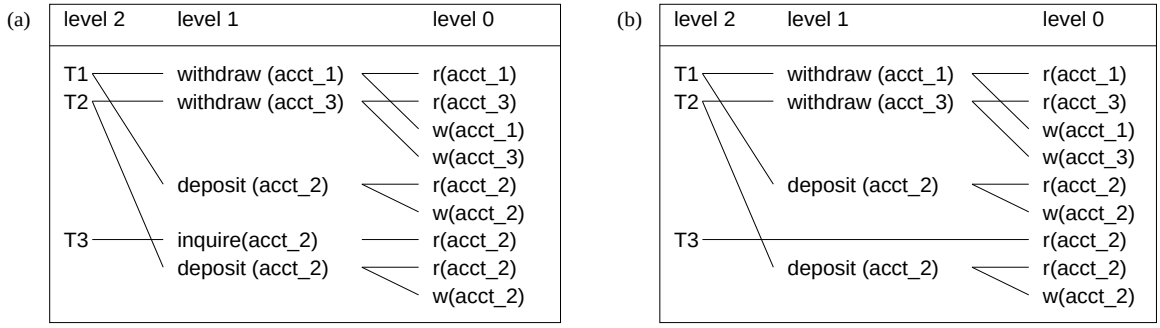


Figure 2.2: (a) a multilevel transaction system; (b) an equivalent open nested system that is not multilevel.

2.4 Active Databases

Active databases enforce long-term constraints on information in a database that are specified independently of the transactions actually run by the users of the database. As different transactions are run on an active database, the database itself monitors the changes made by each transaction to ensure that no constraints are violated. If some constraint is violated, then the database itself initiates some action to rectify the information in the database so that the constraint once again holds.

Examples of active databases include Cactis [HK86] and HiPAC [DBC+88]. Dayal *et al.* [DHL90] developed the event-condition-action (ECA) rules as a part of their active database work. ECA rules were the model we originally used when determining how to specify a weak conflict.

Recently, work has also been done by Buchmann *et al.* [BÖH+91] on active multidatabases.

Interactions resemble active databases in that they monitor the multidatabase for specific changes in the values of data items, and react to those changes. However, their response to the change is not to directly rectify the information in the multidatabase to make it consistent in some predefined sense, but rather to rectify the execution of the affected Interactions so that they remain internally consistent. That is, the response to a weak conflict violation is limited to partially or fully undoing an Interaction, and then possibly re-executing the Interaction again. Consequently, the lifetime of a weak conflict is bounded by the lifetime of the Interaction that set it. Interactions also use change in the database to pace their execution; for example, to ensure that a traveler has actually completed his trip before committing.

While reactive tasks can be supported by active databases, we view this process as conceptually different than activating the database. In particular, our requirements state that if a condition is violated, *the task* reacts to restore *its own* internal consistency. Thus, the constraints that define the interesting events really belong to a planning task. Also, since reactive databases require testing only for simple stability of individual items in the database, the predicates are much simpler. In an active database, the constraints are more complex and are independent of the transactions. When a constraint is violated, it is *the database's* responsibility to restore the consistency of its own data.

2.5 Compensation-Based Recovery

Once we allow Interactions or other open nested transactions to make visible portions of their work before they commit, we need to deal with the issue of rolling back the entire open nested transaction, including the committed portion. This is because the "all-or-nothing" requirement usually assumed for all types of transactions forces the model to support a complete (semantic) undo in case of transaction failure. In the case of open nested transactions, it is not feasible to do a state-based undo, which is the traditional undo model. Rather, the open nested transaction needs to *semantically undo* the transaction back to some state that is semantically equivalent to what would have happened if the open nested transaction had not run.

Open nested transaction models such as Sagas [GS87], Flex Transactions [ELLR90], and ConTracts [WR91] use a form of recovery called *compensation*. This form of recovery was first defined in detail by Garcia-Molina and Salem [GS87]. In their Sagas, each transaction has a corresponding compensating transaction. A correct

and complete execution of a Saga looks like a sequence of transactions:

$$T_1, T_2, \cdots, T_n.$$

However, the Saga may also be undone in midstream, causing the compensating transactions to be run in the inverse order of their corresponding original transactions:

$$T_1, T_2, \cdots, T_i, C_i, \cdots, C_2, C_1.$$

This approach requires that, as a Saga executes, its code and the compensating code are stored persistently. Nested Sagas [GGK+90] require that the compensating Sagas be invoked recursively.

The ConTracts paper [WR91] makes some additional observations concerning compensation. Compensating transactions do not need to run in the inverse order of their corresponding transactions: In fact, compensating transactions are often completely independent and can run concurrently.

Because open nested transactions make partial results visible early, recovery of one open nested transaction can cascade to others. ConTracts address this problem by explicitly maintaining all dependencies. This, however, is not feasible in a multidatabase environment.

Korth *et al.* [KLS90] did a study on compensation with respect to their entitywise 2PL correctness specification for transaction execution. They examine the effects of intervening conflicting transactions on the success of compensating transactions, and give some ideas on how to constrain executions so the resulting history is approximately sound. This paper provides a good characterization of some of the stickier problems that you can encounter when trying to implement any scheme that uses compensation.

Related to this work is the compensation strategy that is embedded in the Optimistic Commit Protocol [LKS91a]. While they support semantic atomicity (as in *Sagas*), Levy *et al.* also observe that the compensating transactions do not need to be coordinated as strongly because they are largely independent.

Karychniak *et al.* [KRST92] deal with issues of compensation in multidatabases that support assignment/deassignment transactions. In their scheme, if a global transaction is first done, then later semantically undone, the major effect this will have on the intervening transactions is that they may not get as good an assignment (because the semantically undone transaction is assigned something that the intervening transaction wants). They support a relaxed view of serializability, not quite so general as in Sagas, called *K-serializability*. With K-serializability, a global transaction and its compensating transaction can have at most K global transactions that conflict with them on any single data item between them. This bounds how far a transaction can be from its "best available assignment". To aid in their scheduling mechanism, they also introduce a notion of *horizon of compensation*, which states that a child transaction cannot be compensated for after its parent commits.

Multilevel transactions also require compensation-based recovery strategies [WHBM90]. Since multilevel transactions operate in a more restricted environment they can maintain an accurate, single log at each transaction level. During recovery, level 0 redo is first completed. Then, each level i log, starting from the level 1 log and working up, is processed in inverse order. The inverse log pass generates compensating level (i-1) steps for each level i transaction that has not terminated. This recovery strategy has been implemented in the DASDBS project. A similarly structured technique is proposed for nested and multilevel transactions in [Lom92].

[Wei91] notes several other approaches, with different strategies for maintaining (semantic) undo and redo information at the various levels. The checkpoint strategies take periodic checkpoints. After a crash, the checkpoint is restored and undo and redo operations are applied level by level until the database is brought up to date. These strategies require that semantic undo and redo information be maintained at each transaction level. Because of other limitations, the checkpoint schemes work best in a centralized database.

The second new class of recovery strategies noted in [Wei91] they call "subtransaction oriented". These strategies basically treat subtransactions as independent transactions at their level. A log is kept of the active and completed subtransactions at each level. When a crash occurs, the log is read level by level. At each level (except level 0 which is read/write only and uses state-based undo/redo), the subtransactions that had not been completed at the time of the crash are semantically undone. The subtransactions that did complete but are not reflected in the database are semantically redone.

Very little of this recovery work applies directly to the reactivity problems that occur with Interactions. Because Interactions are *reactive*, we support conflict- or event-driven partial rollback of committed portions

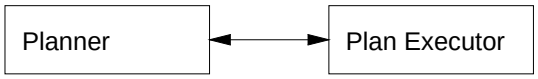


Figure 2.3: System design for an Artificial Intelligence planner.

of an open nested transaction. The characteristics of our reactivity problem are also quite different from those of recovery, in that we cannot actually find a spot in the log before which everything is correct, and after which everything is not correct. This is because of the reactivity bias towards preserving the original plan as much as possible, while still regaining Interaction consistency. In fact, we can't even point to a time in the Interaction execution where we know that everything after that time is invalid; certainly parts of the execution may be invalid, but others may not require any form of recovery action. For instance, in the example in Section 1.3.3, if the new flights are on the same day as the old flights, the car rental is still valid. We do not know of any other work that examines compensation in the manner required for reactivity.

2.6 Artificial Intelligence Planning Work

Much work on planning has been done by Artificial Intelligence researchers. This work examines planning in different domains such as planning the movement and actions of a robot or planning how to move different items around (such as blocks or packages). An overview of Artificial Intelligence planning work can be found in [DW91].

Much of the problem studied in this context focuses on gathering information and developing the plan based on the information that has been gathered. Thus, their system designs are as shown in Figure 2.6. The plan developer starts out with a set of specific goals to reach and a totally unspecified plan. Ultimately, a more detailed plan in developed. The final details are filled in as the plan is actually executed and real-time interference is dealt with. The actual process of developing the plan is a searching process, often structured hierarchically with lower levels representing more detailed refinements of the plan.

In our travel agent example, the goals represent the things the traveler wants with respect to his trip. The goals are not really explicitly specified; rather the planning application knows that one of the (well-specified) plan alternatives will achieve the goals. In our system, we focus more on deciding between the relative merits of different specific plans. In fact, we could put an AI-style planner on top of our planning application that feeds potential plans to the planning application (currently we assume this is a person). The plans do tend to be specified hierarchically, in that the more general plans are the ones that make the reservations for getting the traveler from town to town; the more detailed ones have to do with what the traveler would do in each place.

In this section, we discuss some of the planning work that most closely relates to our work. This includes work on reactivity in robots and other planning agents in the Procedural Reasoning System (PRS) by Georgeff et. al. [Geo87] and Reactive Action Packages (RAPs) by Firby [Fir89]. It also includes replanning and plan reuse research in the PRIAR system of Kambhampati [Kai90]. The plan reuse work is relevant to Interactions in that the problem of how an Interaction should react to a weak conflict violation is very similar to how Kambhampati formulates the problem of conservative plan reuse.

2.6.1 Procedural Reasoning System (PRS)

The *Procedural Reasoning System* is a flexible general-purpose planning system developed by Georgeff *et al.* [Geo87]. PRS considers both the making of a plan and the execution of the plan in the same system. PRS is designed to be a *reactive* planning system in the same sense that Interactions are reactive. When things in the environment impact the execution of a plan, then the system reacts by replanning. Unlike our Interactions work, PRS is designed for environments where there is frequent interference among plans. Thus, PRS not only considers replanning to accomplish the same goals, but also considers whether the interference with the plan should cause its goals to be modified, or even if the plan should be abandoned altogether.

PRS is designed to be a general-purpose planning and plan execution system, and thus bears more resemblance to our Interactions work than more specialized Artificial Intelligence planning systems. However,

they focus more on plan execution as a means to gather information which is fed back in the agent's planner and used to improve future plans. Thus, planning, replanning, and plan execution are deliberately interleaved. While Interactions can interleave replanning with execution, this interleaving is not deliberate.

2.6.2 Reactive Action Packages (RAPs)

In his thesis [Fir89], Firby considers adaptivity and reactivity in robots. Adaptivity concerns moving the robot around obstacles as it executes its plan. Reactivity is the ability for the robot to change what it is doing based on unexpected situations in the robot's environment. The robot system monitors the environment for these situations as it executes a task. If a monitor senses one, it can modify its plan to cope with the (higher-priority) problem.

In Firby's work, the requirements of reactivity are somewhat different from our reactivity requirements because of the different planning environment he works in. For one thing, a robot operates in real-time, and may need to execute an approximate plan quickly rather than making a perfect plan that is slower to construct. Also, a robot may consider several tasks simultaneously, executing the highest-priority one required at any time. For instance, a robot may interrupt a package delivery to sound a fire alarm. In our system, we consider each task independently and effectively assume one agent per task. RAPs also operate with a more limited set of knowledge that is found in a multidatabase. The robot's world knowledge is based on what it is initially told and what it has observed on its own. Much of Firby's work is in determining what kinds of information can actually be sensed and processed by a robot. Also, he needs to deal with uncertainty, in that the robot's world knowledge may not be accurate. The information in a multidatabase is generally both more complete and more accurate.

Firby's work, however, does consider some issues that we do not consider in this thesis. One important issue he considers is what to do when the reactions to different plans interfere in a way that causes a reaction loop. That is, one replanning effort triggers a second replanning effort that impacts the first replanning effort. We need to think about this problem in our own context. Firby also considers merging plans, which may or may not be useful in our multidatabase planning applications.

2.6.3 The PRIAR System

In his PRIAR system, Kambhampati [Kai90] examines issues of plan reuse and modification. Plan modification is very related to the Interaction's recovery work in that, when a plan becomes invalid we attempt to modify the old plan into a new one, making as few changes to the original plan as possible.

PRIAR annotates each plan with a set of internal dependencies that it uses when modifying one plan (slightly invalid in the current context) into a second plan (valid in the current context). This internal dependency information serves a similar purpose to our read/write information that we store for the different global transactions. Unfortunately for us, we cannot keep explicit dependencies because of the possibility of autonomous activity in the multidatabase.

The PRIAR system maintains similar requirements to Interactions. It attempts to be conservative in how much modification is made to the original plan; that is, it attempts to minimize how different the new plan is from the original one. It considers protections (similar to weak conflicts) that are set by particular subtasks and must persist for some period of time. It also considers how to eliminate *phantom tasks* from the plan – i.e., tasks whose effects are already true. The elimination of phantom tasks is motivated by the same issues that motivate us to eliminate replacement transactions in our reactivity work.

2.6.4 Terminology

In many cases our terminology differs from the terminology used by the Artificial Intelligence planning community. In this section we discuss the correspondence between our terms and their terms.

Planning in their parlance is the process of deciding how to achieve a particular goal. This is true in our terminology as well. However, in Artificial Intelligence systems this goal is specified as some property of the planner's recorded world knowledge, whereas in our environment it is determined by the specifier of the plan. In the Artificial Intelligence community, they speak of *making* the plan and *executing* the plan;

we, however, do not explicitly consider plan execution beyond providing the ability for an Interaction not to commit until the execution is complete.

In Artificial Intelligence planning, *reactive systems* execute a plan, monitor the execution and the environment for potential problems, and replan around those problems. Since the Artificial Intelligence planning work focuses more on autonomous planning agents that collect their own information, these agents react as they sense problems during plan execution (when they are "on the spot" to observe what they may not have anticipated). In our multidatabase environment, we have a common method to access information about the world that the planner may not know based on his own experience. We can use this advance knowledge to react earlier, before the plan is actually executed. Thus, our reactive system plans, monitors, and replans.

In Artificial Intelligence terminology, a *protection* is a condition that must be held for the plan to remain valid. We call this a *weak conflict*. They call the situation where a protection gets violated a *conflict*; we call this a *weak conflict violation*. The existence of a conflict is detected by *monitoring* the protections, which is what our event detecting functions do. When a conflict occurs, an Artificial Intelligence planning system enters into a *replanning* phase (or does *plan recovery*). This is the process by which the plan is brought bask into a state where it is possible to achieve its goal given the planning agent's current world knowledge. We call this process *reacting* or *evolving the plan into a new consistent state*. In this thesis, we avoid the use of the world *recovery* for this process as it is already a database term for bringing the database into a consistent state after a process, disk, or communication failure.

During the process of reacting or replanning, it is sometimes necessary to abandon some piece of the plan. With Artificial Intelligence planning agents this process is called *backtracking*. However, in the multidatabase realm the making of the original plan involved modifying the multidatabase (for example, as reservations are made). Thus, backtracking also involves undoing those effect on the multidatabase. We call the process of undoing these changes *compensation* or *semantic undo*.

Chapter 3

Specifying Tasks with Interactions

Interactions allow an application to define and execute tasks in a non-serializable way on a multidatabase. Each Interaction defines a complete procedure for executing the task in the multidatabase, including alternative ways of accomplishing particular subtasks, conditions that can't be violated by other tasks for this one to succeed as executed, and steps to be taken when some other task interferes and violates those conditions. Each Interaction also defines its own requirements for isolation from other tasks.

An Interaction is specified from some application (such as a travel agent application) as a program or command sequence. The multidatabase that serves the application contains an Interaction Manager that controls the execution of each Interaction and ensures that its isolation requirements are met.

In this chapter, we discuss the characteristics of planning tasks and the requirements they place on the underlying multidatabase transaction model. Based on this, we describe the Interactions transaction model, and explain how planning tasks are defined in that model. We then define the language TaSL (Task Specification Language), which we use to define tasks in our multidatabase implementation.

3.1 Characteristics of Planning Tasks

In this section, we examine in detail the characteristics of planning tasks and the requirements they place on a transaction model. Some of these characteristics have been discussed in Chapter 1, but here we really focus in detail on the tasks themselves. We see that planning tasks executed on a multidatabase do not necessarily have the same database access requirements as the type of task that is implemented as a transaction on a typical database. We also explore why traditional transaction models do not suffice to fill these requirements.

3.1.1 Identifiable

A multidatabase task is really a grouping of related operations on different databases that accomplish a single, specific purpose. It may not be wise to view different parts of the task as completely different pieces, because the relationships that form between those parts must be recognized and maintained. For example, if the customer misses his flight on his business trip, all of his travel plans may need to be adjusted. The example of Section 1.3.1 notes that all of the plans including the car rental may be affected if the customer's flight is canceled. Thus, it is important to be able to make conclusions about what to do about the car rental based on the status of the flight.

Another reason that a task requires a unique identity has to do with resource management. For example, the task may want to keep track of the status of the resources it needs for the correct recovery in case the task is aborted. If something happens to some required resource, the task needs to know that recovery is now no longer possible. Another case where the task needs to manage resources is when it is watching for operations that interfere with its work. For example, once the travel agent has made the flight reservation, he needs to monitor that reservation to ensure that it is not canceled. If the flight is canceled, then the task needs to do some work to recover its consistency. Clearly, the transaction associated with the subtask that made the reservation has committed, because the flight reservation is available for the airline to do the

cancellation. However, there is something (the task) that is still responsible for initiating the watch on the resource and for receiving and processing any notification that the reservation has gone away.

3.1.2 Non-Atomic

In a traditional database, the user manipulates information using atomic transactions that take the information in the database from one consistent state to another consistent state. Given that all transactions follow this guideline, a traditional database guarantees that the its information always remains consistent by ensuring that individual transactions are executed effectively in isolation *(atomically)*. Atomicity is usually enforced by guaranteeing that the database history is *serializable*. An atomic transaction is either completed, or any effects that it had on the database and on other transactions are removed. A transaction history is serializable if no transactions interleave their operations in such a way as to leave the database in a state that is not the same as some state that could be reached by executing those transactions in some serial order. Basically, this means that any concurrency among transactions does not cause any undesirable side effects.

Unfortunately, long-lived tasks such as planning tasks should not be represented to the database as atomic transactions [Gra81]. This is because atomicity requirements mean that concurrent transactions interfere with each other (e.g., by deadlocking). The longer the transactions tend to run, the worse that interference becomes. This, coupled with the fact that the autonomy requirements for local databases in a multidatabase can cause the multidatabase to make very conservative estimates on when interference occurs, means that atomic planning transactions would frequently need to be aborted due to conflicts.

Another reason planning tasks should not be atomic is because they do not necessarily need to maintain a *repeatable read*. Repeatable read guarantees that reading a specific data item more than once in the same transaction always yields the same result. In the travel agent example, we see that a flight is initially available and therefore a seat can be booked for the travel agent's customer. Later on, the flight is canceled, so the travel agent's query concerning the flight should reflect the change. In this situation, a non-repeatable read is considered correct.

Since it is not reasonable to throw out correctness completely and allow executions on a multidatabase to be completely uncontrolled, multidatabases need to guarantee different types of correctness than atomicity. Since portions of each planning task may need to run in isolation, each task can be specified as a partial order of atomic units of work, where the partial order indicates where one unit of work must logically follow another. This structure then can form a basis for a new, non-atomic correctness criterion.

Because of the all-or-nothing properties traditionally associated with transactions, multidatabase tasks are expected to execute in a way that preserves *semantic atomicity*. Semantic atomicity states that a transaction either runs or is left is a state that is *semantically equivalent* to some state that would have been reached if the original transaction had not run at all.

3.1.3 Recoverable via Compensation

Since planning tasks can commit some of their subtasks early, problems can occur when some transaction commits, and then the overlying task fully or partially aborts. Garcia-Molina and Salem [GS87] argue that in these situations, the only way to back out the committed changes is to ensure that some *compensation procedure* runs to completion. For each transaction, a compensating transaction is designed to reverse its effects. For example, if the transaction books a particular airplane reservation, the compensating transaction would cancel the reservation.

Compensation-based recovery is a sloppier form of recovery than the traditional *undo*. There is a window of time between the point where the original transaction commits and the point where the compensating transaction begins. During this time, other transactions may read the effects of the original transaction and use that information in their own execution. These transactions then are ultimately dependent on the effects of some transaction that, in some sense, didn't run [KLS90]. While these dependencies may be difficult to compute directly, it may be important to be able to determine their existence for compensation to execute correctly. Thus, in addition to maintaining the traditional logs and defining recovery procedures that use those logs, we need to be able to detect these dependencies or their effects during recovery.

One place this problem becomes apparent is when other transactions do operations that prevent a compensating transaction from completing. This occurs in the bankrupt airline example from Section 1.3.4.

3.1.4 Evolvable

The information in a multidatabase evolves over time. Planning applications make changes to the data in the multidatabase, and these changes serve effectively as *promises* for the feasibility or success of future endeavors. For example, a plane reservation is a promise that the traveler will be able to take the flight at the stated fare.

Interaction among tasks is implicitly allowed any time one task makes its results visible to another task before it commits, thus allowing the second task to act on those results before the first task is committed. In some cases, this interaction is desirable. For instance, if a travel agent has booked a flight that is canceled, it is desirable for him to find out about that cancellation as soon as possible, to give him the best chance of making other reservations.

Because of the atomicity requirement usually placed on database transactions, the traditional response to a conflict in a transaction is to abort the whole transaction. However, because planning tasks are long-lived and non-atomic, it is not necessarily desirable to abort the whole task and start all over again when a conflict occurs. Since a task can be of long duration, aborting it may involve undoing large amounts of work. Since a tasks is nonatomic, aborting it may involve semantically undoing committed work that is still valid. In contrast, our goal is to enter into a "reactivity" phase when some already-completed work becomes invalid. This phase should have as little impact on the rest of the task as possible while still regaining consistency.

Reactivity bears some resemblance to recovery, in that it requires similar information to be kept as that which is logged for recovery. However, in situations in which some subtask's work has been invalidated, the approach taken to regaining consistency in the reactivity phase differs from the recovery process. First, as with recovery, the process managing the reactivity needs to recognize that any subtasks that do not obviously depend on the invalid work (e.g., anything that was completed before the reservation was made on the plane flight that was canceled) should not be touched. Second, it needs to determine what of the remaining work was invalidated and what was not. In this situation, it cannot with certainty determine what the minimal set of recovery actions are until it is in the middle of the reactivity process. It cannot just point to a spot in the task execution and know that everything after that spot must be undone.

Thus, the reactivity manager needs more sophisticated ways of examining the different subtasks in the current multidatabase and Interaction context, and determining for each one in turn whether or not its work is valid in the current context. This involves trying to replace the work, in the original execution order of the subtasks, while impacting as little of the following subtasks as possible. Replacement can be done using compensation, and by redoing (using a different alternative where necessary). This same type of problem is encountered in the Artificial Intelligence planning community, where replanning algorithms are designed to impact the original plan as little as possible [Kam90].

3.1.5 Flexible

One of the goals for a planning task is not only to find a workable plan, but also to find the *best* workable plan. Thus, different ways of achieving the plan (or parts of it) are specified as alternatives. These alternatives are prioritized where needed. Which of the alternatives succeeds depends on the state of the underlying database at the time the steps execute. Thus, the actual execution path for the task depends on the state of the multidatabase at the time the task is executing.

The following types of flexibility have been defined for multidatabase tasks:

- Do some step in one of several ways; e.g., book either the United flight or the TWA flight.
- Allow some step to be optional; e.g., don't rent a car if no reasonable car is available.
- Allow divergent sets of steps; e.g., if you get a flight then rent a car at the destination, otherwise rent a car at the source and drive.

These types of flexibility are all useful in supporting planning applications. However, because we generate long-lived, non-atomic transactions, flexibility can be implemented at a higher level. For example, if the attempt to make the reservation on the United flight fails, it would be useful to commit that portion of the task's work (or maybe even abort it), so that the resources touched by it are no longer tied up. This leads us to conceive that we could divide the task up into a set of short atomic transactions as suggested earlier.

Figure 3.1: The execution of a planning task is bursty. Ellipses indicate potentially long periods of time where the Interaction is inactive.

These are committed or aborted soon after their (successful or unsuccessful) completion. Flexibility can then be expressed with respect to these short transactions, as opposed to being expressed on a step-by-step basis. Thus, the types of flexibility we support in Interactions take the following forms:

- Do one of several global transactions each of which accomplishes the same task; e.g., book either the United flight or the TWA flight.
- Allow some global transaction to be optional; e.g., if the global transaction to rent the car fails, don't abort the Interaction.
- Allow divergent sequences of global transactions to be executed; e.g., if the global transactions to get a United flight and get a TWA flight both fail, then rent a car at the source and drive.

Implementing flexibility in this manner has the advantage of being efficient in how long database resources are held. If a global transaction fails to accomplish a specific alternative, it can be aborted immediately to release the data items it holds.

One advantage flexibility gives us, beyond allowing the original plan to be selected among several alternatives, is to provide different ways of bringing a task back to a consistent state once a conflict occurs. In the trip example with flight cancellation, the traveler's original flight reservation no longer holds. However, the multidatabase can still examine the flexible Interaction specification (as originally defined by the traveler and the travel agent) to find an alternative airline booking. This selection process is very similar to what would have happened had the original booking failed immediately.

3.1.6 Paced

Planning tasks start when the plan begins to be specified and typically end when the execution of the plan is complete. Thus, planning tasks can span days or even months. Naturally, the whole time is not spent executing multidatabase transactions. Rather, a planning task typically generates bursts of work followed by long stretches of inactive time.

Figure 3.1 shows a potential execution of an Interaction. Here, the Interaction executes a few, short global transaction and has long periods of time where it is just waiting. For example, if this Interaction were planning a trip and the different global transactions represented different reservations, the commit may not occur until the trip is complete. Thus, once GTn commits, the Interaction has finished the updates it is making to the multidatabase and is merely waiting for an event representing the completion of the trip.

An Interaction enters an inactive period when nothing can be done currently, and remains inactive for a potentially indefinite period of time. Therefore, all of the Interaction's work should be committed before the inactive period begins. The inactive period can end under specific conditions such as:

- More input is given by the task specifier, so there is something to do next.
- Some event happens in the multidatabase that causes some replanning to occur, such as a flight cancellation.
- Some event happens in the multidatabase that indicates that some portion of the plan has executed, such as the traveler taking the flight.
- Some time constraint has been reached, such as a deadline for paying for the flight.

This means that planning tasks are partially event-driven. The occurrence of the events controls the pace at which the task executes.

3.2 Task Specification

An Interaction is a program that defines the flow of some task, and an associated set of conflicts that define the isolation requirements for the task. In addition, an Interaction may have local variables to hold internal information (including results of completed program steps) for use later in the execution. Each Interaction has a unique identity, maintained over its entire life.

The parts of the program that actually interact with the local databases are called *steps*. Each step is executed atomically on a single local database to accomplish some part of the Interaction's task. The Interaction program specifies how the different steps can be combined to accomplish the task correctly. This includes specifying alternative ways to accomplish the same subtask. For instance, if there are no more flights from airport *from* to airport *to*, the traveler may wish to rent a car and drive to his destination.

Programs may also specify steps and sequences of steps that may be attempted concurrently. Concurrency is possible when two actions are entirely unrelated, such as booking a rental car and a hotel reservation in a city, once the travel times are known.

In addition to defining the flow of a particular task, an Interaction also must define the task's isolation requirements. Isolation requirements are defined over spans of the execution of the Interaction. This model allows two forms of isolation. The first, called a *strong conflict*, defines an execution span where all of the steps are executed atomically as a single atomic multidatabase transaction.

The second form of isolation is called a *weak conflict*. It defines a set of conditions which should be preserved for a specific span of the Interaction's execution. If these conditions are violated, then the weak conflict also specifies what the contingency plans are. For example, a weak conflict might be specified that, once the traveler's flight is booked, that reservation stays around until he actually flies. If the reservation goes away, the weak conflict specifies where and how to begin the reactivity process. If reactivity is ever invoked, then the steps associated with the weak conflict are issued automatically by the multidatabase.

3.2.1 Example

Figure 3.2 gives a flow diagram for the example task from Section 1.3.1, with the added caveat that the traveler may wish to rent a car and drive if he cannot get flights at the appropriate times.

In this figure, states are represented by circles, with the start state indicated by a $>$. Transitions are indicated by arrows ($\rightarrow$), and are labeled with the global transactions that cause the transition. Some of the transitions in the diagram have more than one head. These indicate that there is a fork in the execution of the Interaction, and the portions of the flow diagrams at the heads of the transition may be executed concurrently. There also may be arrows with multiple tails, indicating the joining of two concurrent execution paths.

Final states are enclosed in a darkly shaded box. If there is a fork in the execution, then there may be multiple final states, all of which need to be reached for the Interaction to terminate successfully. These final state sets are indicated in the figure as a set of states that share the same darkly shaded box.

One other thing represented in this flow diagram is a place (state b) from which one or more alternative actions can be taken. In this case, flying is the preferred alternative, and that execution path is indicated by a [1]. A second choice is driving, and that execution path is given a lower priority ([2]).

Figure 3.2 also shows examples of a weak conflicts for the defined Interaction. The duration of each conflict is indicated by a lightly shaded box. The label associated with the box indicates an event that should occur during the span delimited by the box. The example weak conflict on the United flight ensures that the reservation stays around until the traveler actually takes the flight.

3.2.2 Overview of a Task Specification and Execution

A task is specified as a flexible, open nested transaction called an Interaction. An Interaction specifies different alternative ways of accomplishing its task, with priorities associated with each place the Interaction may need to choose among the alternatives. When a task is executed, it is assumed that exactly one of these different ways succeeds, and further that it is the highest-priority way possible given the state of the multidatabase.

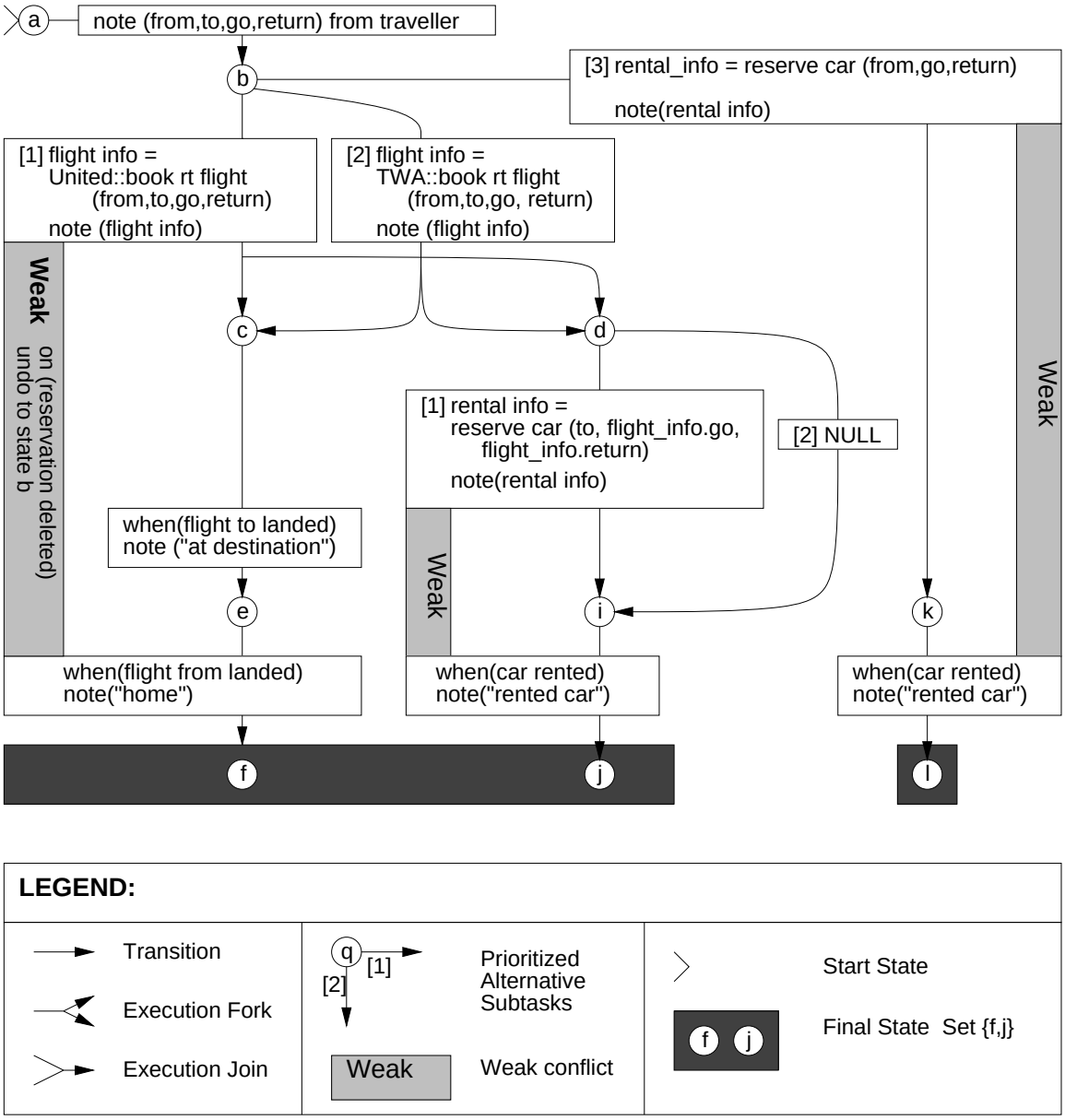


Figure 3.2: Task flow for a trip reservation. Example conflicts are shown in dashed boxes.

Different parts of an Interaction are specified as atomic subtasks. Each subtask is defined as a single global transaction. For example, in Figure 3.2, each box containing steps is assumed to be executed atomically. The flexibility, which allows the planning application to specify alternative ways of accomplishing the task, is specified as a prioritized set of choices to take from the current point; thus, the priorities are also shown for each alternative in Figure 3.2.

Each global transaction in the Interaction is specified as an atomic set of *steps* on the local databases. We assume that the steps are defined individually by the local databases, and specify the interface between the local database and the multidatabase. A global transaction may specify steps on more than one local database.

We assume the architecture described in Section 1.6, which is similar to the one proposed for Sagas [GS87]. The entire Interaction specification is saved in a layer between the application itself and the multidatabase. Thus, it is available during the Interaction execution and recovery, when different alternatives may need to be tried.

When an Interaction is executed, the best alternative set of global transactions is attempted, in the partial order defined by the specification. If some alternative fails to accomplish its subtask, the work done so far is minimally backed out of (that is, the fewest number of global transactions possible are compensated for) and the next highest priority set of alternatives is attempted. The resulting search through the possible different ways to accomplish the task is similar to a depth-first-search order.

In terms of the execution as viewed by the multidatabase, an Interaction looks like a partial order of global transaction executions. It can also be viewed as a task network.

3.2.3 Global Transactions

Strong conflicts define what other operations cannot interleave their execution with this Interaction. The conflicting operations need not be issued by other Interactions. They could, for instance, be operations initiated by some other, independent transaction on some local database.

For correct operation in the multidatabase, each strong conflict must be enforced by executing steps within a single atomic global transaction on the multidatabase. This transaction prevents other global transactions on the multidatabase or independent transactions on the local database from inappropriately accessing the resources used by the steps it contains. For example, since a strong conflict is defined between the time the travel agent finds an available reservation for the traveler and the time the travel agent actually makes the reservation, there must be a single atomic global transaction that finds the empty seat, blocks access to the seat until the reservation is made, and ultimately makes the reservation.

Global transactions can specify step statements that are to be executed on different local databases. Steps in a single local database that are a part of a single global transaction are all executed within a single transaction on the local database (a *global subtransaction*). The global subtransactions on the local databases that execute the steps of a global transaction must commit atomically with the global transaction commit.

3.2.4 Alternatives

In an Interaction definition, flexibility is specified as choices for continuing from a given state if that state is reached during the execution. These alternatives are prioritized, so the highest-priority one is tried first and so on, until the one unique alternative path succeeds in accomplishing the remainder of the task. For example, in Figure 3.2, booking a United flight, booking a TWA flight, and renting a car are all different alternatives to be considered when the Interaction is in state b.

Subtasks in an Interaction may be specified as *vital* or *non-vital*. A vital subtask must be accomplished for the task to succeed. A non-vital subtask does not need to be accomplished. Non-vital subtasks are expressed using the *NULL* alternative. In Figure 3.2, the subtask that reserves the car at the destination *(from)* is non-vital, so there is a *NULL* alternative (with priority [2]) from state d to state i. Thus, the car rental is tried first. However, if the car rental did not succeed, the *NULL* subtask succeeds automatically.

In other work [GGK+90, ELLR90], subtasks were explicitly tagged as vital or non-vital. If a non-vital subtask could not be accomplished, it still succeeded. We instead provide an explicit *NULL* alternative, because the non-vital construct is ambiguous in the presence of prioritized alternatives. For example, consider the case where there is a higher-priority non-vital alternative and a lower-priority vital alternative. If the

first alternative fails, do you assume it succeeds or do you try the second alternative? If you try the second alternative, what do you do if it fails? Also, there is the case where there are two equal-priority non-vital alternatives leading to different execution states. If both fail, what state does the task end up in?

3.2.5 Steps

A *step* defines a cohesive set of operations on a single local database. In the task flow, an example of a step is *United::book_rt_flight (from, to, go, return)*. Its function is to check the database for United airlines to see if they have available flights at the specified times, and make a reservation if both are found.

In this work, each step is implemented as a procedure call on the *Step Library* associated with the local database. The step is responsible for taking its arguments and issuing commands directly to the local database in its data manipulation language. The step call and return information are logged by the step.

Because compensation is used during task evolution and recovery, each step also has a compensating procedure defined for it in the Step Library. Since the step knows exactly what is run on the local database, it is responsible also for logging what its compensating step call and arguments are. If other information is needed for Interaction-level reactivity and recovery, such as the data items that were read and written and their values, it is the responsibility of the step to also deduce and log that information. This may possible require running extra queries on the local database within the same global transaction that the step is executing.

Steps are executed atomically. Therefore, the operations of a step must be executed within a single global subtransaction on some local database. Because the transactions on the local database are assumed to be atomic and recoverable, the pieces of those local transactions that execute the steps are also atomic and recoverable.

3.2.6 Events

An *event* is something that happens externally to the Interaction, but which influences the execution of the Interaction. An event is defined as an update of a specific data item that causes some condition to be violated.

There are two ways we use events in an Interaction definition. The first way is to delay the execution of the Interaction until some event has occurred. For example, the travel agent may wish to delay the commitment of an Interaction until the traveler has completed his trip. An example of this is shown in Figure 3.2, where the Interaction waits for the flight to land before entering final state f.

The second way events are useful is in noting when weak conflicts are violated. A weak conflict specification defines some event, and an event handler to be executed if the event ever occurs. This is explored in more detail in the section on weak conflicts.

3.2.7 Weak Conflicts

Weak conflicts indicate the conditions that should be preserved in the database during a specific execution span of an Interaction for it to be able to continue executing from its current state. In addition, a weak conflict defines what is necessary to recover when its conditions are violated. Weak conflicts do not indicate something that should be prevented necessarily, but rather indicate conditions that require the task to react if they are violated. Usually, a weak conflict violation indicates the full or partial invalidation of some step that has already been executed. Thus, the violation of a weak conflict could be treated analogously to the explicit aborting of some completed action in the Interaction and possibly by the explicit invocation of other recovery mechanisms.

It is the support for weak conflicts in the multidatabase that allows the planning task to be able to react to changing conditions and to replan around the changes.

Weak conflicts are not necessarily enforced by holding locks or any other equivalent function in the local database. For example, in the case where the plane flight is canceled, the traveler obviously does not want to hold the lock on his reservation. That would block his notification of the flight cancellation. Holding weaker notification locks is also not possible, as this functionality is not likely to be already supported by the autonomous local databases.

Instead, weak conflicts may be allowed to occur, but their occurrence must be externally detectable so that appropriate measures can be taken to regain the Interaction's consistency. Because of the autonomy requirements on the local databases, the detection of weak conflicts needs to be done by the multidatabase directly. This is because (with the exception of active databases), most local databases do not support the ability to compute when specified conditions are violated, while at the same time allowing the operations that cause the violation to continue. Instead, the multidatabase must monitor the local databases for weak conflicts itself. This can be done using a variety of techniques, including periodic polling and explicitly monitoring the input command streams to the local databases.

3.2.8 Concurrent Execution

We need also to allow unrelated subparts of a task to execute concurrently. For instance, in the trip example from Section 1.3.1, the hotel reservation and car reservation steps can happen concurrently once the plane reservation is made. However, fairly simple methods for specifying concurrency, such as *fork* and *join*, are adequate.

3.3 Interaction Definition Language

This section defines a language, TaSL (for Task Specification Language), for defining Interactions. TaSL is modeled primarily after block-structured programming languages.

In TaSL, an Interaction definition is enclosed between the statements *begin_Interaction* and *end_Interaction*. Following the begin statement are declarations for the variables internal to the Interaction. The values of these variables are considered part of the Interaction's state. After the declaration section is the body of the Interaction.

The Interaction definition language requires facilities to define alternatives, global transaction boundaries, variable assignment and use, strong and weak conflicts and parallelism. The following sections describe the language facilities available to describe these functions. A more complete, formal definition of TaSL is given in Figure 3.3.

At the end of this section, the travel agent example task is given as an example TaSL definition.

3.3.1 Variable Declaration, Assignment, and Use

Variable declarations are of the form <*type*> <*name*> or <*type*> <$name_1$>, ..., <$name_n$>. A type is either the standard *int*, *char*, etc. used in C or some object view of information on some local database.

Variables are used in the normal way. At any time, the results of a step can be assigned to an internal variable. This is done with a simple assignment statement:

$$< variable > = < step\_result > .$$

The type of the result must match the type of the variable. As in C, the components of the variable can also be addressed using a dot notation, e.g. *flight_info.airline*. When passed as a parameter, all variables use call-by-value semantics.

3.3.2 Alternatives

The representation of alternative actions that can fulfill the task is represented using a modified form of the standard if-then-else construct. An example of the full form of the construct is as follows:

if not { <$global\_transaction_1$> }
then if not { <$global\_transaction_{2a}$> **or** $\cdots$ **or** <$global\_transaction_{2m}$> }
then if not ...
then { <$global\_transaction_n$> }

```
<interaction> ::=          begin_interaction {
                                  <variable_decls> <body>
                           } commit_interaction
                           <subroutine_set>
<variable_decls> ::=       ϵ | <type> <namelist>; <variable_decls>
<type> ::=                 int | string | ...
<namelist> ::=             <name> | <name> , <namelist>
<body> ::=                 ϵ | <tl_statement> <body>
<tl_statement> ::=         <statement_list> | <alternative_list> | <atomic_block>
<alternative_list> ::=     if not { <equal_pri_set> } then <alternative_list> |
                           if not { <equal_pri_set> } then { <equal_pri_set> } |
                           if not { <equal_pri_set> } then NULL;
<equal_pri_set> ::=        <atomic_block> | <atomic_block> or <equal_pri_set>
<atomic_block> ::=         atomic { <statement_list> }
<statement_list> ::=       <statement> |
                           <statement> ; <statement_list>
<statement> ::=            <step_call> | <assignment_stmt> | <subroutine_call> |
                           <fork_stmt> | <join_stmt> | <wait_stmt> |
                           <weak_conflict_stmt>
<step_call> ::=            <db_name>::<procedure_name> ( <arglist> ) |
                           <label>: <db_name>::<procedure_name> ( <arglist> )
<label> ::=                <ident>
<db_name> ::=              <ident>
<procedure_name> ::=       <ident>
<arglist> ::=              <arg> | <arg> , <arglist>
<assignment_stmt> ::=      <ident> = <step_call> |
                           <ident> = <subroutine_call>
<fork_stmt> ::=            fork ( <name> ) { <body> }
<join_stmt> ::=            join ( <namelist> )
<wait_stmt> ::=            wait ( <event> )
<weak_conflict_stmt> ::=   on ( <event> ) { <abort_stmt> } |
                           until <label> on ( <event> ) { <abort_stmt> }
<event> ::=                <db_name> :: <name> , <condition>
<abort_stmt> ::=           abort ( <label> )
<subroutine_call> ::=      <procedure_name> ( <arglist> )
<subroutine_set> ::=       ϵ | <subroutine> <subroutine_set>
<subroutine> ::=           <type> <procedure_name> ( <arglist> ) { <body> } |
                           void <procedure_name> ( <arglist> ) { <body> }
<name> ::=                 <string>
<ident> ::=                <string>
<arg> ::=                  <string>
```

Figure 3.3: Formal Definition of TaSL.

In this construct, each alternative is represented by a block of code and has some priority. Two alternatives A and B of the same priority are joined with an **or** to indicate that they can be tried in any order and/or in parallel. The **if not** A **then if not** $B \cdots$ **then** N indicates an ordering to the priorities, where A should be tried first. If A should fail, then B should be tried, etc. If N fails, then the whole set of alternatives fails, and consequently the Interaction fails. If the definer wants to express that the subtask executed by the alternative global transactions is not necessarily vital, then the last alternative should be **then NULL**.

3.3.3 Global Transactions

In TaSL, global transactions are expressed as blocks of code that are to be executed atomically. This is defined using the *atomic* construct, which is of the form **atomic** {<body>}. Statements not contained explicitly in the body of some atomic block are executed individually in their own global transactions.

3.3.4 Steps

Steps are specified as procedure calls to a Step Library associated with the local database. The Step Library procedures generate the DML that is sent to the local database.

Each step is thus specified as follows:

$$< local\_db\_name > :: < step\_procedure\_call > (< arglist >)$$

The step returns information as reformatted and returned by the Step Library call.

3.3.5 Events and Waiting

Events in TaSL correspond to the updating of a specific data item in a way that violates some condition. Events are defined as follows:

$$< local\_db\_name > :: < data\_item > , < condition >$$

where the data item is some specific item in the local database, and the condition is some boolean algebraic expression. The variables in the expression may consist of constants or values read from the local database in which the event occurred.

When the event occurs, the Interaction Manager notifies the relevant Interactions by sending it the event specifier.

The *wait* command is used when an Interaction wants to pause in its execution until some event occurs. When a wait command is reached, the Interaction is blocked until the specified event occurs. Then the execution resumes with the statement after the wait. A wait command has the form **wait** (<*event*>).

3.3.6 Weak Conflicts and Aborting

Weak conflicts do not fit well into the block structure of the language. In particular, one possible use for a weak conflict is to guard some specific change made for the benefit of the user, such as a flight reservation. The specific nature of the weak conflict is not determined until the actual reservation is made, and the flight, date, and time are known. However, there may be a window of time between the point where the reservation is made and the point where the weak conflict is put into place. If this window exists, then there may be a point where the reservation could get canceled, but the weak conflict would not notice. To close this window, the weak conflict must be in place before the global subtransaction that makes the reservation releases its resources. This means that the weak conflict must be set within the same global transaction as the step that makes the reservation. However, the weak conflict needs to persist for a longer time, and thus most certainly should be released outside of that global transaction. This situation may be seen in the example at the end of this section.

Because of this, TaSL treats weak conflicts more like exceptions than like locks. It uses a structure derived from Dayal *et al.*'s event-condition-action (ECA) rules [DHL90] to define the exception. Since the weak conflict may be released at some point of time that is not the end of the Interaction, there must be an

option to specify the ending time of the weak conflict. This is done using an *until* statement, which specifies the label indicating the statement that ends the weak conflict.

The full construct for defining a weak conflict is as follows:

$$[\ \textbf{until} < label_1 >\ ]\ \textbf{on} < event > \{\ \textbf{abort}\ (< label_2 >)\}$$

where square brackets indicate optional parts of the construct. If the **until** statement occurs, there should be some statement "downstream" from the weak conflict declaration that has that label. During execution, the weak conflict is released atomically with the execution of the labeled statement. If no such statement is reached, the weak conflict persists until the Interaction ends.

The event in the **on** statement is a standard event as defined earlier. The handler typically specifies a step to abort. This does not necessarily mean that the global subtransaction that executed that step must be aborted. Likely, that transaction is no longer running. Rather, the effects of that step have been undone and recovery should be initiated.

An abort statement has the form **abort** (<label>) where *label* is the label of the statement that should be aborted. While we limit the handler in an event to be just an abort statement, more flexibility may be needed in the long term. [GSW87] claims that, in the implementation of the collaborative editor CES, it would have been useful for the user to have some explicit way of specifying user-oriented recovery actions. Thus, the handler may be expanded to include clean-up actions as well as actions to undo the work that is now invalid. However, to prevent circular triggering of handlers in the database, the handler should be restricted so as to not modify information in the multidatabase.

3.3.7 Concurrency

The standard **fork** and **join** primitives are used to express concurrency in an Interaction. When an Interaction begins a concurrent thread, it uses the fork construct and names the thread, as follows:

$$\textbf{fork}\ (< name >)\ \{< body >\}.$$

In this case, a concurrent thread named *name* is created with the same context as the current thread, and that thread executes the code in *body*. The concurrent thread completes at the end of the code in *body*.

If the fork statement occurs within a global transaction, the new thread is assumed *not* to be a part of the same global transaction as the current thread. Instead, the new execution thread executes outside the scope of that global transaction, with its (execution) predecessor being the forking global transaction. The continuing thread continues executing the forking global transaction. The one constraint on the execution of the Interaction is that the forked thread may not commit a global transaction until the forking thread's transaction has committed (for recoverability).

A thread can join with a named thread using a join. A join construct has the form:

$$\textbf{join}\ (< name_1 >, \ldots, < name_n >)$$

where $name_1$, ..., $name_n$ are names of concurrent threads. The thread that executes the join statement is called the *primary thread*, and the named threads are called the *joining threads*. When the join statement is reached, the primary thread first waits until all the joining threads are complete, then it executes the join statement. With Interactions, forks and joins can nest, which means that a primary thread can only join with some thread that it forked itself, or with some thread that was forked by another thread that it is joining at the same time.

3.3.8 Subroutines

To allow for more code modularity and reuse, we allow subroutine calls within Interactions. A subroutine call may return a typed value, which can be assigned to some Interaction variable. Subroutine definitions are of the form:

$$[< type >] < procedure\_name > (< arglist >)\ \{< body >\}$$

```
begin_Interaction {
  place from, to;
  date go, return;
  flight_info * go_info, * return_info;
  car_rental * rental_info;

  get_dates (&go, &return);
  get_cities (&from &to);
  Agent::note ("leaving from ", to, "on ", go);
  Agent::note ("returning to ", from, "on ", return);

  if not {                          /* Try a United flight first */
    atomic {
      book_United_flights (from, to, go, return);
      fork (T2) {
        atomic { book_Avis_car (go, return, to); }
        car_rented:  (wait (Avis::check_car_status) == RENTED);
      }
    }
    wait (United::check_flight_status (return_info.flight) == LANDED);
  }
  then if not {                     /* No go, so try TWA */
    atomic {
      book_TWA_flights (from, to, go, return);
      fork (T2) {
        atomic { book_Avis_car (go, return, to); }
        car_rented:  (wait (Avis::check_car_status) == RENTED);
      }
    }
    wait (TWA::check_flight_status (return_info.flight) == LANDED);
  }
  then if not {                     /* Oops, maybe just rent a car */
    atomic { book_Avis_car (go, return, from); }
    (wait (Avis::check_car_status) == RENTED);
  }
  then abort_Interaction;           /* Well, so much for that trip. */

  commit_Interaction;               /* This executes once the wait events occur */
}  /* End of this Interaction definition */
```

Figure 3.4: Interaction definition for trip reservation example.

As with steps, subroutines are allowed to return values. The return value may be assigned to an Interaction variable. Information may also be passed back to the main body of the Interaction through its global variables.

A subroutine call is of the form:

$$[< variable\_name > =] < procedure\_name > (< arglist >)$$

3.3.9 The Trip Example

Figure 3.4 shows the trip example from Figure 3.2 with its associated conflicts, defined in TaSL. Figures 3.5 and 3.6 contain the subroutine calls made by the code in Figure 3.4. In this example, *note* is the action that places information in the travel agent's private database.

```
/** Steps to book a United round trip flight **/
status book_United_flight (from, to, go, return)
{
  flight_to: go_info = United::reserve_flight (from, to, go);
  if (go_info.status == OK) then {
       flight_from: return_info = United::reserve_flight (to, from, return);
       if (return_info.status == OK) then {
            Agent::note ("departure flight ", go_info);
            Agent::note ("return flight ", return_info);
            on (United::check_reservation (go_info.reservation) == NOT_OK)
                 abort_to (flight_to);
            on (United::check_reservation (return_info.reservation) == NOT_OK)
                 abort_to (flight_from);
       }
       else FAIL;
  }
  else {
       United::delete_flight(go_info.reservation);
       FAIL;
       }
}

/** Steps to book a TWA round trip flight **/
status book_TWA_flight (from, to, go, return)
{
  flight_to: go_info = TWA::reserve_flight (from, to, go);
  if (go_info.status == OK) then {
       flight_from: return_info = TWA::reserve_flight (to, from, return);
       if (return_info.status == OK) then {
            Agent::note ("departure flight ", go_info);
            Agent::note ("return flight ", return_info);
            on (TWA::check_reservation (go_info.reservation) == NOT_OK)
                 abort_to (flight_to);
            on (United::check_reservation (return_info.reservation) == NOT_OK)
                 abort_to (flight_from);
            }
            else FAIL;
       }
  else {
       TWA::delete_flight(go_info.reservation);
       FAIL;
       }
}
```

Figure 3.5: Subroutine calls for the plane reservations in the trip reservation example.

3.4 Summary

In this chapter, we discussed planning applications and the requirements that planning tasks place on the multidatabase. We specified a transaction model, Interactions, to support planning applications and other applications with similar requirements. In particular, the Interaction model addresses the following issues:

- Breaking up a potentially long-lived task into a set of related shorter tasks each of which can execute atomically on the multidatabase.
- Specifying constraints that must hold on the multidatabase for the shorter tasks to be consistent.

```
/** Steps to book an Avis rental car **/
status book_Avis_car (go, return)
{
  rent_car:  rental_info = Avis::book_car (go, return);
  if (rental_info.status == OK) then {
       Agent::note {"car rental ", rental_info);
       on (Avis::check_reservation(car_rental.reservation) == NOT_OK)
            abort_to (rent_car);
       }
  else FAIL;
}
```

Figure 3.6: Subroutine calls for the rental car in the trip reservation example.

- Specifying tasks flexibly; that is, giving alternative ways to accomplish the task. Also specifying priorities among the alternatives so the multidatabase can find the best feasible alternative. This implies that the multidatabase must support some ability to search through the different alternatives.
- Specifying backtracking properties, so that if some constraint is violated and the task becomes inconsistent, the multidatabase can compute what backtracking needs to be done. This also allows the multidatabase to determine, once the backtracking is complete, how to continue the search for the best possible alternative.

The Interaction model differs from other flexible transaction models in that it implements flexibility above the global transactions – the global transactions are not themselves flexible, but they can be combined in different ways to accomplish the task. Another unique feature is that it supports reactivity, or the ability to backtrack and re-execute committed work if the task becomes inconsistent.

Interactions expect the multidatabase to present a procedural interface to the data in the local databases. In particular, each local database presents a set of procedures, or *steps* that the multidatabase applications can use to access its data. This interface is less restrictive that the *restricted access* schemes, in that the procedures in a restricted access multidatabase are themselves complete, atomic transactions. Interactions expect to be able to group steps together more flexibly into atomic global transactions, and to commit an entire set of steps globally and atomically. It is more restrictive than the *unrestricted access* multidatabase schemes, in that the multidatabase does not have unlimited access to the information in the multidatabase. However, steps do make an easy and intuitive way to specify the task at the global level, as they encapsulate the details of the local databases.

TaSL as a database language is the most procedural of the languages that support flexible executions. IPL [CBE93] specifies the subtransactions and dependencies independently, and has no concept of preferences among different alternatives. ACTA [CR90] is more of a formalism than a language. Other flexible transaction models do not define the language they use.

With respect to planning, TaSL is the only database language we know of that inherently supports a notion of backtracking to regain consistency. The planning languages used in the Artificial Intelligence community do not tend to provide features useful in manipulating databases, such as definition of atomic sequences of actions, or explicit commit and abort operations.

The purpose of this chapter was to provide a detailed notion of the types of tasks we are trying to support in this transaction model. In the next few chapters, we provide an underlying theory for the correctness of a task execution in a multidatabase.

Chapter 4

Interaction Correctness and Synchronization

In Section 3.1.2 we argued that transaction atomicity and serializable histories may not be an appropriate approach for defining correctness for planning tasks in multidatabases. Instead, we defined other ways of expressing the correctness of a planning task. Each Interaction defines its own expectations, including a procedure to express both the steps that need to be taken for the Interaction to complete and the ordering constraints on those steps. The Interaction definition also includes isolation information for different sets of steps.

At any point in the multidatabase execution, the ordering information for each Interaction is used to constrain the order its steps are executed in the multidatabase and in the local database. The isolation information is used to define the boundaries of the atomic global transactions that execute the steps. In this way, we ensure that the overall operation of each Interaction conforms to its isolation requirements.

In this chapter, we define correctness criteria for multidatabase executions whose Interactions do not specify weak conflicts. This is because violating a weak conflict results in entering a process to evolve the Interaction into a new consistent state. Consequently, maintaining correct operation with respect to weak conflicts means correctly detecting when the weak conflict is violated, and correctly reacting to the ensuing abort. In Chapter 6, we address correctness in the presence of weak conflicts and recovery.

4.1 Multidatabase Histories

This section lays some groundwork for defining correctness more rigidly. We formalize our correctness requirements in Section 4.1.6.

A multidatabase is a time-varying collection of local databases:

$$MDB = \cup_i LDI_i$$

where LDI_i is some local database in the multidatabase.

The set of information that a multidatabase can provide access to is the union of the sets of information available in its current collection of local databases. Note that while the multidatabase does not store extra information independent of the local databases, it may choose *not* to provide information that is stored in some local database.

The work done at each level in the multidatabase (i.e. in the multidatabase itself and at the level of the local databases) is done in atomic steps appropriate to that level. For instance, the steps available at the local level include reads and writes of individual data items. The *history* of a database at a given level is the partial order of those atomic steps as they were executed in the database.

In this section we define histories at both the multidatabase level and at the local level. We define a notion of a *merged history*, that encapsulates history information at both levels in the multidatabase. We also define correctness based on the merged history definition.

4.1.1 Interactions

An Interaction execution is defined as a partial order of steps, where each step is an atomic unit of work on a single local database. The execution of a specific Interaction is represented as follows:

$$I = \left(S_I, <_I^S\right),$$

where

1. S_I is the set of steps that are executed as a part of I, including also the delimiters *begin_interaction(I)*, *commit_interaction(I)* and *abort_interaction(I)*,

2. $<_I^S$ is a partial order of those steps as executed, where $S_a <_I^S S_b$ in the history if S_a and S_b share the same execution thread and S_a is executed before S_b, or if S_b begins a new execution thread and S_a was the last step before the fork in the forking thread, or if S_b reads some Interaction variable whose value was written by S_a,

3. for a specific Interaction I, *begin_interaction(I)* $<_I^S S_i$ for any $S_i \neq$ *begin_interaction(I)*, and

4. for a specific Interaction I, for $S_z \in \{$*commit_interaction(I), abort_interaction(I)*$\}$ and any $S_i \neq S_z$, $S_i <_I^S S_z$.

A committed Interaction I_c is a set of partially-ordered steps beginning with a *begin_Interaction(I_c)* and ending with by a *commit_interaction(I_c)*. Commitment here means that the task has run to completion, and its effects are all reflected in the local database. Since the commitment of the Interaction's steps (which all occur before the Interaction commit) make the Interaction's effects both visible and persistent, the Interaction commit does not need to provide any additional functions in that respect.

An aborted Interaction I_a is a set of partially-ordered steps and compensating steps, followed by an *abort_interaction(I_a)*. Aborting a partially-executed Interaction requires that the steps that have been committed be *semantically undone*. Aborting an Interaction is discussed in the next chapter.

4.1.2 Global Transactions

Interactions follow a two-level open nested model with respect to how their isolation requirements are specified. Thus, adjacent steps of an Interaction may be grouped together and run as a single, atomic global transaction over multiple local databases.

Global subtransactions execute the work of a single global transaction on a single local database. The set of global subtransactions that cooperate together to execute a single global transaction are said to *participate* in that global transaction. In the Interaction model, we restrict global subtransactions such that a global subtransaction can only participate in a single global transaction. Therefore, the lifetime of a global transaction encompasses the lifetime of each of its global subtransactions. Also, no two global subtransactions that participate in the same global transaction execute on the same local database, because the local database itself could not ensure the atomicity of all the steps executed as a part of several global subtransactions, even though they are a part of the same global transaction.

The steps executed by the global subtransactions that participate in a single global transaction are atomically committed when the global transaction commits.

The partial order of global transactions in an Interaction execution can be derived from $<_I^S$:

$$I = \left(T_I, <_I^T\right),$$

where

1. T_I is the set of global transactions that are executed as a part of I, plus the delimiters *begin_interaction(I)*, *commit_interaction(I)* and *abort_interaction(I)*,

2. $<_I^T$ is a partial order of those global transactions as executed, where $T_a <_I^T T_b$ in the transaction history if for some $S_a \in T_a$ and $S_b \in T_b$, $S_a <_I^S S_b$.

4.1.3 Interaction History

The Interaction history contains an interleaving of the global transactions of a set of concurrent Interactions. Intuitively, an Interaction is represented in the history as a partial order of global transactions, beginning with a *begin_interaction(I)* and ending with either a *commit_interaction(I)* or an *abort_interaction(I)*. The part of the history where the Interaction was active is exactly the part between the two delimiters. For each Interaction active in the Interaction History, its global transactions must be ordered correctly with respect to $<_I^T$.

Formally, an Interaction History containing a set A of concurrent Interactions is represented as a partial order:

$$IH = \left(HE_I, <_{IH}^T\right),$$

where

1. $HE_I = \cup_{I \in A} T_I$,
2. $(\cup_{I \in A} <_I^T) \subseteq <_{IH}^T$, and
3. for any two global transactions T_a and T_b that run on the same local database, either $T_a <_{IH}^T T_b$ or $T_b <_{IH}^T T_a$.

Intuitively, this definition states that the global transactions in the Interaction History are executed serializably with respect to each other. Note here that we assume that any two transactions that execute on the same local database conflict. This overly conservative estimate is necessary because, without examining the executions of all transactions (global subtransactions and independent transactions) on the local database, we cannot determine when a conflict actually occurs.

4.1.4 Local Database History

We define a local database history in a manner similar to that of Bernstein *et al.*[1] A local history contains a set of n local transactions $t_1 \ldots t_n$, where t_i can be either a global subtransaction or an independent transaction. Each transaction t_i is represented in the local history by a partial order of local steps $<_i$. These steps form a partial order

$$LH = (HE_L, <_{LH}),$$

where

1. HE_L is the set of local database operations in the transactions $(\cup_i t_i)$,
2. $(\cup_i <_i) \subseteq <_{LH}$, and
3. for any two conflicting local database operations $p, q \in HE_L$, either $p <_{LH} q$ or $q <_{LH} p$.

Here, conflict is defined and enforced by the local databases.

The local database history is assumed to be serializable. Its *local serialization order* is the partial order of the transactions in the equivalent serial execution. We can derive the local serialization order for a database from its history, by examining the relationships between the transaction operations on the local database. This is because the order in which conflicting operations execute defines the order in which transactions must be serialized. The local serialization order is a partial order:

$$LSO = \left(t, <_{LS}^T\right)$$

where

1. t is the set of all transactions (global subtransactions and independent transactions) in the local database, and
2. if $\{s_i, s_j\} \subseteq HE_L$, $s_i \in t_i$, $s_j \in t_j$, and $s_i <_{LH} s_j$ (that is, s_i and s_j conflict in the local database), then $t_i <_{LS}^T t_j$.

[1] [BHG87], page 29.

Because we assume that the local databases are autonomous, we must also assume that an operation-level knowledge of the local database histories is unavailable to the multidatabase. However, given some knowledge of the concurrency control mechanisms of the different local databases, we may be able to deduce the local serialization orders. Also, small violations of local database autonomy, such as inserting a ticket in each local database schema for use by a forced local conflict scheme could be adequate to support the needs of the multidatabase.

4.1.5 Merged History

The merged history MH is formed by merging the histories of all the local databases forming a relationship (not necessarily a partial order!) $\rightarrow_{MH}$ defined as follows:

$$MH = (T_M, \rightarrow_{MH})$$

where

1. $T_M = T_G \cup T_I$.
2. T_G is the set of global transactions on the multidatabase.
3. $T_I = \cup_{L \in \text{MDB}} t_I^L$ (where t_I^L is the set of independent transactions that ran on local database L),
4. if s_a and s_b are subtransactions on a local database, with s_a participating in global transaction T_a and s_b participating in global transaction T_b, and $s_a <_{LS}^T s_b$ in the local database history, then $T_a \rightarrow_{MH} T_b$, and
5. if $T_i <_{IH}^T T_j$, then $T_i \rightarrow_{MH} T_j$.

The merged history reflects the operation of the multidatabase, including both the Interactions' executions and the executions of the independent transactions on the local database. It is this history that we use in defining correct operation for multidatabases.

Note that $\rightarrow_{MH}$ is not necessarily a partial order, because if two steps are executed in the same local database, their execution order in the local history may not be the same as the completion order of their steps reflected in the Interaction history. As an example of this, say Interactions I_1 and I_2 both have steps to make reservations on the same flight, and to rent a car at the destination. If their steps execute concurrently, then it is desirable for one Interaction's steps to consistently occur before the others across all databases. Consider the case where there is exactly one reservation left on the flight, and exactly one car. If a consistent ordering of the Interactions is not enforced, then the following effective execution order is possible:

1. I_1 gets the flight reservation from the airline database.
2. I_2 gets the rental car reservation from the car rental database.
3. I_1 tries to get the rental car from the car rental database and fails.
4. I_2 tries to get the flight from the airline and fails.
5. I_1 aborts the trip.
6. I_2 aborts the trip.

In this case, both trip reservations failed, even though there was a flight and a rental car that one of them could have taken. This is because the two sets of concurrent global subtransactions associated with the two Interactions did not serialize in a consistent order.

Given that I_1 and I_2 both execute their steps as a single atomic global transaction, then their subtransactions should not serialize in an inconsistent order, and the above execution should be prevented. Thus, a correct merged history should not contain this kind of anomaly. Intuitively, this is because $\rightarrow_{MH}$ should define a *global serialization order* for the global transactions in the merged history. The different local databases should thus serialize their global subtransactions in an order consistent with the global serialization order.

In a serializable merged history, then, $\rightarrow_{MH}$ forms a partial order called the *global serialization order (GSO)*:

$$GSO = (T_M, <^T_{GS})$$

This is the order in which the global transactions are actually serialized in the multidatabase. The global serialization order also specifies the order in which the global subtransactions are serialized in the local databases (in a correct execution).

4.1.6 Correctness Requirements

With Interactions, the unit of isolation is the global transaction. Ignoring the issues of durability and compensation (which are addressed in Chapter 6) we have the following definition for correct execution of global transactions in an execution of concurrent Interactions:

Definition 4.1.1 *A merged history is* correct *if it is serializable, and if for each Interaction,* $<^T_I \subseteq <^T_{GS}$.

The first requirement for correctness ensures that the global transactions are in fact atomic. The second requirement ensures that the internal consistency of the Interaction, as defined by the user, is maintained in its execution on the multidatabase.

Theorem 4.1.1 *The set of committed global transactions and independent transactions in a merged history for some multidatabase is* serializable *if and only if the following conditions hold:*

1. *The subtransactions of the global transactions on each local database are all serialized in the respective local histories in an order consistent with some unique global serialization order. That is, if any two global transactions* T_a *and* T_b *execute global subtransactions on the same local database (call them* s_a *and* s_b*), then* $s_a <^T_{LS} s_b$ *iff* $T_a <^T_{GS} T_b$.

2. *All subtransactions involved in the same global transaction are committed atomically.*

PROOF: Define the *merged serialization graph* as the graph constructed by taking the union of all of the (acyclic) local serialization graphs and merging all nodes that represent global subtransactions in the same global transaction. Since no global transaction has more than one global subtransaction on the same local database participating in it, we are never merging two or more nodes associated with the same local database. The resulting graph contains one node for each global transaction, and one node for each independent transaction in some local database. It contains all the information concerning serialization restrictions from all the local databases.

First, consider what happens if a global transaction does not commit atomically. Assume global transaction T accesses data in two local databases X and Y using subtransactions x_a and y_a, respectively. Assume without loss of generality that the commit of x_a happens before the commit of y_a. y_a may then later abort, ultimately causing x_a to have to be semantically undone. However, the results of x_a were visible in X for a span of time, violating the global transaction's atomicity constraints, and therefore also violating serializability.

Now, assume all global transactions are committed atomically. Examine the merged serialization graph. We know[2] because the local histories are serializable that no cycles are formed in the merged serialization graph that are induced by a single local database history. Therefore, all cycles must involve multiple local database histories. However, if all global transactions are serialized on the local databases in the same order (the specified global serialization order), then all paths spanning multiple atomic blocks in the merged serialization graph must reach those atomic blocks in an order consistent with the global serialization order. Therefore, there are no cycles in the merged serialization graph that were induced by the merging process. The merged serialization graph is acyclic, and the merged history is serializable.

Now, assume that there is a cycle N_a, N_b, ..., N_n, N_a in the merged serialization graph. This cycle must involve edges induced by multiple local histories, because each local history's own serialization graph is acyclic. For each local database whose transactions are involved in the cycle, find a path in the graph induced by its history that contains all nodes representing its transactions in the cycle. This path traverses the nodes

[2] [BHG87], p. 33, Theorem 2.1.

in the local serialization order for that database. Because there is a cycle, we know that at least one of these paths traverses at least two nodes in the opposite order from the global serialization order. Therefore, the local database represented by that path did not serialize its global subtransactions in an order consistent with the global serialization order. □

4.2 Interaction Scheduling

Interactions' global transactions are allowed to interleave arbitrarily in the merged history; the only order enforced for them is the order declared by the Interaction specifier. Given that we can use the mechanisms discussed next in Chapter 5 to enforce global transaction atomicity (or any of the other, published schemes discussed in Chapter 2), scheduling the execution of the global transactions is fairly simple.

Scheduling of Interactions has basically two requirements. The first is that the different steps in the Interaction should be executed in the defined order and grouped into global transactions as specified. The second is that the serialization order of the global transactions for each Interaction be consistent with their specified execution order.

4.2.1 Scheduling Global Transactions

At any specific time, the different in-process Interactions in the database have one or more concurrent threads executing the different tasks and subtasks. The Interaction scheduler maintains these as a set of *active* Interaction threads and a set of *waiting* Interaction threads. An Interaction thread is in the active set if its next action can be scheduled for execution immediately. An Interaction thread is in the waiting set if it has no current action to execute.

When executing an Interaction, the Interaction Manager basically executes a search to find the most desirable alternative set of steps that succeeds in accomplishing the task. Since conflict is enforced at the local level by the proper use and ordering of the global subtransactions, we can view each Interaction as an independently-schedulable execution thread.

The following is the algorithm for what an Interaction thread should do when it is at a specific point in its task where it needs to execute the next global transaction:

1. Choose the highest-priority next global transaction from the current state in the Interaction that has not yet failed. If none exists, semantically undo the "parent" global transaction and indicate that that failed. This causes alternative global transactions to those attempted at an earlier point to be explored. If we allow the Interaction definition to be changed interactively, another option here would be to allow the Interaction definer to specify a new alternative, and then attempt that global transaction.

2. Begin the global transaction.

3. Try the steps of the global transaction in the defined sequence. This involves possibly beginning a global subtransaction on the local database, then making a call to some procedure. The arguments for the procedure call are taken from the arguments to the action and the information specified by the user. There are three possible outcomes:

   (a) The step is aborted. This is not necessarily a failure; for instance it could have been involved in a deadlock. In this case, the step may be allowed to fail, or it may be retried. If the step is allowed to fail, abort this global transaction and go to step 1.

   (b) The step is not aborted, but the results are that the step failed to accomplish its subtask. Abort the global transaction (it failed) and go to step 1.

   (c) The step is not aborted, and it accomplishes its subtask. In this case, the step succeeds and the action succeeds. Return the results and log them in some stable location.

4. Commit the global transaction (it succeeded) and go to step 1.

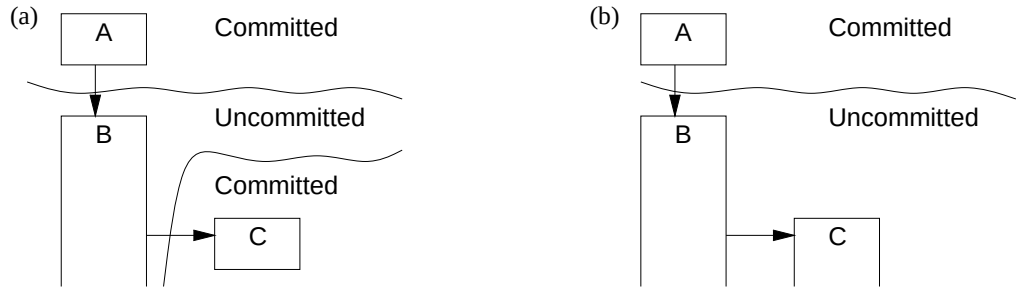


Figure 4.1: Various forms of current global transaction executions in an Interaction: (a) Uncommitted Global Transaction B has forked a second execution thread, and has a successor committed Global Transaction C. (b) Same Interaction execution, but in this case, Global Transaction C is restricted to not commit until Global Transaction B commits.

4.2.2 Forking and Joining Execution Threads

New execution threads can be forked from an existing Interaction thread at any time. If the forking thread is not currently executing a global transaction, and if T_a was the last global transaction that ran on the forking thread, when the new thread begins a global transaction T_b the two transactions will be ordered with $T_a <_I^T T_b$. If the forking thread is in the middle of a global transaction T_c when the fork statement is executed, then the new thread must begin its own global transaction T_d outside of the sphere of control of the global transaction in the forking thread. In this case, the two global transactions will be ordered in the Interaction with $T_c <_I^T T_d$.

Any time an Interaction thread is not actively executing a global transaction, it may specify other threads to join with it. These joining threads must be ones that the thread has in the past (transitively) forked, thus forks and joins for an Interaction are nested. The joining threads must all have completed their execution before the join itself can be executed (i.e., they can have no active global transactions). For each joining thread, if T_y is the last global transaction that executed before the join and T_z is the first global transaction to execute on the joined thread, then $T_y <_I^T T_z$.

4.2.3 Scheduling Global Transaction Commits

When the new thread executes its first global transaction, care must be taken to ensure that that global transaction does not commit before the global transaction in the forking thread that precedes it. This could happen if the forking thread is in the middle of executing a global transaction when the fork statement is executed. If the global transaction in the forked thread were to commit first, we would have the situation such as the one shown in Figure 4.1(a). In this situation, if the forking global transaction B were to abort, then global transaction C would have to be semantically undone.

To avoid this undesirable situation, we control not only the serialization order but also the commit order of the global transactions. Specifically, we delay executing commit instructions as needed so that the following property holds for the Interaction:

Definition 4.2.1 (Open Nested Recoverability Property) *The* open nested recoverability property *for an Interaction states that no global transaction can commit until all global transactions that precede it in* $<_I^T$ *have committed.*

If this property is enforced, situations such as the one in Figure 4.1(a) would be avoided, some delays in commit processing would occur, and the analogous executions such as the one in Figure 4.1(b) would be produced instead.

4.3 Summary

In this chapter we discussed the correctness of a history of concurrent Interactions, in the case where there is no reactivity (no weak conflicts are violated). First we formally defined Interactions and global transactions. Then, we used these definitions as a basis for defining what a correct history should look like at the global level of the multidatabase (the Interaction History). Intuitively, a correct Interaction history should be an arbitrary interleaving of the global transactions of the different Interactions, where each Interaction's global transactions are ordered in the history consistently with the order the global transactions were defined/executed in that Interaction. The order of the global transactions in the Interaction History becomes the *global serialization order (GSO)* in the multidatabase.

Different local databases have their own histories and serialize their own transactions according to their own concurrency control mechanisms. A single global transaction may execute several global subtransactions, one on each local database. The history of the multidatabase is really a merging of all the local histories, with the different subtransactions of a global subtransaction labeled as belonging to the same global transaction. A merged history is defined to be correct provided that it is serializable and consistent with the Interaction History. Basically, this means that the different local databases need to serialize the global subtransactions in an order consistent with the global serialization order. This requirement has been recognized for years [GP86]. However, this is the most comprehensive formal treatment of correctness that we have seen.

In this chapter we gave algorithms for scheduling and executing Interactions given their flexible definitions. No other such scheduler has been proposed for flexible transactions, though the algorithm is very similar to the basic searching algorithms used in Artificial Intelligence planners.

One last new contribution of this chapter is to define properties on the commitment order of the different global transactions in the Interaction to preserve the ability to easily react/replan and easily recover.

Basically, we can summarize the two requirements on the execution of an Interaction as follows: (1) its global transactions must serialize in the partial order specified in the Interaction definition, and (2) its global transactions must commit in the order they are serialized.

This chapter provides the foundations for the remainder of the formal work on correctness, synchronization, and recovery in Interactions. In the next chapter (Chapter 6) we define correctness in merged histories where weak conflicts have been violated. In the following chapter (Chapter 5), we expand on the requirements for correctly executing atomic or semantically atomic global transactions in the multidatabase. These global transactions are the basic building blocks from which Interactions can be constructed.

Chapter 5

Weak Conflict Events and Reactivity

In Chapter 4 we argued that Interactions themselves need to enforce a notion of correctness over broad portions of their execution, beyond the simple notions of the atomicity of their global transactions. In particular, each atomic global transaction has certain effects on the local databases, and these effects need to be preserved to some degree beyond the execution of that global transaction. Potentially, a global transaction may have effects on the local database that should hold up to the time the Interaction commits.

Because of this, we introduced the notions of *weak conflicts* and *reactivity* to our Interaction model. A weak conflict is a condition that is set on the data in the multidatabase during the execution of some global transaction, and that should hold beyond the time that global transaction commits. A weak conflict belongs to the Interaction itself, and defines data consistency properties that should hold during the Interaction's lifetime so that its execution remains correct (i.e., maintains the consistency of the underlying task).

The second property, reactivity, states that the Interaction must itself react when one of its conditions (weak conflicts) is violated. Reactivity is the basic scheme to maintain Interaction consistency, just as concurrency control protocols such as two-phase locking are used to maintain the consistency of an atomic transaction. The primary difference in approach is that schemes like locking enforce atomicity by preventing other transactions from interfering with the running transaction. Reactivity allows that interference but requires additionally that the multidatabase be able to evolve an Interaction into a new consistent state when a conflict does occur.

In this chapter, we focus on *events* generated because of weak conflict violations. We discuss how events are detected and reported in the multidatabase. Furthermore, we discuss *reactivity*, and how we support in the multidatabase the ability for an Interaction to regain consistency after a conflict has occurred.

5.1 Events and Event Processing

In this section, we focus on the role of database events in the execution of an Interaction, including what event types are useful in supporting the requirements of a planning task and how events are specified, detected and used.

An *event* is an update operation on some piece of information in the database that causes some condition involving that data item to be invalidated (for instance, a flight being canceled). There are typically different purposes for events, including:

1. *Wait events.* These are events on the local database that are explicitly being waited for by some execution thread (i.e., those specified in a *wait* command). Wait events are no longer "interesting" as soon as they occur.

2. *Weak conflict events.* These are events on the local database that indicate that a weak conflict has occurred (i.e., those specified in a current *on* statement). Weak conflict events are associated with global transactions. When the weak conflict occurs, the associated global transaction is aborted or otherwise undone. Weak conflict events cease to be "interesting" when they are no longer relevant to the Interaction, when the Interaction terminates, or when the global transaction associated with the weak conflict is semantically undone.

The multidatabase monitors for events and detects them on each local database individually. An *Activator* associated with each local database examines the database periodically to check for weak conflicts, and notifies the global level of the multidatabase when a weak conflict has occurred. When a weak conflict occurs, the multidatabase must change the Interaction to be consistent with the new multidatabase state. Because this change requires executing different alternatives from the ones that succeeded initially, this change requires understanding the entirety of the Interaction definition. Because of this, the actual process of deciding how to react to an event occurs in the Interaction storage and management layer (as defined in Section 1.6). This layer stores the Interaction definition as well as the information on which global transactions in the Interaction have been attempted and which have succeeded in their subtask. This information is required when deciding how to re-execute the plan, as the multidatabase is essentially continuing the search for the best feasible alternative.

5.2 Compensation-Based Reactivity

Each weak conflict is associated with the global transaction that initially set the condition on the local database. A weak conflict event indicates that that global transaction's work is now invalid, and that the Interaction must react. In this situation, we call the global transaction associated with the weak conflict the *abort point.*

When a weak conflict occurs, the Interaction reacts by considering the global transaction at the abort point, as well as the global transactions that transitively depend on it. It is this set of transactions that must be brought into consistency with the current state of the multidatabase, and which must further be made internally consistent. Thus, each of these global transactions must be considered during the reactivity process.

One obvious approach to implementing reactivity is using compensation. Compensation was first defined explicitly in the context of Sagas [GS87]. Recall from the related research section that a correct execution of a Saga looks like a sequence of transactions:

$$T_1, T_2, \cdots, T_n.$$

If the Saga may also be undone in midstream, the compensating transactions are run in the inverse order of their corresponding original transactions:

$$T_1, T_2, \cdots, T_i, C_i, \cdots, C_2, C_1.$$

Sagas are fully-ordered sequences of global transactions. We can generalize this concept to open nested transaction models where the global transactions are partially-ordered in order $<_I^T$, as defined for Interactions in Section 4.1.2. In this case, the compensating transactions must be ordered in the inverse of $<_I^T$.

The compensation-based reactivity strategy does not consider backing out of the entire Interaction. Rather, it backs out of the Interaction through the abort point, and then continues the Interaction execution from the resulting state. Thus, we have a compensation-based algorithm for reactivity that works as follows:

Algorithm 5.2.1 (Compensation-Based Reactivity) *Input: The abort point.*

1. *Let the* invalid portion *i of the Interaction be the set of global transactions in the Interaction that includes the abort point and all of the global transactions that (transitively) follow it in* $<_I^T$.
2. *Compensate for the global transactions in i in the inverse order of* $<_I^T$.
3. *Re-execute the Interaction once the compensation is complete.*

With this algorithm, we assume that compensating global transactions can be generated for each global transaction in the Interaction. Recall from Chapter 3 that each global transaction is specified as a sequence of *steps* on different local databases. In this work, we assume that each step has an associated compensating step, and that the compensating step call can be derived during the step execution from the step's parameters and its return values. We can thus construct and execute the compensating transaction for a committed global transaction GT as follows:

Algorithm 5.2.2 (Compensating Transaction Execution) *Input: The global transaction identifier* GT *of the transaction being compensated for. The log of the derived compensating step calls.*

1. *Select from the log, in the order the original steps were executed, the compensating step calls for each step executed as a part of GT. This is the sequence C.*
2. *In inverse order, execute the compensating steps whose calls are specified in C.*

Note the similarities between the execution of the global and compensating transactions in an aborted Saga, and the execution of steps and compensating steps in an undone global transaction.

5.3 Problems with Compensation-Based Reactivity

Unfortunately, there are several problems with applying the compensation-based approach to the problem if implementing reactivity in the Interaction model. In this section, we give examples of these problems and examine them in more detail. Then in the next sections we outline the replacement scheme, which avoids some of these pitfalls.

5.3.1 Non-Persistence of Compensation

Persistence of compensation states that all (compensating) subtransactions in an open nested transaction must succeed if retried enough times. This requirement was first identified for Sagas [GS87]. While persistence of compensation may hold true in restricted situations, it does not hold true in a general open nested transaction environment. Open nested transactions periodically release the resources they hold, allowing other database transactions to interleave with their own operation. This release of control by the open nested transaction implies that the other transactions may do things that prevent compensation from succeeding.

Consider as an example, a travel agent making a reservation for a customer on a Pan Am flight. The customer pays for the flight and gets his ticket. Then, before the customer actually takes the flight, Pan Am goes bankrupt. The customer may no longer be able to go on the trip, but fully canceling the Pan Am flight and getting a refund is impossible. The customer has relinquished control over his money. [1]

In the early work on Sagas [GS87], a concept called *recovery blocks* was proposed. A recovery block provides an alternate method of compensating a particular action, under the assumption that if a particular method fails it is because its definition was faulty. Recovery blocks work if the compensation is possible, but do not apply in our Pan Am bankruptcy example.

Instead, we propose that compensating steps be specified as prioritized sets of guarded code blocks. The top priority code block exactly semantically undoes the original step, provided no intervening transactions have changed the values of any data items that it reads. Lower-priority code blocks could take alternative, automated steps if certain conditions hold true. These blocks may not semantically undo the transaction at all, but instead take other options to back out of the original work as much as possible. For example, one compensating code block for buying the ticket could specify that if the airline was bankrupt, a letter should be written to claim the ticket money. If no compensating code blocks succeed, the default could be to allow the user to fix the problem himself. This ability we call "sloppy compensation".

5.3.2 Dealing with Flexible Transactions

Flexible transaction models such as [GGK+90, ELLR90, WR91, Nod93] allow the execution flow of an open nested transaction to depend in part on the information in the database. For example, a flexible transaction may allow the same goal to be accomplished in different ways, such as getting a United flight or a TWA flight (with United preferred). The differences could be fairly significant, such as allowing a person to rent a car and driving instead of flying and then renting a car at the destination for local transportation. Another way an execution can be made flexible is to allow non-vital steps, such as renting the car at the destination, to be optional. If the car is not available, do not make a reservation.

[1] Ultimately, compensation may succeed once the bankruptcy proceedings have completed. However, this process is not very timely with respect to getting the money back to use on a different flight for the trip.

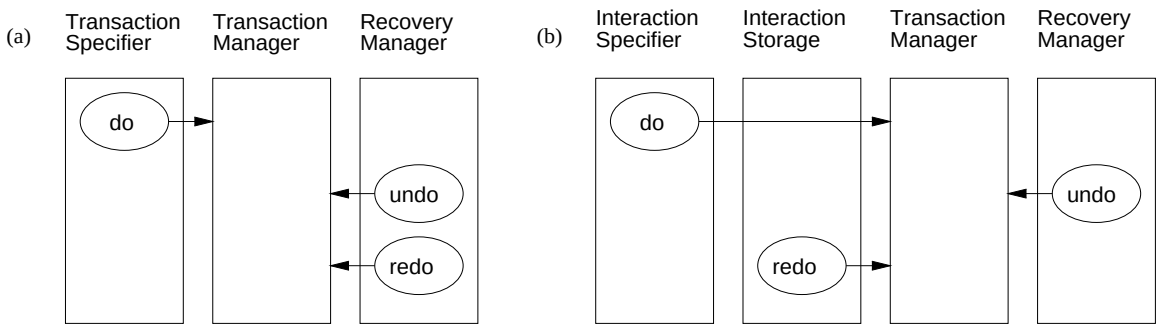


Figure 5.1: (a) Distribution of functions in traditional transaction and recovery management. (b) Distribution of functions when managing the execution and recovery of Interactions.

With flexible transaction models, the multidatabase still can determine compensating steps for the ones executed, and thus automatically execute compensating subtransactions as needed. However, if the original transaction failed due to some real problem, then the re-execution path is likely to be different from the original execution. For example, if a traveler's flight is canceled and no other flights are available, the logical re-execution would take the next priority step to try, which is renting a car and driving. Unfortunately, these new alternatives are untried, and therefore cannot have been logged in the same way that the compensating transactions are logged.

Although trying perhaps radically different alternatives to find a good plan is not a bad idea initially, it is important to be careful when evolving an already existing plan. This is because the plan leaves the traveler with certain expectations; it makes certain promises. While some divergence from the original plan may be necessary because of unavoidable conflicts, unnecessary divergence can be annoying and counterproductive. Thus, the goal of reactivity is not just to bring the Interaction back to a consistent state, but also to ensure that the new execution does not diverge unnecessarily from the original execution.

Thus, with Interaction reactivity we really need a different architecture than is used by a normal recovery system. In traditional compensation-based recovery, the re-execution of the transaction is assumed to be identical to the original execution. Therefore, the recovery manager can control the re-execution itself. This situation is shown in Figure 6.1(a). With reactivity, the re-execution of a particular subtransaction might want to attempt to reproduce the original execution. If that does not work, the re-execution path must be chosen according to the different alternatives specified in the Interaction, using the priorities originally defined for that Interaction. However, only the alternative that was actually executed originally is known to the recovery manager. The control of the re-execution must be turned over to the Interaction storage and management module, which must save *all* of the alternatives originally specified. Along with each stored alternative it notes whether it originally failed, originally succeeded, or was specified but not considered. During the re-execution, these possibilities are examined to determine what should be done this time, given that the underlying state of the database has changed since the original execution occurred.

Since the Interaction storage and management module has a record of the original specification, including all of the alternatives, it can choose an alternative that differs from the original one where needed. This feature is an added bonus from having a flexible transaction model in the first place. Also, since the Interaction storage and management module has a record of what originally succeeded, it can make more intelligent decisions about what *not* to change from the original execution. This is especially useful when trying to preserve parts of a plan which were only indirectly affected by the weak conflict. If the new plan does not in fact change the things that the rest of the plan depended on, then those parts of the original plan may still be salvageable.

5.3.3 Managing Concurrency and State During Recovery

Allowing concurrent subtransactions to run within a single Interaction, and allowing Interactions to maintain internal state both create new problems in determining what to compensate at a given time, and in what

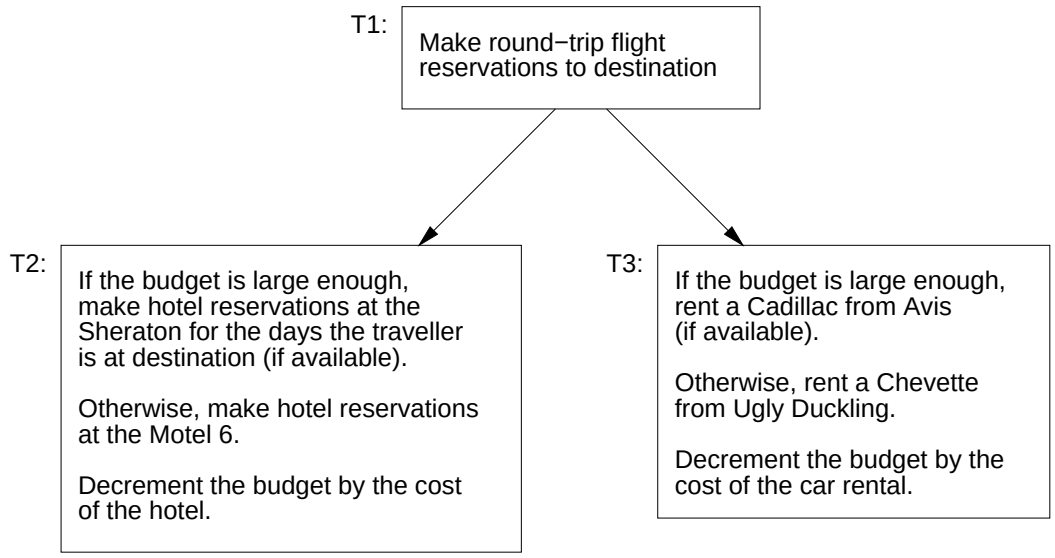


Figure 5.2: Does the traveler get a cheap hotel and an expensive car, or an expensive car and a cheap hotel?

order. Concurrency within an Interaction is specified explicitly using the *fork* and *join* constructs. Once a concurrent thread has forked, it can execute atomic subtransactions concurrently with the thread that forked it.

One issue with concurrent threads is that it may be necessary to semantically undo one thread while leaving the other concurrent threads intact. A second issue is that, when two concurrent threads both need to be semantically undone, it is usually possible to run the compensating subtransactions concurrently as well. In fact, more concurrency is possible during the compensation process, because compensation steps are usually fixed code (no branches or loops) with predetermined inputs (based on the inputs and/or results from the original step) [WR91]. However, depending on how the state is managed, two concurrent subtransactions that conflict on some state variable may not be able to be compensated for and re-executed concurrently. Consider the Interaction shown in Figure 6.2. In the initial execution, the traveler books rooms at the Sheraton, and gets a Chevette from Ugly Duckling. If the re-execution order is not consistent with the order the two subtransactions were originally executed, the traveler's new plans could be to book rooms at the Motel 6, and to rent a Cadillac from Hertz.

The recovery manager for the Interaction must maintain enough information about its structure to be able to determine exactly what compensating and re-execution subtransactions to issue and in what order during a specific Interaction recovery process. The information that should be maintained includes the partial execution order of the initial execution of the subtransactions, as well as the partial order in which conflicting accesses to state variables were resolved.

5.3.4 Eliminating Unnecessary Work

During compensation work may unnecessarily be undone and redone, causing various undesirable scenarios. Consider again the Interaction from Figure 6.2. If the plane reservations must be semantically undone, say, because a flight was canceled, it is likely that the traveler would want the new reservations made for the same travel days. If this can be done, then why should the hotel and car reservations need to be compensated for and redone anyway, given that nothing about the plane flights that is relevant to them has changed?

Doing the unnecessary compensation and re-execution can also cause other annoying problems as well. Consider a simpler case where a traveler books a hotel reservation at the Sheraton based on the dates of his plane reservation. If the plane reservation needs to be changed, say, to different flights on the same day, the hotel reservation will be automatically compensated for as well. When the hotel reservation is remade, the poor traveler may find that his Sheraton reservation was given to someone else in the interim, and he is now

stuck in the Motel 6.

This problem can be solved using a combination of two basic strategies. The first is to attempt to *replace* invalid subtransactions within the original execution in the order that they were originally executed. Thus, if the plane flight is canceled, the first attempted step should be to try to (atomically) replace the old flight with the new one using a *replacement transaction*. The replacement transaction first semantically undoes what the original transaction did, then tries to make a new reservation.

The second strategy is to attempt to *eliminate* these replacement subtransactions when the resulting execution would basically be a no-op. For example, if the replacement transaction would atomically remove the Sheraton reservation then get it again, those steps are eliminated during the recovery process.

The next sections define in detail the replacement algorithm for Interactions. We define what it means for a history to be *correct* when replacement has occurred in some Interaction. We give algorithms for determining when replacement can occur, and when replacement subtransactions can be eliminated. We also discuss the advantages and disadvantages of using the replacement approach for making Interactions reactive instead of using just compensation.

5.4 Replacement Overview

The compensation-based approach to reactivity is sufficient to bring an Interaction back to a state where it is consistent with the current state of the multidatabase and also internally consistent. As we saw in the previous section, however, it fails because it does not attempt to keep as much of the existing work unchanged as possible. Unlike compensation, replacement tries to replace any work in an Interaction that has become invalid without disturbing unrelated work. If we consider an Interaction to be a partial order of subtransactions, then we do not want to disturb unaffected work *even if it follows some invalid subtransaction in the partial order*.

Replacement is based on the fact that the compensation-based approach does correctly evolve an invalid Interaction into a new consistent state. Thus, we define our replacement algorithm based on this compensation strategy. Replacement considers a similar set of steps as would be executed using compensation, if the compensation were to be done in isolation and if it were to attempt to maintain the status quo wherever possible. However, replacement tries to eliminate some of the problems by working in a different order and by partitioning the steps into different atomic units.

If we examine the flow of compensation in the partial order of subtransactions that defines an Interaction, we see that like recovery, reactivity has a compensation phase where the subtransactions are semantically undone in the *inverse* of the partial order. This is followed by a re-execution phase, which is done in the original partial order. Replacement is always done in the original partial order. Starting with the invalid subtransaction, replacement works *forward* in the order, replacing each function as needed. Each replacement function is executed as a single atomic global transaction, with a set of compensating steps followed by a set of redo steps. The entire transaction is called the *replacement transaction*, and the portion of the replacement transaction executed on a single local database is called a *replacement subtransaction*.

In the plane and hotel example from section 6.3.4, the compensation-based approach would compensate for the hotel, compensate for the plane, then re-execute some plane reservation subtransaction followed by some car subtransactions to get the car reservation and the hotel reservation back. Replacement would compensate for the old plane reservation and make a new one (as a single replacement transaction). Then if needed, it would compensate for the hotel reservation and make a new one (as a second replacement transaction), etc.

5.5 Replacement Correctness

We define a replacement history as correct if, when done in total isolation, it leaves the database in exactly the same state when it completes as a compensation-based recovery would leave.

5.5.1 View Commutation in a Projected History

In order to determine if a replacement history is correct, we first define equivalence classes on histories, and determine tests to find out if two histories belong to the same equivalence class. In this section, we explore what we call *view commutation*, which is the process of rearranging a (projected) history into a history that is view equivalent to some other history. This rearrangement happens during replacement, and affects the order in which the replacement transactions are scheduled. It also helps us to determine when certain replacement operations need not be executed. We call the process defined here *view commutation* because we allow two (projected) transactions to commute if the commutation does not affect their projected view of the database or of their Interaction's internal state.

We give the following definitions used in the rest of this section:

Definition 5.5.1 *For a given subtransaction T, let*

- *T^C be the sequence of steps that semantically compensates for the effects of T on the database and on the internal variables of T's containing Interaction.*
- *T^R be the sequence of steps that would re-execute the work done by T.*
- *T' be the sequence of steps in T^C concatenated with the sequence of steps in T^R, executed atomically (the* replacement *transaction).*

Definition 5.5.2 *For any sequence of steps S, let*

- *$R_V(S)$ be the set of internal Interaction variables read by S.*
- *$R_D(S)$ be the set of database objects read by S.*
- *$R(S) = R_V(S) \cup R_D(S)$ be the* read-set *of S.*
- *$W_V(S)$ be the set of internal Interaction variables written by S.*
- *$W_D(S)$ be the set of database objects written by S.*
- *$W(S) = W_V(S) \cup W_D(S)$ be the* write-set *of S.*

We base view commutativity on the following fact:

Fact 5.5.1 *Code is deterministic. If a piece of code is executed twice, and if it reads the same variables and values the second time that it read the first, it writes the same variables and values the second time that it did the first time.*

The following theorem defines the conditions under which a projected subtransaction T_B can commute backwards[2] over some projected subtransaction T_A. View commutation addresses the question, "If T_B were to execute before T_A, would T_A do the same thing it would have done had it executed first?" Note here that we are not looking at transactions that have already been executed, rather we are looking at a schedule for future execution.

Theorem 5.5.1 (View Commutation Theorem) *Given subtransactions T_A and T_B, where T_B directly follows T_A in the projected history of the multidatabase, T_B can commute backwards over T_A if and only if the following conditions hold:*

1. *$W(T_A) \cap R(T_B)$ is empty.*
2. *For each variable or database object in $R(T_A) \cap W(T_B)$, T_B writes the same value that T_A read.*
3. *If there are any transactions that follow both T_A and T_B in the projected history, then for each variable or database object in $W(T_A) \cap W(T_B)$, T_A writes the same value as T_B.*

[2] Commuting backwards in this circumstance means, neither T_A nor T_B have been executed yet. T_A logically comes first, but we move the actual execution of T_B logically backwards over the (not yet executed) T_A, so that T_B actually executes before T_A. Commuting backwards succeeds of this does not affect the ultimate outcome of T_A or T_B.

Proof: If $W(T_A) \cap R(T_B)$ is empty, then the execution of T_B does not depend on the execution of T_A. Therefore, commuting T_B backwards over T_A will not affect the outcome of T_B (from Fact 6.5.1). If the intersection were nonempty, then we cannot commute because we do not know how the execution of T_A affects the underlying database state, except that it could impact the execution of T_B.

If $R(T_A) \cap W(T_B)$ is nonempty, then after the commutation T_A will read from (and therefore depend on) T_B. The outcome of T_A will be affected only if T_B writes a different value for at least one of these data items, causing T_A to read a different value for that item. However, as long as the values read by T_A remain the same, its outcome will not be affected, by Fact 6.5.1. If $R(T_A) \cap W(T_B)$ is empty, then the execution of T_A cannot be affected by the execution of T_B.

If there are transactions that follow T_A and T_B in the history, then if the two subtransactions do write different values any transactions following them will not read the same values. As long as they both write the same values, any transactions following them in the projected history will read the same values, and therefore have the same outcome, by Fact 6.5.1.

Since view commutation of T_B over T_A when the above conditions are defined affects the outcome of none of the transactions in the projected history, then the commuted history is equivalent to the original one. □.

5.5.2 Elimination of Work

In addition to commuting adjacent subtransactions in projected replacement histories, we also need to be able to determine when a subtransaction should not change the state of the database. *Elimination* during replacement is the process of determining which of the subtransactions that are considered are in fact still consistent, regardless of the fact that earlier subtransactions of the partial order have been replaced. That is, if a subtransaction is still consistent, it does not need to be replaced, so we say the replacement transaction execution can be *eliminated*. For example, again using the plane and hotel example from Section 6.3.4, changing the plane reservation may not affect the hotel reservation at all. Therefore, the subtransaction(s) to compensate for the hotel and remake the hotel reservation need not be executed at all.

In this and subsequent sections, we use the term "semantic undo" frequently. Earlier we mentioned the concept of "sloppy undo", which is also a semantic notion of undo. In our terminology, a "sloppy undo" really doesn't reverse the effects of the subtransaction it undoes from the point of view of the data. For example, a sloppy undo of a payment for a plane reservation might write a letter to claim refund money from a bankrupt airline.

"Semantic undo" in these sections really does try to reverse the effects of the original subtransaction from the point of view of the data. A semantic undo of a payment would get a refund. If the values read by the semantic undo are those that were written by the original transaction, then executing the semantic undo should leave the relevant data in the state it was in before the original transaction was run. This is an "exact semantic undo".

In replacement, the steps involved in re-executing a subtransaction T immediately follow the steps involved in compensating for T. Thus, the replacement subtransaction $T' = T^C|T^R$, where the | means that the steps of T^R immediately follow those of T^C in the execution on the local database. During elimination, the question we ask is, "Can T^R do something semantically equivalent to what T did?". If so, then since T^C semantically undoes what T did, T' will just be a big semantic no-op, and can be eliminated.

The setting in which elimination works can be seen more clearly by following the diagrams in Figure 6.3. In this figure, we have an original transaction T, its compensating transaction *Tc*, and the replacement transaction *Tr*. In Figure 6.3(a), T reads input *I0* from the multidatabase, executes, and writes output *O0*. When the compensating transaction *Tc* is run, the state of the database has changed, so *Tc* reads the new database state *O1* and writes database state *I1*. Because we are assuming that this is a transaction where the compensation and replacement steps are done atomically, we know that *I1* is also the state read by *Tr*. The resulting state written by *Tr* is yet another state *O2*.

In Figure 6.3(b), we assume that *Tc* reads the same database state (at least for the subset of the database state that contains the data items it accesses) that was in effect at the time T committed. Thus, we have *O1* = *O0*. The compensating transaction *Tc* is then an exact undo, so we know that the state that it writes should be identical to the state that T read, so *I1* = *I0*. Thus, the replacement transaction *Tr* reads the same database state read by T. If it executes the same code, then the database state at the time it commits is the same as the state was at the time T committed, so *Tc* and *Tr* do not have to be run at all.

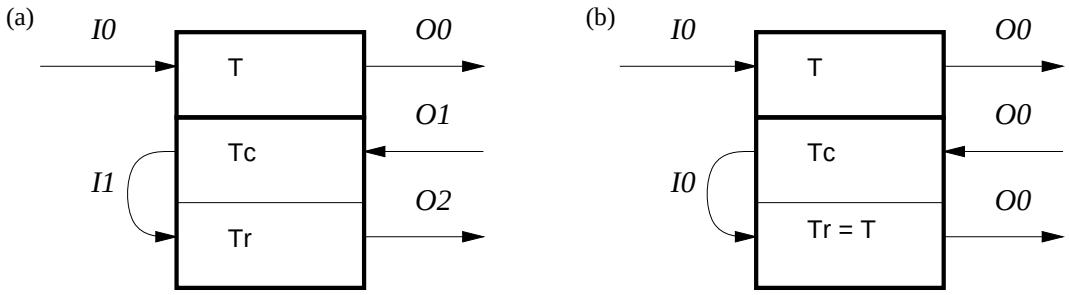


Figure 5.3: Elimination. (a) Normal situation where elimination does not work. (b) Situation where elimination works.

We base our test for elimination on the following theorem:

Theorem 5.5.2 (Elimination Theorem) *If*

1. *for each variable or database object in* $R(T') - W(T^C)$, T' *expects to read the same values that* T *read, and*
2. *for each variable or database object in* $R(T^C) \cap W(T)$, T^C *expects to read the same value that* T *wrote, and*
3. *for each variable or database object in* $R(T^R) \cap W(T^C)$, T^C *expects to write the same values that* $T(= T^R)$ *read,*

then T' *can be eliminated without affecting the consistency of the Interaction.*

Proof: First consider the execution of the steps of T^C. If T^C is exactly compensating T, then T^C should have the same write-set that T had. Similarly, the read-set of T^C is a subset of the write-set of T. The semantic undo will exactly undo T if T^C reads the same database state that was left after T (for the relevant portion of the database). This is certainly true if the following conditions hold:

1. For each variable or database object in $R(T^C) - W(T)$, T^C expects to read the same values that T read.
2. For each variable or database object in $R(T^C) \cap W(T)$, T^C expects to read the same value that T wrote.

Now consider the execution of the steps of T^R. If T^R reads the same variables and values as T, it should do exactly the same thing T did, by Fact 6.5.1. This would occur if the following conditions hold:

1. For each variable or database object in $R(T) - W(T^C)$, T^R expects to read the same values that T read.
2. For each variable or database object in $R(T) \cap W(T^C)$, T^C expects to write the same values that T read.

If T^C exactly undoes T (which happens if the first set of conditions is true), and T^R exactly redoes T (which happens if the second set of conditions is true), then $T^C|T^R$ should combine semantically to have no effect on the underlying data. We also know that, if exact compensation occurs, T^C will reverse the effects of T on the database; therefore, the two transactions write the same data items ($W(T) = W(T^C)$). These conditions, combined, give the conditions of the theorem. □

Note that elimination favors the situation where the semantic redo does the same thing as the original transaction, even if other alternative higher-priority re-execution paths are valid.

5.5.3 Correctness

An Interaction is defined as a *partial order* of subtransactions. This partial order defines the dependencies among the different subtransactions. When a subtransaction T is explicitly invalidated because of some external circumstance, we define the *invalid portion* of the Interaction as the part of the partial order that includes T and all of its descendants. We call all of the subtransactions in the invalid portion *invalid subtransactions.*

Definition 5.5.3 (Compensation-Correct Execution) *A complete execution of an Interaction is* compensation-correct *if the following conditions hold:*

1. *Each invalid subtransaction has a compensating subtransaction in the execution. The compensating subtransaction is ordered after its corresponding original subtransaction. If the invalid subtransaction is read-only, the compensating subtransaction may be empty.*
2. *If two invalid subtransactions were ordered in the original execution, then the corresponding compensating subtransactions are ordered in the inverse order.*
3. *Each new subtransaction follows the previous valid committed subtransaction in the Interaction definition's partial order.*
4. *If the Interaction was aborted, the execution contains no valid subtransaction, only invalid subtransactions and compensating subtransactions.*

Definition 5.5.4 *A correct compensation-based history contains a set of compensation-correct Interaction executions, arbitrarily interleaved at the subtransaction level.*

Now we can look at correctness with respect to alternative recovery approaches.

Definition 5.5.5 (View Equivalence) *Two executions of a given Interaction are* view equivalent *if, when executed in isolation, one can be made into the other using only correct view commutation and subtransaction elimination operations, as defined in Theorems 6.5.1 and 6.5.2.*

Definition 5.5.6 (Correctness) *A correct replacement-based history contains an arbitrary interleaving at the subtransaction level of a set of Interaction executions, each of which is view equivalent to a compensation-correct execution.*

5.6 The Replacement Algorithm

The replacement algorithm attempts to replace as much of the work done by the invalid portion of an Interaction execution as is possible, while defaulting to a compensation-based strategy when replacement is not possible. In the worst case, a schedule for replacement is identical to the compensation schedule. In the best case, the schedule for replacement atomically replaces only the invalid subtransaction and those subtransactions in the invalid portion whose view of the database differs from the view that the original subtransaction read.

In the replacement algorithm we make use of two rules, based on our earlier observations concerning view equivalence and elimination. The first is that a subtransaction can be replaced if the following two conditions hold for it:

1. All of its ancestors in the invalid portion have been replaced or their replacement subtransaction has been eliminated.
2. The replacement subtransaction view-commutes over all of the (yet to be executed) compensating subtransactions for its descendants in the invalid portion.

The second rule states that, if a given subtransaction can be replaced, then the replacement can be eliminated if it is semantically has no effect. With elimination, the replacement subtransaction effectively only reads the values of some objects in the database, so when we test for commutativity we only need to commute those read operations.

We assume that when the replacement algorithm is invoked, all subtransactions in the invalid portion have committed. If there are uncommitted subtransactions in the invalid portion, we can simply abort them. However, we need to ensure that once the subtransactions are aborted, there is still a path in the invalid portion to all descendants of the invalid subtransaction. This is ensured because we restrict the commit order of the subtransactions to be consistent with the partial order of the Interaction; thus, no uncommitted subtransaction can have a committed subtransaction as its descendant in the partial order.

We assume the presence of a log. The log contains the read- and write- sets (and their values) for each subtransaction. Also, we log projected read-and write-sets for the compensating subtransaction. We can project these read-and write- sets *during the execution of the original subtransaction* as follows:

1. Since the compensating subtransaction compensates for changes made by the original subtransaction, we assume that the write-set for the original transaction is the same as the write-set for the compensating transaction.

2. The compensating subtransaction should not have to read any data item that it does not write, we assume that the read-set for the compensating transaction is identical to the write-set.

3. If the original subtransaction sets a value of a data item relative to its read value, then its read- and write-values are logged as *unknown*. Otherwise, the read-value is logged as the value the original subtransaction wrote, and the write-value is logged as the value the original subtransaction read.

Note here that we log the data items and their values that should be read and written if the semantic undo is exact – that is, it has the same effect on the local database that a state-based undo would have. This is because these data items and values are logged for the purpose of deciding whether a global replacement transaction can be *eliminated*, and elimination is not possible unless all compensating transactions actually can do an exact semantic undo. Therefore, these logged values suffice to make the decision. One other advantage of this approach to logging is that if the original transaction does a large query, the compensating transaction record in the log does not replicate the large read-set.

Algorithm 5.6.1 (Test Elimination) *Input: Read- and write-sets (and values) from T's execution, read from the log. Projected read- and write-sets (and values) for T^C, read from the log.*
Output: The subtransaction ID if elimination succeeds, else NULL.

1. *Begin a subtransaction.*

2. *For each item in T's read-set that is not in the projected write-set for T^C, read its current value and compare it to the saved read-value for T. If the value is different, abort the subtransaction and return NULL.*

3. *For each item in both T's write-set and T^C's read-set, compare the saved write-value for T with the projected read-value for T^C. If the value is different, abort the subtransaction and return NULL.*

4. *For each item in both T^C's write-set and T's read-set (where T^R is projected to be identical to T) compare the projected write-value for T^C with the projected read-value for T^R (which is the read-value for T). If the value is different, abort the subtransaction and return NULL.*

5. *Return the TID of the subtransaction.*

Note that the algorithm to test for elimination assumes that T^R will be identical to the original subtransaction T. This is certainly true if elimination succeeds. If the read values diverge, elimination certainly will not succeed. Therefore, this assumption is valid.

The algorithm for view commutation of two transactions is as follows:

Algorithm 5.6.2 (View Commutation) *Input: P, the invalid subtransaction and its descendants. The read- and write-sets for T'.*

Output: R, the set of compensating subtransactions T failed to commute over.

1. *Let T be the subtransaction that was originally invalidated. Remove T from P. Initialize R to ϕ*
2. *From a log, get the read- and write- sets for the compensating subtransactions for each subtransaction in P.*
3. *Given the input read- and write- sets for T', the replacement of T, do the following until P is empty:*
   (a) *Let C be the compensating transaction for some transaction B in P that has no ancestors in P's partial order.*
   (b) *If the intersection of T''s read-set and C's write-set is nonempty, place B and all of its descendants in P into R and remove them from P.*
   (c) *If the intersection of T''s write-set and C's read-set is nonempty, check for each item that T' writes the same value C expects to read. If this is not true for some item, place B and all of its descendants in P into R and remove them from P.*
   (d) *If P has more than just C in it, then if the intersection of T''s write-set and C's write-set is nonempty, check that T' and C will write the same value. If this is not true for some item, place B and all of its descendants in P into R and remove them from P.*
   (e) *Remove C from P.*
4. *Return R. If R is empty, the commute was successful.*

Algorithm 5.6.3 (Replacement) *Input: P, the invalid portion.*

1. *Choose some invalid subtransaction T whose ancestors in the invalid portion have all been replaced.*
2. *Test if the replacement for T can be eliminated using Algorithm 6.6.1. If so, do the following:*
   (a) *Using Algorithm 6.6.2, view commute T' over the compensating subtransactions for all of T's descendants in the partial order, with* read-set$(T') =$ read-set$(T) \cup$ read-set(T^C) *and the read-values set to the earliest values that would be read by T'. The write-set of T' is empty.*
   (b) *If the view commute succeeds, abort the transaction begun by the elimination algorithm and remove T from the invalid portion.*
   (c) *If the view commute fails to commute T over some compensating subtransaction C, call the compensation algorithm defined below, with the returned set S equal to the set R returned by the view commute algorithm.*
3. *If elimination fails, do the following:*
   (a) *Continuing with the transaction begun by the elimination algorithm, execute T^C but do not commit it. Pass the transaction control to the Interaction storage and management module.*
   (b) *The Interaction storage and management module executes a new T^R as a part of the same transaction, and passes control back. Now a new T' has been executed but not committed.*
   (c) *Get T''s read- and write-sets from the current execution.*
   (d) *Using Algorithm 6.6.2, view commute T' over all the compensating subtransactions for all of T's descendants in the partial order.*
   (e) *If the view commute succeeds, commit T' and remove T from the invalid portion.*
   (f) *If the T' does not commute over some compensating subtransaction C, abort T'. Call the compensation algorithm defined below, with the set S equal to the set R returned by the view commute algorithm.*

4. *If there are any subtransactions remaining in the invalid portion, go back to step 1 and try again. Otherwise, allow the Interaction to continue its execution from the current state.*

Algorithm 5.6.4 (Compensation) *Input: S, the set of subtransactions that can't currently be replaced.*

1. *Choose some subtransaction $U \in S$ that has no descendants.*
2. *Execute and commit the compensating subtransaction for U.*
3. *Remove U from the invalid portion and from S.*
4. *If S is empty, return. Otherwise, go back to step 1.*

Note that the replacement algorithm falls into compensation when replacement fails. Compensation (including, where necessary, sloppy compensation) always succeeds in bringing the Interaction back into a consistent state. Therefore, replacement should always bring the Interaction back into a consistent state as well.

5.7 Replacement Examples

This section contains three examples of situations where replacement is invoked, to demonstrate how useful it is in different situations.

5.7.1 Flight Cancellation Example

Let us examine how the situation described in Section 1.3.3 would manifest itself in the multidatabase, and how it would be handled. In this example, assume that the flights, car rental, and hotel have been booked, but the traveler has not left yet. Furthermore, there are weak conflicts on the existence of the plane, hotel and car reservations in effect.

At this point, assume that the airline cancels the outgoing flight. Then the following steps occur:

1. The airline invokes an independent transaction on its own database to cancel the flight and delete all of the reservations on that flight.
2. The Activator on the airline's local database receives a copy of the transaction message that deletes the reservation, detects that the update may be of interest, and generates a query to check the state of the information in the database concerning the flight reservation.
3. The local database returns the query response to the Activator, confirming the reservation has been canceled. (Note that the supplementary query is necessary, as the transaction that canceled the reservation could have been aborted.)
4. The Activator notifies the Interaction storage and management module of the violation of the weak conflict.
5. The Interaction storage and management module responds by telling the Recovery Manager to roll back the Interaction through the plane reservation and re-execute it.
6. The Recovery Manager then semantically undoes the plane reservation global transaction (but doesn't commit it). Then, as a part of the same transaction, the Interaction storage and management module books a new flight and gets new flight information. In this example, we assume that the new flight occurs on the same date as the canceled flight. During this process, the Activator is also told to remove the old event from the event table and to place a new event on the existence of the new flight into the event table.
    - In the old flight's airline database, the replacement transaction writes the old reservation information and the passenger information.

- In the new flight's airline database, the replacement transaction writes the new reservation information and the passenger information.
- In the Interaction's local variables, the replacement transaction writes the flight number, the date, and the time. Note that the date did not change.

7. The Recovery Manager tries to view commute the replacement transaction for the flight to the front of the projected history, over the compensating transactions for the car and hotel reservations. This succeeds for both. For example, the compensating car rental transaction reads only the the rental company name and the car reservation number from the Interaction's internal variables. The only writes it does are to delete the car reservation from the rental company's database and delete the record of the car reservation from the itinerary.

8. The Recovery Manager then commits the replacement transaction.

9. The projected information to be read and written by the car and hotel replacement transactions does not change the state of the multidatabase. For example, the car rental replacement transaction reads and writes the following information from the multidatabase:

   - It reads the car reservation information from the car rental database.
   - It reads the car reservation number, the date the plane arrives, and the date the return flight leaves from the Interaction's internal variables.
   - It writes the new (identical) car reservation information to the car rental database. We know it is identical because the car reservation information read and the information read from the Interaction's variables is identical to what was written by the original car rental transaction.
   - It writes the car reservation number into the Interaction's internal variables.

   this means that the replacement transactions for the rental car and the hotel can be eliminated from the recovery execution.

This entire process is shown in Figure 6.4.

Eventually, the customer takes the outgoing flight. At this point, the Interaction execution continues as follows:

1. The airline's database is updated by some independent transaction to indicate that the flight landed.

2. The Activator on the airline's database sees the copy of the update, and generates a query to check the flight state.

3. The query returns indicating the plane has indeed landed, and the Activator sends a notice of this to the Interaction storage and management module. The Activator then removes this event from the event table.

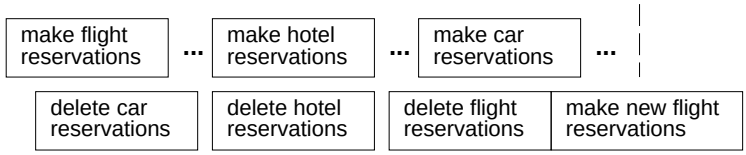
4. The Interaction storage and management module receives the notice, and continues executing the appropriate Interaction thread. This causes the travel agent's database to be updated.

5. The Interaction storage and management module cancels the weak conflict on the outgoing flight reservation, because the specified execution span of the weak conflict ended with the completion of the plane flight.

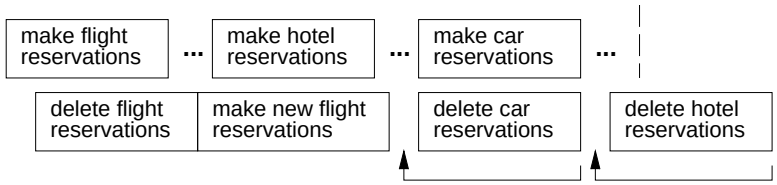
5.7.2 Ferry Example

An example where the replacement plan diverges quite a lot from the original plan is shown in Figure 6.5. Here we have a traveler who is trying to use his United frequent flyer miles to get a free rental car. However, his trip includes a visit to an island, and if he has the car he needs to make a ferry reservation for the car. Otherwise, he can just get on the ferry with no reservation. Now, let us assume that the United flight is canceled, and the best flight to replace it is on another airline. Thus, the traveler cannot get the free rental car, and does not have to make the ferry reservation any more.

(a) Project the beginning of the execution with compensation order.



(b) Commute the flight reservation changes to the beginning of the projected execution (if possible).



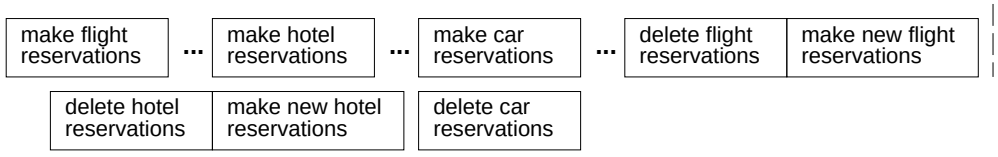
(c) Try to eliminate flight reservation changes.

This does not work because the original flight reservations were cancelled.

(d) Execute the flight reservation changes.

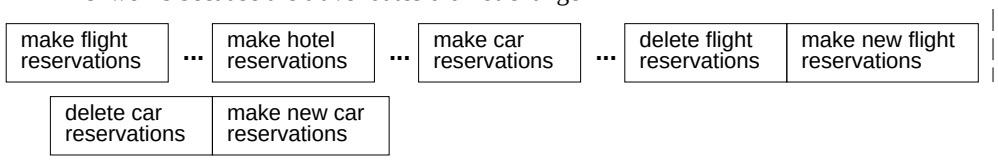
Assume that new reservations can be made for the same travel dates.

(e) Commute the hotel reservation changes to the beginning of the projected execution (if possible).



(f) Try to eliminate hotel reservation changes.

This works because the travel dates did not change



et cetera

Figure 5.4: The replacement process for the plane, hotel, and car example. In each stage, the end of the current history is delimited by a dashed vertical bar, and the projected execution follows that bar. Boxes delimit atomic sequences of steps. If two boxes abut each other, the two sequences of steps they represent are concatenated together into a single atomic transaction.

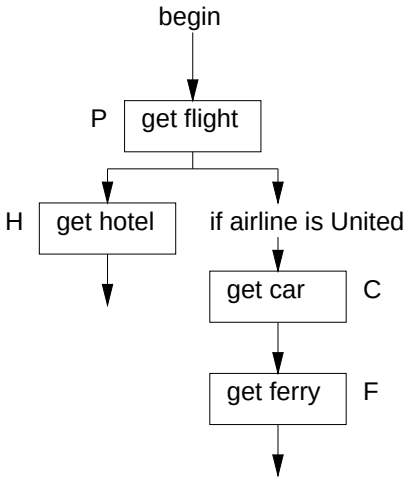


Figure 5.5: Example: Frequent flyer miles give traveler a free car rental, requiring ferry reservations to be made for the rental car.

The original execution dependencies are reflected in the arrows between the atomic blocks in the figure. The "get hotel" block is dependent on the airline flight dates. The "get car" block is dependent on both the flight dates and the airline. The "get ferry" block is dependent on the existence of the rental car reservation.

In this example, the initial history before the airline flight is canceled is shown in Figure 6.6(a). Figure 6.6(b) shows the projected compensation-based execution; the actual execution should be equivalent to this one.

At the point in the scheduling shown in Figure 6.6(c), the flight has been successfully replaced by another flight on a different airline, and replacing the rental car is considered. However, since not enough frequent flyer miles are available for the free rental car (because of the change of airline), the car reservation is just canceled. At this point the ferry reservation is not required, so no code to replace the ferry reservation is executed. Note also in this figure that, since the compensating block for getting the ferry reservation does not read any information about the rental car (unlike its initial counterpart), the atomic block dealing with the car does commute over it.

By the point shown in Figure 6.6(d), the ferry needs to be dealt with. The compensating transaction for the initial ferry reservation commutes over the hotel reservation, since the two have nothing to do with each other. Then, the replacement process notes that the hotel replacement transaction would not change anything, and thus it can be eliminated from the replacement history. The final history after replacement completes is shown in Figure 6.6(e).

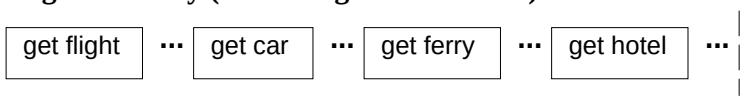
5.7.3 Budget Example

In the budget example from Section 1.3.2 a careful budget is being kept, and the trip must cost less than the budget. In this situation, replacement is constrained by data access. Even though the transaction to get the hotel and the transaction to get the car theoretically can execute concurrently, a sequence is forced on them because they both have to access the budget internal variable.

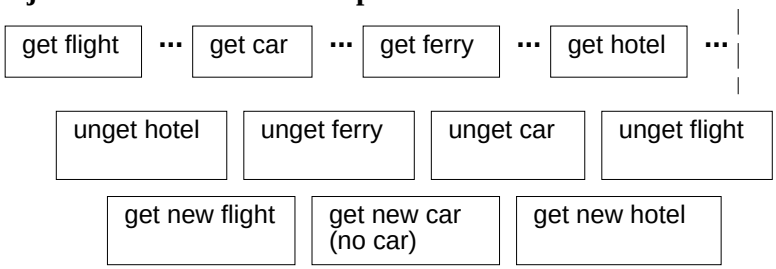
In this example, replacement can do nothing to optimize what needs to be done to bring the Interaction back to a consistent state. This is because every original transaction, every compensating transaction, and every re-executed transaction read and change the budget variable. Thus, each such transaction must read from the previous such transaction. Therefore, no transaction executed during replacement can commute over the previous transaction, and the replacement order deteriorates to be identical to the compensation order.

The only way that this example could be made more efficient is to take semantics into account when testing for commutation. This is a subject of future research.

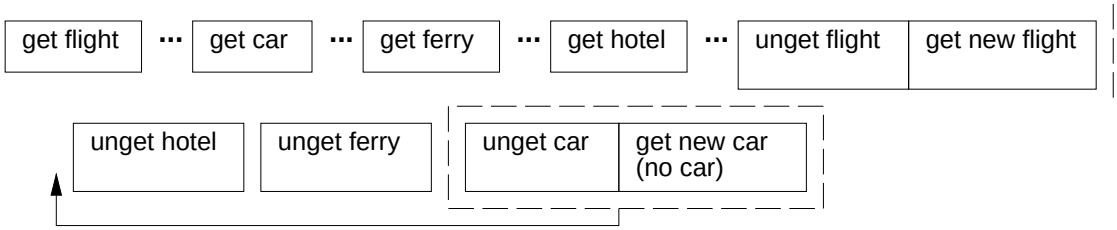
(a) Original history (before flight is cancelled).



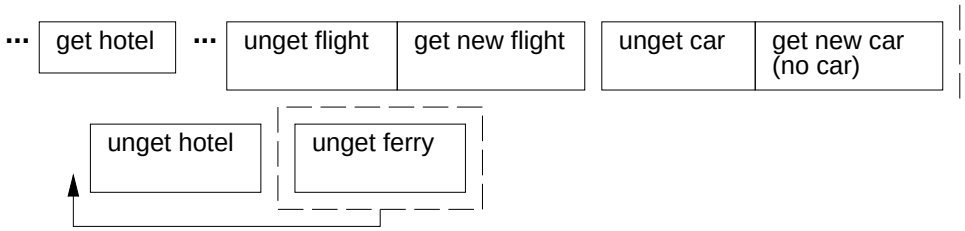
(b) Projected execution with compensation order.



(c) History and projected execution after new plane reservations are made (working on car).



(d) History and projected execution after no new car is gotten.



(e) Final history after recovery, assuming the hotel reservation stands.

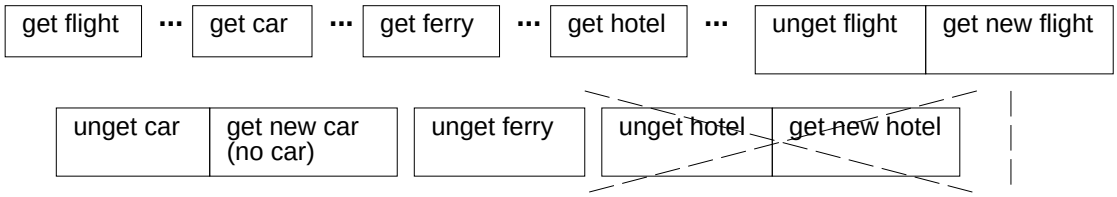


Figure 5.6: Replacement steps for ferry example.

5.8 Summary

In this section, we discussed the basic theory behind making Interactions reactive. We discussed how events are specified and detected in the multidatabase. We described a way to implement reactivity using compensation. This approach has several problems, but nonetheless serves as a basis for defining correctness of reactive Interaction executions.

The major problems with using compensation for reactivity have to do with the fact that valid reservations are lost when compensating transactions are run and committed, and that the replacement execution can diverge unacceptably from the original execution. We proposed a different algorithm for reactivity, called *replacement*. Replacement differs from the compensation scheme in that it processes invalid work in the order that it was originally executed, attempting to replace the invalid transaction and the transactions that depended on it originally as needed. It also attempts to undo and redo specific transactions atomically, so control over specific data items is not lost.

We defined correct reactive Interaction execution as the set of executions that are view equivalent to an execution that is correct and uses a compensation strategy. View equivalence means that the transactions in the Interaction execution can view commute the steps to form a compensation-correct execution, if executed in isolation from all other transactions on the multidatabase. If a set of compensating steps followed immediately by a set of redo steps combines to have no effect in the multidatabase, the execution of those steps can be eliminated from the reactive execution.

One other difference between the replacement approach to reactivity and the compensation approach is how the reactive part of the execution is divided up into global transactions. The compensation execution does the compensation for some original transaction as one global transaction, and the re-execution as a separate global transaction. The replacement approach does the compensation and re-execution as a single global transaction.

This chapter concludes our theoretical discussion of Interaction correctness. In the next chapter we discuss how to enforce global transaction atomicity in a multidatabase. This is a necessary prerequisite for Interaction correctness as we have defined in these last two chapters.

Chapter 6

Global Transaction Atomicity

From Theorem 4.1.1, recall that the two requirements that the multidatabase must fulfill to ensure global transaction atomicity are (1) it must serialize the global transactions in a consistent order on all local databases, and (2) it must enforce an atomic commit protocol. If we relax the global transaction to only require semantic atomicity, then only a semantically atomic commit protocol is required.

This chapter explores how these two requirements can be fulfilled in the multidatabase. We emphasize general principles rather than presenting new, specific algorithms. Chapter 2 describes many specific algorithms that can be used to support global transaction atomicity as required for Interaction correctness. In our work, we provide a framework for understanding, classifying, and building real multidatabase systems that require global transaction atomicity.

In the first section, we provide a general approach to maintaining a consistent serialization order on all local databases. This section parallels work done by Elmagarmid and Du [ED90]. In the second section we discuss atomic and semantically atomic global transaction commitment. We give a complete theory as to when and how it can be allowed. We also present two new semantically atomic commit algorithms.

In this chapter, we assume that each global transaction has at most one global subtransaction on a given local database. We also assume that the local databases ensure transaction atomicity. We assume (from the architecture described in Section 1.6) that there is a single layer at the global level of the multidatabase that coordinates global transaction execution. We call the module that executes in this layer the *Global Transaction Manager (GTM)*. We also assume there is an extension of the multidatabase associated with each local database, whose job is to coordinate the execution of each global subtransaction that runs on that local database. This piece we call the local database's *Agent*. The Agent also intercepts other (non-multidatabase) communication with the local database as needed to maintain correct operation in the multidatabase.

6.1 Global Serialization Order

Correct operation of the multidatabase requires that the subtransactions of the different global transactions be executed in a consistent order on all of the local databases. This can be done using one of two methods. One method is to have the Global Transaction Manager define a global serialization order, which is told to the Agents. Then the Agents are responsible for enforcing that order. We call this a *normal scheme*. Alternatively, the Agents could determine on the fly (using their knowledge of their local database's protocols) in what order the local database is serializing its subtransactions. They then each could inform the Global Transaction Manager of their local order, so it can detect inconsistencies and abort global transactions as needed. We call this a *certification scheme*. Examples of existing normal schemes and certification schemes were described in Chapter 2.

6.1.1 Normal Schemes

In a normal scheme, the Global Transaction Manager dictates a global serialization order to the Agents, and the Agents enforce that order on their respective local databases. The basic structure for an algorithm that follows a normal scheme is shown in Figure 5.1.

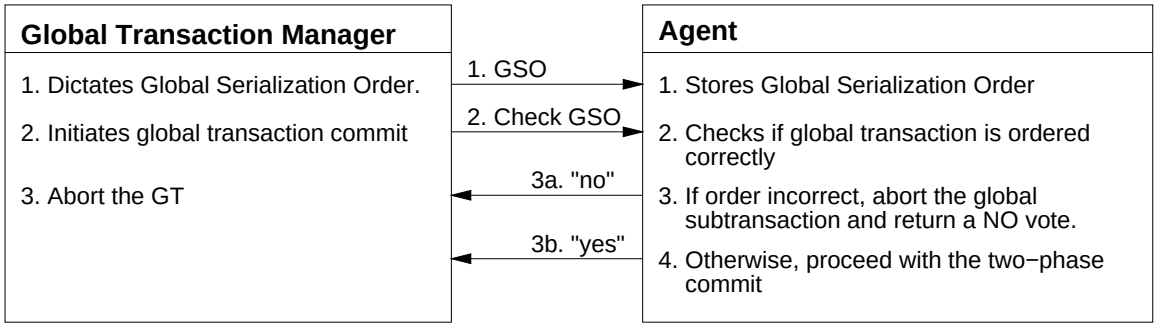


Figure 6.1: Normal algorithm for ensuring consistent global serialization order.

With a normal scheme, the Global Transaction Manager uses some consistent method to determine the global serialization order. For example, it could dictate that the order be the same as the order in which the global transactions begin. Alternatively, it could use the order in which the global transactions commit. Regardless, the order dictated by the global transaction managers is of necessity the one enforced between the global subtransactions on the local databases.

Since the multidatabase is heterogeneous, different data items in the multidatabase may be protected using different concurrency control schemes. Therefore, different types of Agents may coexist in the multidatabase, one for each concurrency control strategy used by some local database. However, since the purpose of the Agents is to enforce a consistent serialization order, and since this (along with atomic global transaction commit) is a sufficient requirement for a correct merged history, the diversity of concurrency control schemes should not impact the ultimate consistency of the global database.

The normal schemes defined here for enforcing consistent global serialization order in the local databases are similar to the one used by Elmagarmid and Du in [ED90]. The basic idea is to determine for each global subtransaction what its *serialization point* is. This is the point at which the transaction's position in the local database's serialization order is fixed by its concurrency control protocol. A discussion of the different serialization points for different local concurrency control schemes is discussed in Section 5.1.3.

In the remainder of this section, we examine our generic approach that works with local databases that use all types of concurrency control. Unlike forced local conflict [GRS91], this approach is less intrusive but very inefficient. We observe that the global serialization order can be enforced by not allowing any subsequent global subtransaction to begin until after the current one has reached its serialization point. The generic scheme delays beginning a subsequent transaction until after the current one would have reached its serialization point *in any local database concurrency control scheme*. Thus, we can trade off concurrency of global subtransactions for generality.

Unfortunately, the real problems occur when the local concurrency control schemes normally allow more concurrency, such as with serialization graph testing or timestamp ordering. The following theorem indicates that a general approach is impractical:

Theorem 6.1.1 *Given an infinite number of data items in a database, an execution of any arbitrary number of transactions can be constructed in which the serialization order is inverted from the execution order.*

Proof: We will show this by induction. (Base case) Suppose that we have an execution of three transactions T_a, T_b, and T_{ab}, where T_a commits before T_b begins. Let T_{ab} be a transaction whose execution overlaps the execution of both T_a and T_b. This situation is shown in Figure 5.2. In this figure, reads-from relationships between transactions are indicated by arrows. If the local database's concurrency control protocol allows T_{ab} to read from T_a, and similarly allows T_b to read from T_{ab} (for example, if it is based on SGT), then the serialization order would be T_b T_{ab} T_a, even though T_a completely precedes T_b in the execution order.

(Induction Step) Suppose we have a chain of transactions $T_a \ldots T_m$, all of whom ran serially with respect to each other, and have committed in the local database. Say their serialization order has been inverted by the execution of other transactions in the manner described in the base case. Now, say another transaction T_n begins, and there is also a transaction T_{mn} whose execution span overlaps those of T_m and T_n. Then if

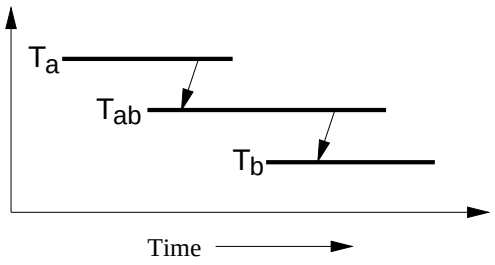


Figure 6.2: Execution order for a set of transactions where the serialization order differs from the execution order.

T_{mn} reads from T_m, and T_{mn} reads from T_n and *no other transactions in the execution*, then T_n will precede T_m in the serialization order. □

Because this arbitrary inversion can occur, the general method needs to be really restrictive. The approach is for the Agent to delay passing any command to begin a subsequent global subtransaction in the serialization order until the following conditions hold:

1. The previous global subtransaction has committed.
2. There has been a point in time in the execution on the local database after the previous global subtransaction commit where no transactions are running at all (a *lull*).

These conditions suffice because overlapping transactions are required for the local database to be able to serialize its transactions in the inverse of the order in which they are executed. To ensure that these lulls occur, once a global subtransaction has committed, the Agent may delay passing on any command that begins any new independent transaction or global subtransaction until all transactions that were concurrent with the global subtransaction have committed or aborted. Alternatively, we can avoid exerting control over the input stream to the local database, and just wait for the lull to occur.

Note that we can bypass the long wait or the enforcement of the lull in situations where we know that the next global subtransaction reads from the previous one, or has a sharing relationship as defined in [ZE93b]. In that case, we can schedule the two global subtransactions concurrently, and merely ensure that they commit in the global serialization order. However, this may be hard to do in practice without examining the internal structure of the global subtransactions.

6.1.2 Certification Schemes

Certification schemes differ from normal schemes in that they take an optimistic approach towards ensuring consistency in the global serialization order. Rather than dictating a global serialization order, certification schemes allow the local database to serialize global subtransactions without interference. However, they do require that inconsistently-ordered global subtransactions eventually be aborted. Thus, at global transaction commit time the multidatabase must either certify that the global transaction is ordered consistently on all local databases or it must abort the global transaction.

In a certification scheme, each Agent deduces the serialization order of the global subtransactions on its own database from its knowledge of the database's concurrency control strategy and from its observation of certain events in the execution of the global transactions themselves.

The way that an Agent deduces the serialization point of a transaction on the local database depends on the concurrency control method of the underlying database. The different serialization points are described in Section 5.1.3. We assume that the Agents can observe certain types of event in the global subtransaction execution, such as subtransaction begin and commit. Each local database's Agent observes the event that characterizes its underlying database's serialization point. When a subtransaction executes its serialization event, the Agent reports the global transaction ID of its associated global transaction back up to the Global Transaction Manager. The Global Transaction Manager gathers these orders from each Agent, checks for

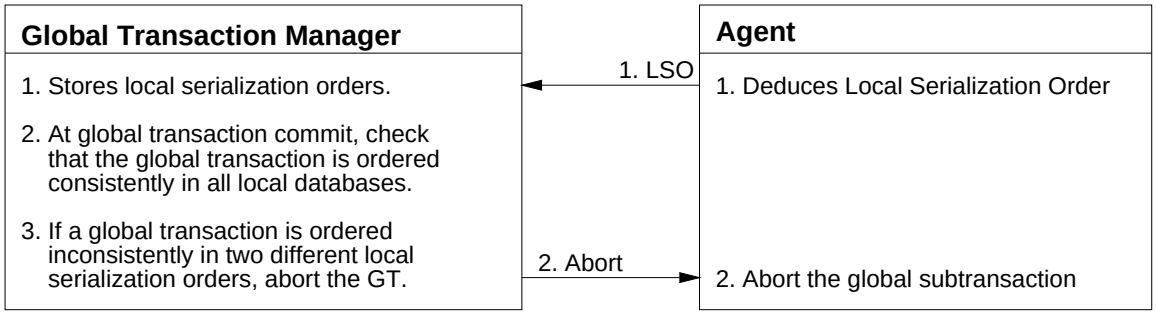


Figure 6.3: Certification algorithm for ensuring consistent global serialization order.

inconsistencies, and aborts global transactions where needed to ensure that a consistent order is generated on all local databases.

Certifiers traditionally check the validity of a transaction at its commit time. Because of the nature of certification in general, the Global Transaction Manager should also certify its global transactions at commit time. Most local databases have concurrency control schemes that determine their serialization order is determined at the latest at commit time. However, in the cases where this is not necessarily true, forced local conflicts [GRS91] can be used to constrain the serialization order of the global transactions before the global transaction commit is processed.

How inconsistencies in the different local serialization orders are dealt with depends on the global serialization order that is being enforced. Two logical orderings to choose for the GSO are the global transaction begin order or the global transaction commit order. In each case, the Global Transaction Manager decides that a particular global transaction T is ordered consistently at T's commit time. At T's commit time, if T's subtransaction is the first uncommitted transaction in each of the local databases that T executed on, then T is ordered consistently and can commit. If not, then transactions need to be aborted for the order to become consistent. If the global serialization order is consistent with the global transaction begin order, then all global subtransactions that precede T in any local serialization order (and their respective global transactions) must be aborted before T can commit. If the global serialization order is consistent with the global transaction commit order, then T itself must be aborted.

6.1.3 Serialization Points

In this section, we examine specific concurrency control strategies to see whether some approach that is tailorable to the individual local database works for any of them. This is motivated by the fact that, if the Agent knows something about its local database's concurrency control scheme, it may be able to deduce and/or manipulate the order in which the local database serializes the global subtransactions. This knowledge can be used in implementing either a normal scheme (to manipulate the local serialization order) or in a certification scheme (to deduce the local serialization order) to enforce a consistent global serialization order.

The different concurrency control strategies we examine here are described in [BHG87]. We first examine the rigorous 2PL, then timestamp ordering and serialization graph testing. We also examine strict variants (that do not always serialize transactions in their execution order) and basic variants (which are, in general, *not* recoverable), as well as relevant conservative variants (which predeclare their read and write sets).

6.1.3.1 Rigorous 2PL Databases

In rigorous 2PL, a transaction takes locks on the objects it accesses up to the commit point, and all locks are released during the commit itself. Each rigorous 2PL transaction reaches its serialization point at the end of its growing phase, when all of its lock requests have been granted. However, it is difficult in practice to observe when a transaction takes its last lock. The job is made simpler by observing that if two transactions overlap in the time that they both have been granted all of their locks, their serialization order must be indeterminate. Therefore, any point between the time the last lock is granted up to and including the time

the first lock is released can be chosen to be the serialization point. In rigorous 2PL, we choose the commit point, because all locks are released during commit and it is easy to observe. Thus, rigorous 2PL agents allow multiple global transactions to run concurrent subtransactions in the same local database as long as the subtransaction commits are passed to the local database in the global serialization order.

Note that if all of the local databases in the multidatabase use rigorous 2PL as their concurrency control strategy, and the global transactions commit atomically, then the Global Transaction Manager does not need to explicitly specify a global serialization order. Rather, the global serialization order is always the global transaction commit order.

6.1.3.2 Timestamp-Ordered Databases

In Timestamp Order (TO) databases, a transaction is issued a timestamp when it begins, and the local database enforces a serialization order identical to the timestamp order. In this case, it is sufficient to ensure that the timestamps on the global subtransactions are ordered according to the global serialization order. Since timestamps are assigned when the transaction begins, it is sufficient to begin the global subtransactions in the global serialization order of their respective global transactions.

One problem with TO schemes occurs when the local database automatically restarts a transaction that has aborted due to a conflict. In this case, the restarted transaction would have a new timestamp, which could change its serialization order in the local database. If this can happen, the Agent would have to use one of the general schemes defined above, because it would have no way of deducing or affecting the restarted transaction's position in the serialization order.

6.1.3.3 Serialization Graph Testing Databases

In a Serialization Graph Testing (SGT) database, a serialization graph is maintained that makes the serialization order explicit. A history is serializable if the graph is acyclic. If an edge is added that completes a cycle, the transaction is aborted.

The problem with serialization graph testing is that the serialization point for a global subtransaction may occur outside of its lifetime [ED90]. This is because arcs can be added from the node representing a transaction in the SG even after the transaction commits. With serialization graph testing, the general approaches described earlier are unfortunately the only way to ensure that the local serialization order is consistent with the global serialization order.

6.1.3.4 Strict and Basic Databases

The strict and basic 2PL methods of concurrency control both allow a transaction to release its locks before it finishes executing. For example, a basic 2PL database transaction has a growing phase, when it takes locks, and a shrinking phase where it releases them. A strict 2PL database only allows read locks to be released early. With these schemes, we know that the serialization point is at the end of the growing phase. However, by the same argument we used with rigorous 2PL, we can choose the effective serialization point in a strict or basic 2PL database to be the time of the first unlock.

We see also that with these schemes, there is a gap of time before the commit where the transaction has released some resources that other transactions can access. This means that basic databases allow cascading aborts. Therefore, the Agent also should delay passing on a commit for any subsequent global subtransaction in the global serialization order until all transactions on the local database that preceded the global transaction have committed or aborted. This ensures that the global transactions avoid cascading aborts.

6.1.3.5 Conservative 2PL and SGT Databases

In the conservative schemes, each transaction is required to predeclare its read set and write set. With conservative 2PL, the Agent can submit a global subtransaction once all earlier global subtransactions in the global serialization order have executed their first instruction. At this point, we know that all earlier global subtransactions have completed their growing phase. If the new global subtransaction conflicts, its execution is blocked by the local database until it can obtain all of its locks. Otherwise, it is independent

of any running global subtransaction, and therefore can always be serialized after all of them in the local database.

With conservative SGT databases, the Agent can maintain an accurate shadow of the local serialization graph, because it knows each of its transaction's read and write sets at the time the transaction begins. Thus, the next global subtransaction in the serialization order can be submitted once the shadow serialization graph in the Agent indicates that the subtransaction will be serializable. A similar approach is used with Request-Ordered Linked Lists [PRR91].

6.2 Atomic and Semantically Atomic Commit

This section describes our theory concerning atomic and semantically atomic commitment in multidatabases. Based on this theory, we also present two new semantically atomic commit algorithms.

Semantic atomicity differs from normal atomicity in that it does not require a state-based undo. Rather, it maintains a notion of semantically reversing the effects of a transaction. With a semantic undo, other transactions may interleave their operations between the original operation on a data item and its compensating operation. Thus, the undo may not bring the database back to its original state. Rather, it brings the database back to some state that is *semantically equivalent* to the state that would have been reached if the original transaction had not been executed at all.

6.2.1 Compensation and Retry Properties

When problems occur during a global transaction commit, there are two approaches used to regain global transaction atomicity. One is to *compensate* for any committed work in a global transaction that needs to be undone. Another approach is to *retry* any subtransactions that did not commit in a global transaction that has already decided to commit.

Following the approach in [MRKS92], we introduce properties related to subtransactions and their ability to be compensated for or retried.

Definition 6.2.1 (Compensatable) *A global subtransaction is* compensatable *if its compensating subtransaction will always succeed, regardless of the state of the local database.*

Definition 6.2.2 (Retriable) *A global subtransaction is* retriable *if it can always be reexecuted successfully regardless of the state of the local database.*

Examples of compensatable subtransactions include a read-only transaction (which has a null compensating subtransaction), and a withdrawal (because the success of the compensating deposit does not depend on the state of the particular bank account). Similarly, a read-only transaction is retriable, as is a deposit. Note also that any global subtransaction can be both compensatable and retriable, or it can be just one, or it can be neither.

We define a subtransaction as *provisionally* compensatable or retriable if the success of the compensating subtransaction or the retry subtransaction does depend on the state of the underlying database. For example, a deposit is provisionally compensatable; as long as no one withdraws the money in the interim it can be compensated for. Similarly, a withdrawal is provisionally retriable.

Given that every subtransaction that can be compensated for has a compensating subtransaction defined for it, we can also specify the following theorems that relate the structure of a transaction and its compensating subtransaction to its properties:

Theorem 6.2.1 *A global transaction is compensatable if compensating code can be specified and executed, and there are no conditional branches in the compensating code that depend on information read from the database.*

Proof: Given that compensating code for the subtransaction can be specified and executed, we know that compensation can take place if the compensating subtransaction serializes directly after the original subtransaction in the local database history. This is basically equivalent to undoing that transaction, as no other local transaction or global subtransaction ever sees the effect of the original subtransaction.

If there are global subtransactions and/or local transactions serialized in between the subtransaction and its compensating transaction in the local database history, then these transactions may see the effects of the original subtransaction. Worse, they may change the values of data items that the compensating subtransaction may later read. Given that there are no conditional branches in the compensating code whose decision is based on the state of the database, then the compensating subtransaction is effectively (with respect to the local database) straight-line code and should run to completion despite these changes. □

Theorem 6.2.2 *A global transaction is provisionally compensatable if compensating code can be specified, even if it has conditional branches in the compensating code that depend on information read from the database.*

Proof: The first part of this proof follows from the first paragraph in the proof of the previous theorem. However, given that there may be global subtransactions and/or local transactions that can serialize between the original subtransaction and its compensating transaction, we must also assume that these transactions may change the values of data items read by the compensating subtransaction. If any of these changed values affects the decision made at some conditional branch, then the compensating code may not directly compensate for the original subtransaction and/or may not run to completion. However, if those values do not change in a way that affects the decisions at the conditional branches, then compensation could succeed. □

Note that we assume here that if a compensating subtransaction can be executed, it has permission to execute as well. Also, we assume that code that has no conditional branches that depend on the database state also calls no subroutines that have conditional branches that depend on the database state.

Theorem 6.2.3 *A global transaction is retriable if there are no conditional branches in its code that depend on information read from the database.*

Theorem 6.2.4 *A global transaction is provisionally retriable if there are conditional branches in its code that depend on information read from the database.*

The proofs of these two theorems are analogous to the proofs of the theorems about compensatable subtransactions.

One last thing we need to consider with respect to compensation and retry is what we gain if a subtransaction is running on a local database that supports a visible prepared state that could be used by the commit protocol. With respect to a prepared state, we have the following theorem:

Theorem 6.2.5 *A subtransaction in a local database that supports a visible prepared state can always be considered to be compensatable. However, its retry properties must depend on the structure of its code.*

Proof: From Definition 5.2.1, we know that compensatable subtransactions can always be semantically undone even after they commit in the local database. During a global commit, compensation will never be invoked by the commit algorithm once a global commit decision has been made. Thus, we can bound the time compensation may possibly be invoked as follows: it begins when the subtransaction commits in the local database and ends when the global transaction makes its decision to commit or abort.

We can emulate compensatability using the visible prepared state. When a local commit decision is made, we put the subtransaction in the prepared state. This guarantees that the subtransaction will eventually commit. Once the global decision is made, the subtransaction can commit or abort from the prepared state.

Retriable subtransactions can always be executed successfully, though early failures may cause the subtransaction to be re-executed several times. There is no way to use the prepared state to emulate a re-execution of the subtransaction. Therefore, the subtransaction's retry properties must be inherent to the subtransaction and not dependent on support from the local database. □

6.2.2 Protocols Used During Commit in Multidatabases

There are different kinds of protocols used in making a commit protocol atomic or semantically atomic in a multidatabase. In this section, we give a general classification of the protocols used according to their function during commit. For each class, we also give examples of different instances of that protocol that have actually been proposed for different multidatabase commit schemes.

6.2.2.1 Agreement Protocols

Agreement protocols are those used to enable the set of subtransactions in a global transaction (or some critical subset thereof) to come to a uniform decision on whether or not the transaction should commit. This decision cannot be revoked once it is made.

The most common example of an agreement protocol used for multidatabases is two-phase commit [BHG87]. In two-phase commit, a centralized coordinator asks each global subtransaction to vote whether to commit or abort. Just one "abort" vote from a global subtransaction implies that the whole global transaction aborts. The central coordinator makes the final decision and passes the result back to the local databases, who enforce the decision.

A second example of agreement protocols is Byzantine agreement [PSL80, Fis83]. Byzantine agreement protocols are less biased, in that "abort" votes must be received from some fractional subset of the global transactions before a global "abort" decision is made. They are meant specifically to work in failure-prone environments. As far as we know, Byzantine agreement has not been proposed for a multidatabase commit protocol. Intuitively, Byzantine agreement is probably quite a bit more complicated to implement correctly than two-phase commit, given the local database autonomy constraints.

6.2.2.2 Blocking Protocols

While an agreement protocol is executing, the global subtransactions need to maintain certain guarantees as to their ability to enact the final decision of the global transaction. This problem is made more complicated by the fact that the local decision on the local database is assumed to be independent from the multidatabase decision, because of the local database autonomy constraints.

In a normal distributed database, the prepared state acts to prevent any unwarranted problems during the global decision process. When the subtransaction on the local database enters the prepared state, the local database ensures that the subtransaction satisfies all of the database's correctness criteria, and therefore will commit. The local database also guarantees that the ability to commit persists even if the local database crashes. During the period the subtransaction is in the prepared state, the local database guarantees that no other transaction does anything to compromise the correctness of the subtransaction. Thus, when the final decision is made globally, that decision can definitely be enforced locally. Because the prepared state is preventing other transactions from interfering with the current subtransaction's eventual decision, we call this a "blocking protocol".

In a multidatabase we cannot guarantee that a the local database provides the ability for the multidatabase to put the subtransaction into the prepared state (i.e., the local database may not have a *visible* prepared state). Thus, the multidatabase must provide other protocols to block other transactions from interfering with its subtransactions during the period the information is being collected to make the global decision and also while that decision is being made and the results passed to the local databases.

Both active and passive blocking protocols have been proposed for multidatabases. An active blocking protocol is one that needs to be "turned on" and "turned off". A passive blocking protocol implements a low-level filtering at all times. The most widely-proposed examples of a passive blocking protocol partition the multidatabase according to what can be accessed in what ways by the global and local transactions [BST90, MRKS91, MRB+92a]. Basically, the partitioning allows the global multidatabase transactions to proceed without unwanted interference from the local transactions.

Several examples of active blocking protocols have been proposed. For example, in Hydro [PRR91], no transactions are allowed to pass operations to the local database while a global commit decision is being made. This type of blocking protocol requires that the multidatabase can intercept the local transactions, at least to the point where they can be delayed. Mullen*et al.* [MJS] propose a reservation protocol where the global transactions use additional data in the database to reserve the resources they need. Mullen's protocol requires all transactions, including local ones, be modified to test the reservation information. A third example of a blocking protocol is the denied local updates proposed for the 2PC agent method [WV90]. This involves slightly modifying the local databases to prevent other transactions from making updates while a global commit decision is being made.

While blocking protocols are required for many (but not all) global transaction commits, they do require violating the local database autonomy. Denied local updates require modifying the local transaction managers themselves. Hydro requires violating the control autonomy of the local database; that is, controlling

how the local databases receive their inputs from all of their users. Partitioning and the reservation scheme both require modifying the local database schemas. Mullen's reservation scheme also requires modifying all local and global transactions to ensure that they honor the reservations.

6.2.2.3 Stratification Protocols

In semantically atomic commit protocols, other transactions may see committed work of some transaction that is later undone. The constraint is that another transaction (global or local) should see either the committed effects of that transaction in *all* of the local databases, or it should see those effects in none of them. It should never read from both the committed global transaction on one local database, and the undone transaction on a different local database.

We call protocols that ensure this type of semantic atomicity during compensation "stratification protocols" after the Optimistic Control Protocol work by Levy *et al.* [LKS91a]. Stratification basically means that, where compensation is used to back out of a global transaction in a multidatabase, semantic atomicity is maintained.

Most multidatabase commit protocols that use compensation ensure stratification by making the compensating global transactions atomic. Levy's work gives two other protocols for stratification that are more complex, but allow for greater concurrency.

6.2.2.4 Serialization Protocols

It is well-known that one requirement for global transaction consistency in a multidatabase is that the different global subtransactions serialize consistently with the serialization order of the global transactions [GP86]. There are many techniques proposed to enforce this consistency during global transaction execution.

One problem with commitment of multidatabase transactions is that some commitment schemes allow a global commit decision to be made before local commitment of a global subtransaction is certain. In these schemes, the global subtransaction may need to be re-executed if a global commit decision is made, but the global subtransaction gets aborted (e.g., because of a process failure or a concurrency control violation). In this case, it is not always certain that the new global subtransaction will serialize in the same place in the global serialization order that the original global subtransaction serialized in. In fact, if later global transactions have been serialized after the one that has committed, then these transactions may need to be aborted if they executed on a local database that is re-executing some subtransaction of some earlier global transaction.

Serialization protocols ensure that, during any phase in a global commit protocol that follows the global commit decision (i.e., any phase that relies on retrying subtransactions), the retry transactions are serialized correctly in the global serialization order.

The most common serialization protocol is that, when some subtransaction aborts, all other global subtransactions that follow it in the global serialization order and that have already reached their serialization points must also abort, e.g., Hydro [PRR91].

Note here that, in all cases, a global transaction must complete its commit *before* any conflicting[1] global transaction that follows it in the global serialization order initiates its commit. Since the retry subtransaction execution may affect subsequent conflicting global transactions, those cannot commit until the earlier ones are complete. Therefore, they cannot yet return a "yes" vote in the voting phase.

Serialization protocols may be used in conjunction with blocking protocols to prevent other local transactions and global subtransactions from acquiring resources required for retry.

6.2.3 Subtransaction Information Flow and Its Properties

To understand the commit properties of global transactions, we first need to examine information flow between the subtransactions in the global transaction being committed. Let the *subtransaction information flow (SIF)* be a relationship

$$(T, \rightarrow_F)$$

[1] Conflicting subtransactions are ones that are executing concurrently on the same local database. These by definition are subtransactions of other global transactions. Because we do not assume that the multidatabase can determine the read and write sets of all transactions on a local database, we cannot adopt any stricter notion of conflict.

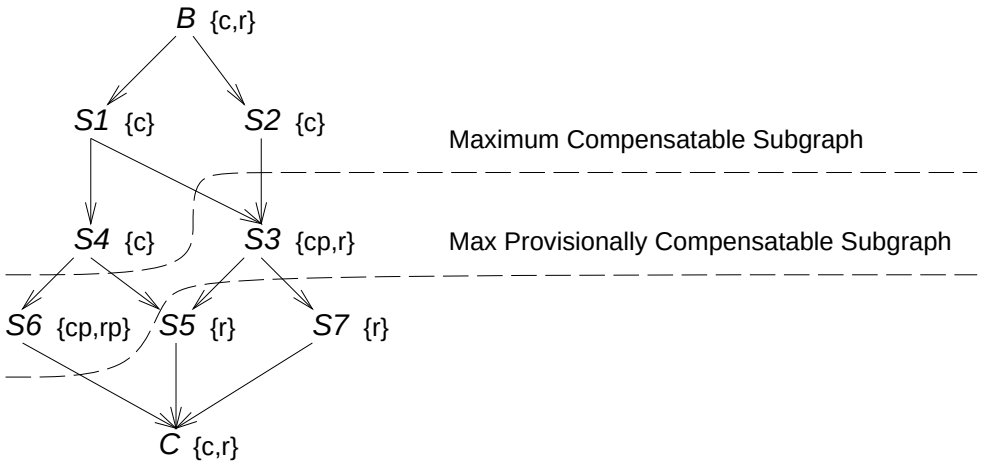


Figure 6.4: Example subtrees with respect to compensation.

where

1. T is the set of subtransactions in the global transaction, and
2. $T_a \rightarrow_F T_b$ if subtransaction T_a and T_b in T contain conflicting steps S_a and S_b, respectively, and S_b reads information local to the global transaction that S_a wrote.

Note that the relationship $\rightarrow_F$ may contain cycles, if subtransactions mutually read from each other. For the purposes of this discussion we begin with the assumption that the subtransaction information flow relationship $\rightarrow_F$ is acyclic. Later we will make extensions to deal with cycles in $\rightarrow_F$. We also assume that there is a begin marker in the subtransaction information flow, which we will call B, and that every subtransaction is a descendant of B. Similarly, we have a commit marker C, where every subtransaction has C as its descendant.

Let G be the directed graph representation of $\rightarrow_F$, with nodes for each subtransaction, as well as for B and C. Given the assumptions in the previous paragraph, we can see that G is a directed acyclic graph with a single source at B and a single sink at C. Define B and C to be both compensatable and retriable.

In this scheme, we label each subtransaction in the global transaction with a c if it is always compensatable or if it executes on a local database that supports a visible prepared state. We label it with a r if it is always retriable, a c_p if it is only provisionally compensatable, and a r_p if it is only provisionally retriable. When labeling is complete, each node in G thus has at most two labels, one of which is either c or c_p, and the other of which is either r or r_p.

Now, we make the following definitions with respect to the graph G:

Definition 6.2.3 (Maximum Compensatable Subgraph) *The* maximum compensatable subgraph *is the largest subgraph of G containing the node B in which all the nodes have c in their label (are compensatable) and whose ancestors are also all in the maximum compensatable subgraph.*

Definition 6.2.4 (Maximum Provisionally Compensatable Subgraph) *The* maximum provisionally compensatable subgraph *is the largest subgraph of G containing the node B in which all the nodes have either a c_p or a c in their label (are compensatable or provisionally compensatable) and whose ancestors are also all in the maximum provisionally compensatable subgraph.*

Figure 5.4 illustrates the maximum compensatable subgraph and the maximum provisionally compensatable subgraph for an example global transaction.

Definition 6.2.5 (Maximum Retriable Subgraph) *The* maximum retriable subgraph *is the largest subgraph of G containing the node C in which all of the nodes have an r in their label and for each node, all of its successors have an r in their label.*

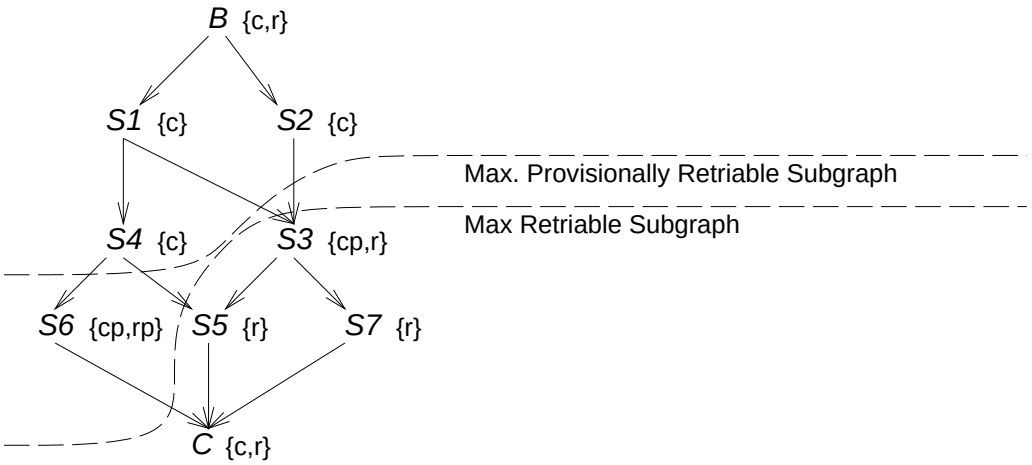


Figure 6.5: Example subtrees with respect to retry.

Definition 6.2.6 (Maximum Provisionally Retriable Subgraph) *The* maximum provisionally retriable subgraph *is the largest subgraph of G containing the node C in which all of the nodes have either an r_p or an r in their label, and for each node, all of its successors have an r or an r_p in their label.*

Figure 5.5 illustrates the maximum retriable subgraph and the maximum provisionally retriable subgraph for the same example global transaction that was used for the compensatable subgraphs.

The maximum compensatable subgraph is a subgraph of the maximum provisionally compensatable subgraph. Similarly, the maximum retriable subgraph is a subgraph of the maximum provisionally retriable subgraph.

The maximum compensatable subgraph and the maximum retriable subgraph define the portion of the global transaction execution that can always be undone and the portion that can always be redone, respectively. In these algorithms we are also interested in the rest of the graph, so we have the following definition:

Definition 6.2.7 (Pivot Subgraph) *The* pivot subgraph *is the subgraph containing those nodes in G that are neither in the maximum compensatable subgraph nor in the maximum retriable subgraph.*

The pivot subgraph characterizes the "problem area" with respect to global commit algorithms.

Figure 5.6 shows two different cases of global transactions where the pivot subgraph is not empty. Note also that the global transaction shown in Figures 5.4 and 5.5 has a pivot subgraph containing only the node *S6*.

6.2.4 Semantically Atomic Commit Theory

Semantic atomicity differs from normal atomicity in that it does not require a state-based undo. Rather, it maintains a notion of semantically reversing the effects of a transaction. With a semantic undo, other transactions may interleave their operations between the original operation on a data item and its compensating operation. Thus, the undo may not bring the database back to its original state. Rather, it brings the database back to some state that is *semantically equivalent* to the state that would have been reached if the original transaction had not been executed at all.

6.2.4.1 Unconditional Compensatability and Retriability

Global transaction commit processing is constrained somewhat by the order $\rightarrow_F$. This is because if a subtransaction must be retried, all subtransactions that follow it in $\rightarrow_F$ must also be retried to ensure that they read the correct information. Starting at the source, intuitively we want to commit each global

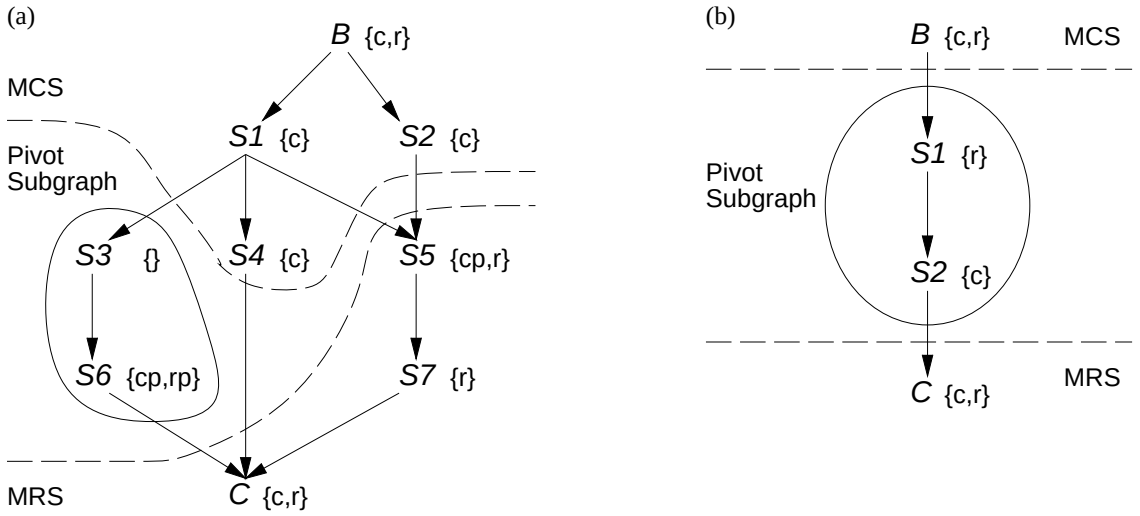


Figure 6.6: Example pivot subgraphs. In this figure, MCS labels the maximum compensatable subgraph and MRS labels the maximum retriable subgraph. (a) shows a pivot subgraph that contains a subtransaction that is neither compensatable nor retriable. (b) shows a pivot subgraph in a case where all subtransactions are either compensatable or retriable, but two of the subtransactions are neither in the maximum compensatable subgraph nor the maximum retriable subgraph.

transaction once all of its predecessors in $\rightarrow_F$ have committed. Since the maximum compensatable subgraph contains the largest portion of the global transaction execution that can always be compensated for, it also represents the set of global subtransactions that can always be committed locally before a global decision is made. Thus, we have the following theorem:

Theorem 6.2.6 *The maximum compensatable subgraph contains the maximum set of subtransactions that can unconditionally be locally committed before a global decision is made, if semantic atomicity is to be maintained.*

Proof: Since it is the *maximum* subgraph, we know that any successors to nodes in the subgraph are either provisionally compensatable, not compensatable at all, or have at least one predecessor that is not compensatable.

If a successor is not compensatable, then committing that successor early would prevent a later abort decision, because that decision would require the successor to be compensated for.

If a successor is provisionally compensatable, then committing that successor early would cause its resources to be released, and potentially overwritten. This overwriting could prevent the compensating subtransaction from succeeding if a later abort decision were made. For example, a deposit is provisionally compensatable because the bank balance needs to remain high enough for the compensating withdrawal to succeed. An intermediate withdrawal could cause the compensating subtransaction to fail. Thus, if the provisionally compensatable subtransaction were committed early, other conditions would have to be placed on the local database until the global decision is final, to ensure that the compensating subtransaction would succeed.

If a successor has some predecessor that is not compensatable, then that successor cannot be committed until its predecessor commits, and its predecessor cannot commit early. If we commit the successor early, and the predecessor later decides to retry, the successor would have to be compensated for before the retry could occur, as the successor's effects are already visible in the multidatabase. □

We also have a symmetric situation with respect to the maximum retriable subgraph:

Theorem 6.2.7 *The maximum retriable subgraph contains the maximum set of subtransactions whose local commit can be unconditionally delayed until after a global decision is made, if semantic atomicity is to be maintained.*

Proof: Since it is the *maximum* subgraph, any predecessors to its nodes are either provisionally retriable, not retriable at all, or have a (different) successor that is not retriable.

If a predecessor is not retriable, then a global commit decision could not be made, because there is no provision (semantically) to insist that the global subtransaction's effects persist in the local database.

If the predecessor is provisionally retriable, then making a global commit decision before it commits would leave the global transaction open to the risk that some intervening transaction would change the state of the local database such that the retry would not succeed. Therefore, conditions would have to be placed on the local database's execution to ensure that the retry subtransaction would read the same local database state.

If the predecessor has some successor that is not retriable, then making a global commit decision leaves open the risk that the predecessor will abort and need to be retried. If the predecessor is not retriable, this is not an option. However, if the predecessor is retriable, it will be retried *along with all of its successors.* Given that a successor exists that is not retriable, this retry process will fail.□

6.2.4.2 The Pivot Subgraph

The pivot subgraph contains those transactions for which an atomic global commit protocol must be followed. Because G is a directed acyclic graph, the pivot subgraph is also a directed acyclic graph. Because of this, we can use a framework similar to the global transaction framework above to critically examine the pivot subgraph to determine its internal properties and how they would affect an atomic commit protocol.

In dealing with the pivot subgraph, we examine the maximum provisionally compensatable subgraph to determine which subtransactions can conditionally commit before a global decision is made. We examine the maximum provisionally retriable subgraph to determine which subtransactions can conditionally be retried after a global decision is made. Because of these conditions, however, a blocking protocol must run on the local database. If the subtransaction on that database is provisionally compensatable, the blocking protocol must run from the time the local commit is done to the time the global decision is made. If the subtransaction on that database is provisionally retriable, the blocking protocol must run from the time the global commit decision is made to the time the subtransaction commits.

The subtransactions in the pivot subgraph may be neither provisionally compensatable, provisionally retriable, compensatable or retriable. Such subtransactions we call *pivot subtransactions.* An example of a pivot subtransaction would be one that fires a missile.

Intuitively, we can see that if a global transaction has more than one pivot subtransaction, it cannot be atomically committed. This is because the pivot subtransactions need to make a unanimous decision. However, their decisions are made independently. Once a pivot subtransaction has made a decision, it cannot change its mind. Therefore, there is no provision for recovery if a pivot subtransaction makes an independent decision that differs from the global decision.

For atomic commit of the pivot subgraph, we partition the nodes in the pivot subgraph into three categories. The first contains nodes that are in the maximum provisionally retriable subgraph of G. The second partition contains nodes that are also in the maximum provisionally compensatable subgraph. The third partition, called the *pivot subgraph's pivot*, contains the remaining nodes in the pivot subgraph. These nodes are neither in the maximum provisionally compensatable subgraph of G nor the maximum provisionally retriable subgraph of G.

Figure 5.7 shows the pivot subgraph's pivots for the global transactions in Figure 5.6.

Note here that the pivot's pivot may contain multiple subtransactions, but they are all not necessarily pivot subtransactions. It could contain the nodes in the subgraph $T_a \rightarrow_F T_b$, where T_a is provisionally retriable only and T_b is provisionally compensatable only. A similar situation is shown in Figure 5.7(b).

The pivot subgraph's pivot contains the set of subtransactions whose local decisions *must* be identical to the global decision, because their local decision is *always* final. It also defines the minimum number of subtransactions that must participate in a global, atomic commit protocol for the global transaction. Thus, we have the following lemma:

Lemma 6.2.1 *If the pivot subgraph's pivot contains more than one subtransaction, then an atomic commit or abort decision by the nodes in the pivot subgraph's pivot is not possible.*

The proof follows directly from the fact that no global agreement protocol can run between multiple completely autonomous subtransactions. □

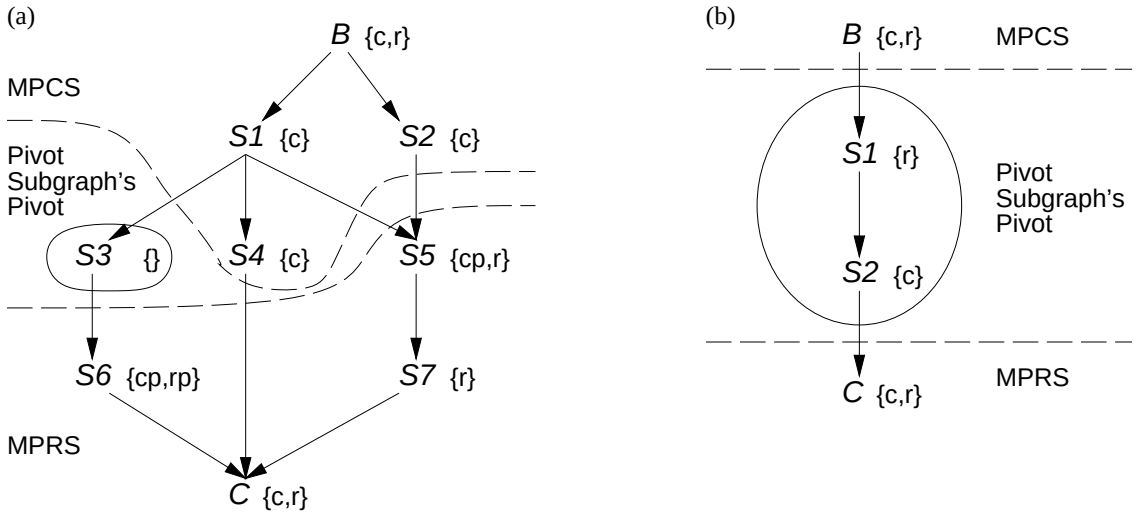


Figure 6.7: Example pivot subgraph's pivots. In this figure, MPCS labels the maximum provisionally compensatable subgraph and MPRS labels the maximum provisionally retriable subgraph.

This lemma leads directly to the following theorem:

Theorem 6.2.8 *If the pivot subgraph's pivot contains more than one subtransaction, then an atomic or semantically atomic commit is impossible for the global subtransaction.*

The proof follows from lemma 5.2.1 and the fact that the minimum set of subtransactions that must agree on the decision is the set that is in the pivot subgraph's pivot. □

Note that in Figure 5.7 a semantically atomic commit decision can be made despite the presence of a pivot subtransaction. However, a semantically atomic commit is not possible for the global transaction in Figure 5.7(b).

6.2.5 A General Framework for Semantically Atomic Commit Algorithms

The general framework for a semantically atomic commit has five phases. The five phases are shown in Table 5.2.5. They are executed in roughly the order shown in the table. Phase 1 operates concurrently with execution – once a subtransaction has completed its execution it can commit immediately. If an abort decision is made during phase 1, a stratification protocol must be used to ensure that compensation is correct. Phase 2 begins once a global transaction commit is explicitly requested. In phases 1 and 2, subtransactions can commit concurrently regardless of $\rightarrow_F$. This is because their compensating subtransactions can run independently. A blocking protocol must be run on the local databases that have subtransactions that commit during phases 2-4. This is because these phases deal primarily with subtransactions that are provisionally compensatable or provisionally retriable. The state of the local database must be maintained for the compensation or retry to succeed.

The global decision is made during Phase 3. Once a global commit decision has been made, compensation is no longer allowed. The primary means to regain consistency becomes subtransaction retry. The blocking protocol can be removed at any time once all of the uncommitted subtransactions that remain are in the maximum retriable subgraph.

Most known commit algorithms for multidatabase transactions fit in this framework. They differ principally in what phase they classify the different subtransactions, and which phases they ignore. For example, the local commit before global decision algorithms all do everything in phases 2-3, and ignore phase 4. Similarly, the global commit before local decision algorithms all use phases 3-4, and ignore phase 2. Semantically atomic commit algorithms all include some form of phase 1 and/or phase 5. For instance, Mehrotra et.al.'s non-blocking commit algorithm [MRKS92] uses mainly phases 1 and 5, with their pivot transaction being the only one that uses a different phase (phase 3).

Phase	Subtransaction Types	Protocols	Comments
1	From Maximum Compensatable Subgraph	T	Can commit once subtransaction ends its execution.
2	From Max. Provisionally Compensatable Subgraph	B	Can commit concurrently. Cyclic information flow allowed.
3	Any	B	Agreement protocol run here.
4	From Max. Provisionally Retriable Subgraph	B,S	No cyclic information flow allowed.
5	From Maximum Retriable Subgraph	S	Can commit once subtransaction ends its execution and previous subtransactions in $\rightarrow_F$ are committed.

Table 6.1: Phases of Semantically Atomic Commit. The first column shows the phase number (in execution order). The second column defines the subgraphs from which subtransactions may commit in the phase. The protocol column defines which protocols must be in effect during the phase (B=blocking, T=stratification, S=serialization). The last column comments on how the phase could be executed.

6.2.6 Cyclic Subtransaction Information Flow

In this section, we extend this theory to deal with situations where the subtransaction information flow $\rightarrow_F$ contains a cycle.

6.2.6.1 Theory

Our approach with respect to commitment of a global transaction with at least one cycle in its subtransaction information flow is to treat each cycle as if it were a single node (cycle contraction). This requires that the contracted node associated with the cycle be labeled with the weakest label possible. We can label the contracted node as compensatable only if all subtransactions in the cycle are compensatable. We can label it as provisionally compensatable only if all subtransactions are either compensatable or provisionally compensatable. We follow a similar strategy with retriable. If some subtransaction is in more than one cycle, then all of the nodes in its strongly connected component must contract into the same node.

When a determination is made as to which nodes belong to which subgraphs, the label associated with that node is used to determine which subgraph all of the nodes in the cycle belong in.

Cycles create a problem if they are a part of the maximum provisionally retriable subgraph but not a part of the maximum provisionally compensatable subgraph. This is because such cycles must be processed after a global commit decision has been made and thus are part of the retry phase of the commit algorithm. Since these retries must be processed in subtransaction information flow order, it is impossible to retry the cycle. Thus, we have proved the following theorem:

Theorem 6.2.9 *If a cycle exists among subtransactions that must be processed after the global decision, no semantically atomic commit is possible.*

Note that if there is a cycle in the pivot subgraph's pivot (the part of the subtransaction information flow in neither the maximum provisionally retriable subgraph nor the maximum provisionally compensatable subgraph), the commit is not possible by Theorem 5.2.8.

6.2.6.2 Implications

Because of the problems with cycles, we can see that with any atomic or semantically atomic global commit algorithm, we have a trade-off as to whether the algorithm should be biased towards compensation or retry.

Let us examine the subtransactions in the intersection of the maximum compensatable subgraph and the maximum retriable subgraph. These subtransactions may commit in any one of the phases described in Section 5.2.5. If we try to commit them in phase 1, we have no problems if some subset of these forms a cycle in the subtransaction information flow. This is because the compensating transactions are all independent.

If we try to commit them during phases 2-4, we may be causing unnecessary blocking, because they are not a part of the pivot subgraph. If we try to commit them during phase 5, we run into problems if some subset of them forms a cycle, preventing retry. This would indicate that, if cycles are a concern, we should try to commit all of the subtransactions in this intersection during the phase 1.

A second argument can be made based on the time it takes to process the global commit. Committing more subtransactions during phase 1 does not appreciably add to the time it takes to process the global commit, because the subtransactions can commit concurrently. However, in phase 5, we must commit subtransactions in the subtransaction information flow order. Committing additional subtransactions during this phase may appreciably lengthen the global commit. This too indicates that we should try to commit all the subtransactions in this intersection during the first phase.

One last argument can be made based on whether we want the global transaction to be biased towards committing or aborting. If we process the subtransactions in the intersection during phase 1 and one of them aborts or returns a "no" decision, the global transaction aborts. If we process all of them during phase 5 and one of them aborts or returns a "no" decision, it will be retried and the global transaction will eventually commit.

Similar arguments can be made with respect to the pivot subgraph, except that we consider the intersection of the maximum provisionally compensatable subgraph and the maximum provisionally retriable subgraph.

6.2.7 Commit Algorithms

This section describes two commit algorithms that we have derived from the general structure provided in the earlier section. We also discuss the constraints that each algorithm requires on the global transactions with respect to the compensatability and retriability of the subtransactions.

The first algorithm described here is the *Best Effort Algorithm*. It takes the position that it will try to commit the global transaction if the commit is at all possible. That is, once the commit has started, it will work forward to ensure a commit where possible, even when "no" votes are received.

The second algorithm defined here is a minimally-blocking algorithm. It's objective is to require a blocking protocol to be used on the local databases for as little time as possible.

6.2.7.1 The Best Effort Algorithm

It is not always true that a single global subtransaction abort must force its entire global transaction to abort. In particular, the database itself can initiate aborts because of deadlock or other transaction conflicts. These aborts are not necessarily caused by problems internal to the global transaction. Rather, they are usually caused by some problem in the running environment, such as a process failure or a bad interaction with some other transaction.

In this commit algorithm, we bias the decision process towards committing the global transaction. That is, we make a global decision immediately and then assume we can enforce that decision. This commit protocol assumes that all of the subtransactions are either retriable or provisionally retriable.

In this algorithm, the commit of a global transaction may take a long time. This is true, for instance, in the situations where one of the subtransactions is aborted at commit time, and is subsequently retried. This causes (at least) the re-execution of an entire subtransaction on some local database in the middle of the global transaction commit process.

The Best Effort Algorithm commits the subtransactions in the global transaction according to the subtransaction information flow. If some subtransaction aborts and has to be re-executed, any subtransaction following it in the information flow order must also be aborted and re-executed (because it depends on incorrect results). It is a "local commit before global decision" scheme in [MR91], with the voting phase proceeding in an order defined by the subtransaction information flow. The detailed algorithm follows:

Algorithm 6.2.1 (Best Effort Commit) *1. Start running the blocking protocol on each local database that executed some subtransaction of the global transaction.*

2. Make a global commit decision.

3. *For each subtransaction S, once all of its predecessors in $\rightarrow_F$ have committed, do the following until the commit of S succeeds:*

   (a) *If S has aborted, re-execute it.*

   (b) *Commit S.*

   (c) *If the commit fails, abort S using the subtransaction abort algorithm defined below.*

   (d) *If the commit succeeds, terminate the blocking protocol on S's local database.*

Algorithm 6.2.2 (Subtransaction Abort) 1. *Abort S if it is still running.*

2. *Abort any subtransactions (of other global transactions) and any local transactions that are serialized after S in the local database.*

3. *For each subtransaction T that is a descendant of S in $\rightarrow_F$:*

   (a) *Abort T.*

   (b) *Abort any subtransactions (of other global transactions) and any local transactions that are serialized after T in the local database.*

6.2.7.2 Minimally Blocking Algorithm

The Minimally Blocking Commit algorithm uses every phase of the general framework defined earlier, and classifies each subtransaction in the strictest category possible. Thus, all of the subtransactions in the maximum compensatable subgraph and/or the maximum retriable subgraph are processed in either phase 1 or phase 5. All of the subtransactions that are in the maximum provisionally compensatable subgraph and/or the maximum provisionally retriable subgraph are processed in either phase 2 or phase 4.

In the Minimally Blocking algorithm, we first commit all of the subtransactions in the maximum compensatable subgraph. If all commit, we follow some sort of *atomic* commit decision process to make a commit/abort decision on those subtransactions in the pivot subgraph. This decision becomes the global transaction decision as well. If the pivot subgraph is empty, then the global decision defaults to commit.

If the global decision is to commit, then all of the remaining subtransactions (which are, by definition, in the maximum retriable subgraph) are committed in the order $\rightarrow_F$. This process is identical to the one in the best effort scheme. If some subtransaction has to be retried, then all of the subtransactions that follow it in $\rightarrow_F$ also have to be retried.

In this approach, we assume that the failure of a retriable transaction may also cause global transactions that follow the current one in the global serialization order also to be aborted. Thus, some form of serialization protocol must be used when a retry fails.

Note that the algorithm defined in [MRKS92] follows the structure of the algorithm defined above, but it only operates on global transactions whose pivot subgraphs contain at most one subtransaction. Also, they assume no dependencies between global subtransactions.

We can now present a full algorithm for multidatabase commit that preserves semantic atomicity, and that minimizes its interference with the local database executions

Algorithm 6.2.3 (Minimally Blocking Commit) *Input: Labeled graph.*

1. *Commit all of the global subtransactions in the maximum compensatable subgraph. If one aborts, make a global abort decision, compensate for the committed subtransactions, and return abort. Note that the commits can all proceed concurrently, because the compensating subtransactions should be independent of one another.*

2. *Execute the pivot subgraph commit algorithm defined below. It makes a global commit or a global abort decision. If the global decision is to abort, then compensate for all of the committed subtransactions and return abort.*

3. *At this point, the global decision is to commit. Commit the remaining subtransactions in an order consistent with $\rightarrow_F$. If one of these commits fails, abort it using the subtransaction abort algorithm (Algorithm 5.2.2) initiate new retry subtransactions in $\rightarrow_F$ order. These retry subtransactions can be committed as soon as they successfully complete their execution and all of their predecessors in $\rightarrow_F$ commit.*

4. *Return commit.*

The atomic commit for the pivot subgraph proceeds as follows:

Algorithm 6.2.4 (Pivot Subgraph Commit) *Input: Labeled pivot subgraph.*

1. *Partition the global subtransactions in the pivot subgraph into three sets:*
   (a) *The subtransactions that are also in the maximum provisionally compensatable subgraph of G.*
   (b) *The subtransactions that are also in the maximum provisionally retriable subgraph of G, but are not in partition a.*
   (c) *The rest of the subtransactions. For atomic commit to succeed, this should contain at most one subtransaction.*
2. *Initiate a blocking protocol on the local databases that have subtransactions in the pivot subgraph.*
3. *Commit all of the subtransactions in partition a. If any of these abort (a "no" decision is returned), make a global decision to abort. Compensate for all of the committed transactions, terminate the blocking protocol, and return abort. Note here that the commits of these transactions do not need to be ordered by $\rightarrow_F$ because compensating subtransactions are independent of one another.*
4. *If there is a subtransaction in partition c, commit it. If the commit fails, make a global decision to abort and proceed as in the previous step. If the commit succeeds or the partition is empty, make a global commit decision.*
5. *Commit all the subtransactions in partition b, in the order indicated by $\rightarrow_F$. If one of these commits fails, abort it using the subtransaction abort algorithm (Algorithm 5.2.2) initiate new retry subtransactions in $\rightarrow_F$ order. These retry subtransactions can be committed as soon as they successfully complete their execution and all of their predecessors in $\rightarrow_F$ commit.*
6. *Once all subtransactions have committed, terminate the blocking protocol and return commit*

6.3 Summary

In this chapter we discussed the two requirements for maintaining global transaction atomicity in a multidatabase. These requirements are (1) to maintain a consistent global serialization order across all local databases and (2) to enforce atomic (or semantically atomic) commit.

We provided a general framework for classifying different existing schemes for enforcing a consistent global serialization order across all local databases. We proved that, if no assumptions are make concerning the local databases, enforcing a consistent global serialization order is hard. Following the observations in Chapter 2 concerning the parallels between database concurrency control schemes and multidatabase schemes for maintaining a consistent global serialization order, we proposed two classifications for the multidatabase schemes. In normal schemes (roughly analogous to pessimistic concurrency control schemes), the multidatabase dictates a global serialization order. The Agents for the local databases delay or block global subtransactions to ensure that their local database serializes them in an order consistent with the global serialization order. In certification schemes (roughly analogous to optimistic concurrency control schemes) the global subtransactions are immediately submitted to the local databases. Each Agent deduces the local serialization order for its database. When a global transaction commits, the multidatabase examines the different local serialization orders and either certifies that the global transaction was serialized consistently or aborts it.

We also proposed ways that the Agents can deduce the local serialization orders of their databases given knowledge of the local database's concurrency control mechanisms. This work parallels work done by Elmagarmid and Du [ED90].

With respect to atomic and semantically atomic commitment we introduced a theory for understanding whether (and how) any global transaction can commit atomically. We took a graph-theoretic approach to characterizing global transactions based on the information passed between its subtransactions. We introduced a method for determining syntactically whether a global subtransaction can be compensated for and/or re-executed; also, whether the compensation and/or re-execution requires any conditions to be maintained on the information in its local database in order to succeed.

We defined four types of protocols which are necessary as building blocks in atomic and semantically atomic commit protocols. Agreement protocols are used to reach a unanimous decision to commit or abort. Blocking protocols are used to maintain conditions on the state of the information in the local database during critical parts of the commit process. Stratification protocols ensure that, if a global transaction is semantically undone, other transactions perceive either all or none of its committed effects. Serialization protocols ensure that, if a subtransaction must be re-executed, the new subtransaction is ordered consistently with the rest of its global transaction in the global serialization order. Blocking, stratification, and serialization protocols all compromise local database autonomy.

Based on the graph-theoretic description of a global transaction and an understanding of the protocols, we proved several things concerning a global transaction's ability to commit. We identified a critical subgraph, the *pivot subgraph's pivot*, that must contain at most one node for the commit to be possible. We identified a subgraph that must not contain any cycles for commit to be possible. We determined how to compute the minimum subgraph that contains the nodes representing the subtransactions that require blocking during commit (the *pivot subgraph*).

Lastly, we defined a general framework that atomic and semantically atomic commit protocols must follow to ensure correctness. Based on this framework, we presented two new multidatabase commit algorithms; the Best Effort algorithm and the Minimally-Blocking algorithm.

Chapter 7

Multidatabase System Design

This chapter describes how planning tasks are supported in our prototype multidatabase system, Mongrel. The basic multidatabase environment in which the planning application executes is shown in Figure 7.1. In this figure, the applications are responsible for accomplishing different tasks in the multidatabase. Just as a normal database executes some set of transactions, the multidatabase executes some set of tasks. The multidatabase is responsible for ensuring that the set of tasks it executes maintains data consistency.

A multidatabase has two logical levels. The local level consists of the set of local databases. All of the information is stored in the multidatabase at the local level. The global level of the multidatabase provides a way to access in a uniform way the collection of information in the local databases. Thus, it needs to have a notion of how the global subtransactions on the different local databases must cooperate to accomplish a single global task. The global level must operate without any cooperation from the local level, as the local databases usually exist long before they are collected together in the multidatabase.

In Mongrel, each task is executed at the global level of the multidatabase as a single Interaction. The Interaction runs subtasks in the form of global transactions. At the local level, the part of the global transaction that runs on a single local database is a *global subtransaction*. It is the global subtransactions that actually do the work of accessing and updating the information in the multidatabase. This structure is shown within the task in Figure 7.1.

7.1 Overview

Figure 7.2 shows the process structure that we use to support planning tasks on a multidatabase. The work done by an Interaction ultimately is accomplished by running atomic database transactions *(global subtransactions)* on the local databases that make up the multidatabase. Other applications may also submit *independent transactions* to the multidatabase.

It is the job of the *Interaction Manager* to coordinate the Interactions and to make specific guarantees about the correctness of their execution. To this end, it is primarily responsible for coordinating the ordering of the different global transactions that execute as a part of each Interaction. Currently, the Interaction Manager is centralized, though we are not precluding distributing it in the future.

The *Global Transaction Manager* coordinates the execution of the different global transactions. It determines the global serialization ordering on the local databases, and also coordinates the emulated two-phase commit. Thus, it ensures that the global transactions are executed atomically in the multidatabase.

One *Agent Manager* is associated with each local database. It has an associated GSO Enforcer that works with the global transaction manager to ensure that the resulting history containing all global subtransactions and independent transactions is serializable with all global transactions serializing in the global serialization order. This can be done using any one of the known algorithms (e.g. [PRR91, GRS91, MRB+92b]).

Each global subtransaction that is executing on a local database on behalf of some global transaction has a *Global Subtransaction* process. The Global Subtransaction process begins a global subtransaction on the local database for the global transaction, executes the steps initiated by the global transaction on the local database, and eventually manages the local database's end of the two-phase commit process for the global transaction.

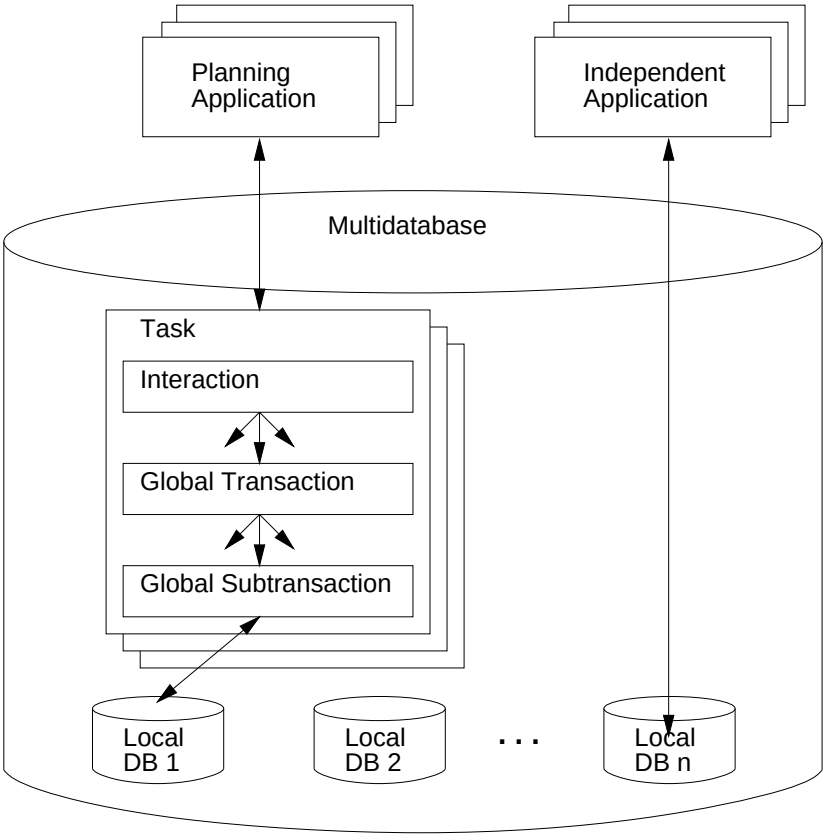


Figure 7.1: Multidatabase Environment

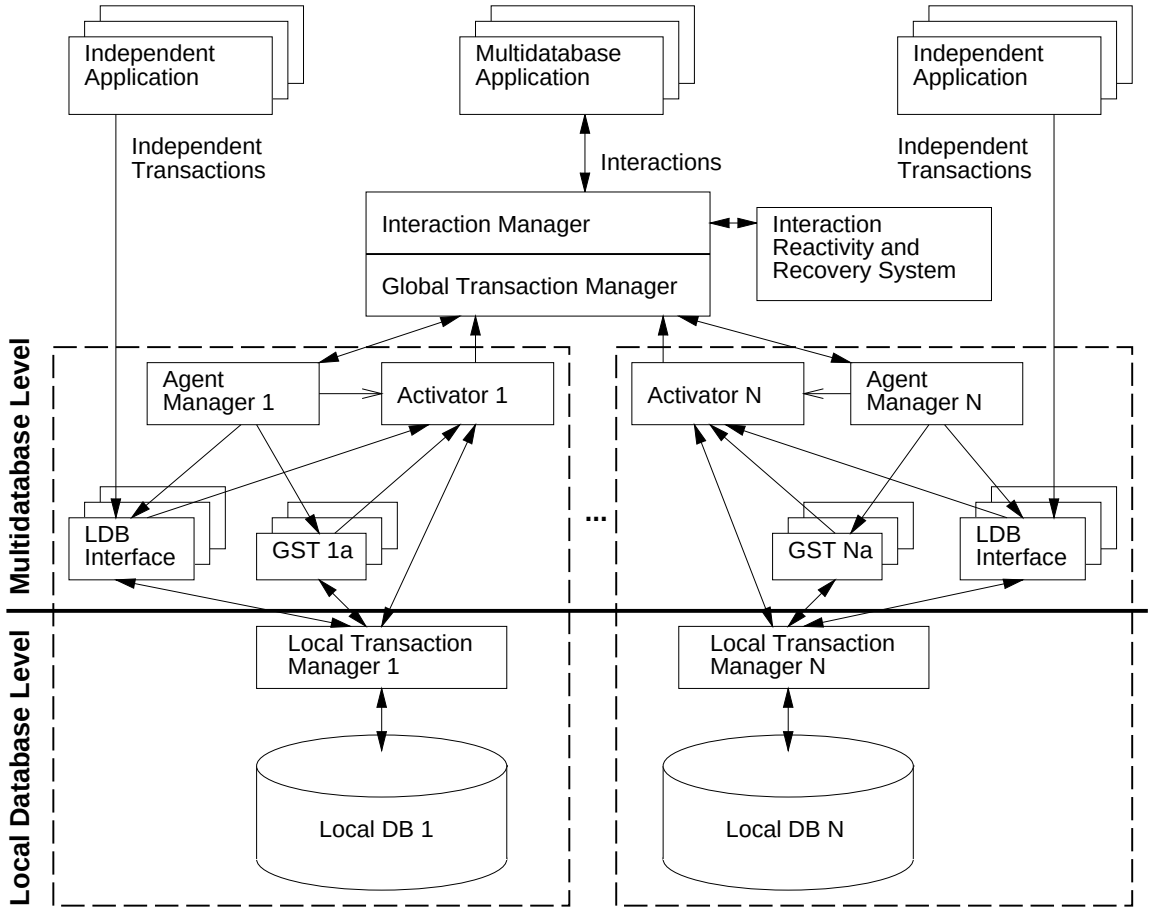


Figure 7.2: Process-level diagram of the Multidatabase.

Because we enforce isolation in the local databases using global subtransactions to hold resources, we require that the local database transactions preserve the ACID properties (atomicity, consistency, isolation, durability), and that they are executed serializably.

Because of problems associated with emulating the two-phase commit, the multidatabase also has to maintain a small amount of control over other transactions that run on the local databases besides the global subtransactions. This is done by the *Local Database Interface* processes for the independent transactions.

The job of the *Activator* is to detect events (update operations) in the local database that are of interest to the active Interactions in the multidatabase, check whether the associated conditions still hold, and inform the relevant Interactions when a condition is violated. The Activator receives a specification of events and conditions to monitor for from the Interaction Manager (via the Agent). It also receives from every Global Subtransaction Process and Local Database Interface a copy of the input stream to the local database that it uses to determine if some undesirable update event (i.e., weak conflict) may have occurred. If so, it generates a query to check the condition associated with the weak conflict. If the condition no longer holds, it passes back a notification event to the Interaction Manager specifying the weak conflict that was violated. The Interaction Manager can then take steps to recover. The Activator maintains its own local database interface functions for coordination during two-phase commit.

The *Interaction Reactivity and Recovery System (IRS)* coordinates both Interaction recovery and replacement when a weak conflict is violated. Global transaction recovery (from failure) basically comes free with the recovery systems in the local databases. The Interaction Reactivity and Recovery System is invoked when an Interaction is aborted or when a weak conflict is violated. It schedules the multidatabase-level recovery or replacement actions to ensure that the resulting Interaction execution remains correct.

In the following sections we discuss the details of the system design for our prototype multidatabase, Mongrel. Each section focuses on a different multidatabase function.

7.2 Atomic Global Transaction Management

The multidatabase supports atomic transaction management as a basis on which to build the Interactions. In this section, we separate out system design issues with respect to maintaining transaction atomicity into different parts: maintaining a global serialization order, emulating a two-phase commit for global transactions, and managing interference from independent transactions.

7.2.1 Global Transaction Execution

A global transaction is executed as a set of global subtransactions, each of which is an atomic transaction on some local database. A global transaction may only execute on a subset of the local databases in the multidatabase, but no local database has more than one subtransaction of the same global subtransaction running on it.

At the global level of the multidatabase, the Global Transaction Manager maintains information on each of the active global transactions, as shown in Figure 7.3. The information maintained for each global transaction includes its global transaction identifier and the set of local databases that have been accessed by the global transaction up to the current point in its execution.

At the local database level, most local databases act as servers with each transaction initiated by a single client. This requires that each global subtransaction run as a separate process, and each process maintain its own connection to the local database. All queries and updates generated by a global transaction must be routed through its global subtransaction processes.

The Agent Manager for each local database acts as a dispatcher to forward all instructions it receives from a global transaction to the appropriate global subtransaction process. It also funnels all responses from the global subtransactions back to their respective global transactions.

7.2.2 Maintaining a Consistent Global Serialization Order

As was stated in Section 5, one condition that must be maintained to ensure global serializability is that all local databases must serialize the global transactions consistently on their local databases. Chapter 5

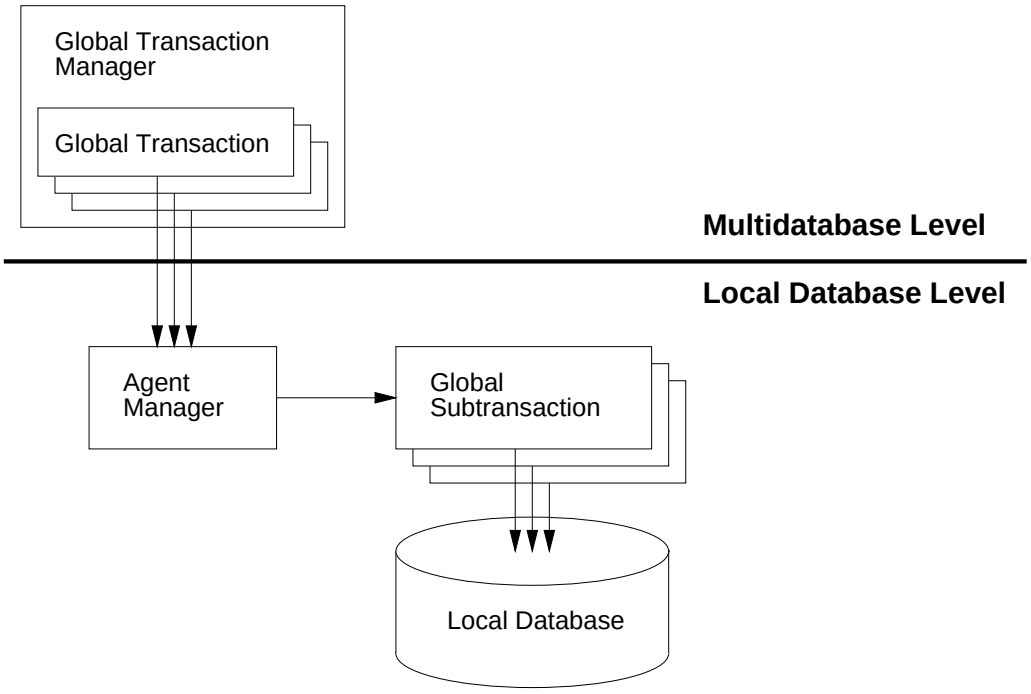


Figure 7.3: System Modules for global transaction execution

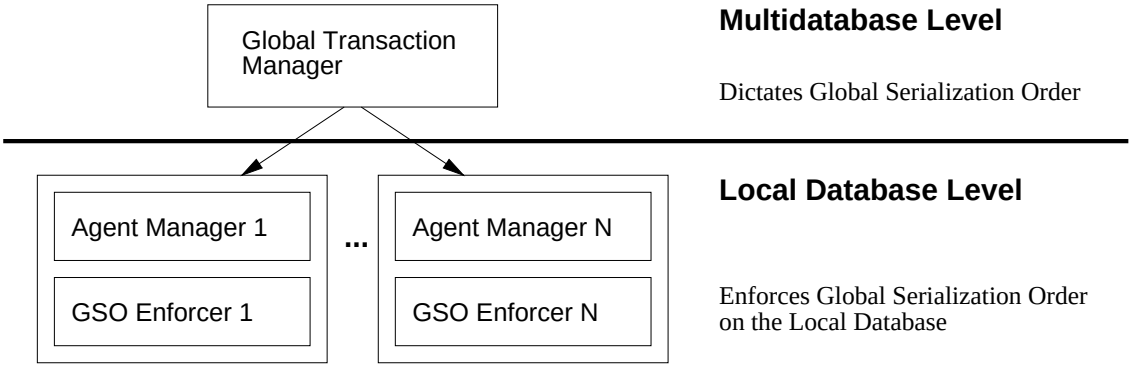
defines algorithms to detect serialization events and ensure consistent ordering. This section describes how those algorithms are actually embedded in the system.

In Figure 7.4, we show the portion of the multidatabase system that enforces global serialization order. In the multidatabase, the local serialization orders are kept consistent with some global serialization order by having a Global Transaction Manager that works in concert with GSO Enforcers for each local database. Exactly how this is done depends on the types of concurrency control in the local databases and whether or not the consistency of the local serialization orders is enforced optimistically or pessimistically.

Pessimistic enforcement of the global serialization order means that the global serialization order is dictated by a centralized source, in this case the Global Transaction Manager. The different GSO enforcers are responsible for taking the dictated global serialization order and enforcing the order of the global subtransactions on the local databases to be consistent with it. The order the Global Transaction Manager dictates can be consistent either with the global transaction begin order or the global transaction commit order. The GSO enforcers can either monitor the serialization events in the global subtransactions on the local database, or generate a forced local conflict in the dictated order. We use forced local conflict only when the serialization effects on the local database cannot be detected easily. When inconsistencies are detected, the GSO enforcer can either delay the execution of the global subtransaction, or abort the associated global transaction.

When the global serialization order is enforced optimistically, then the GSO enforcers deduce the serialization order of the global subtransactions on their local databases. They then report these serialization orders to the Global Transaction Manager, which compares the different local serialization orders to detect inconsistencies. When an inconsistency is detected, the Global Transaction Manager aborts global transactions as needed to ensure that the final schedule is consistent. Each GSO Enforcer can either deduce the local database's serialization order by monitoring for serialization events, or it can deduce a serialization order by generating local conflicts.

(a) Pessimistic Schemes



(b) Optimistic Schemes

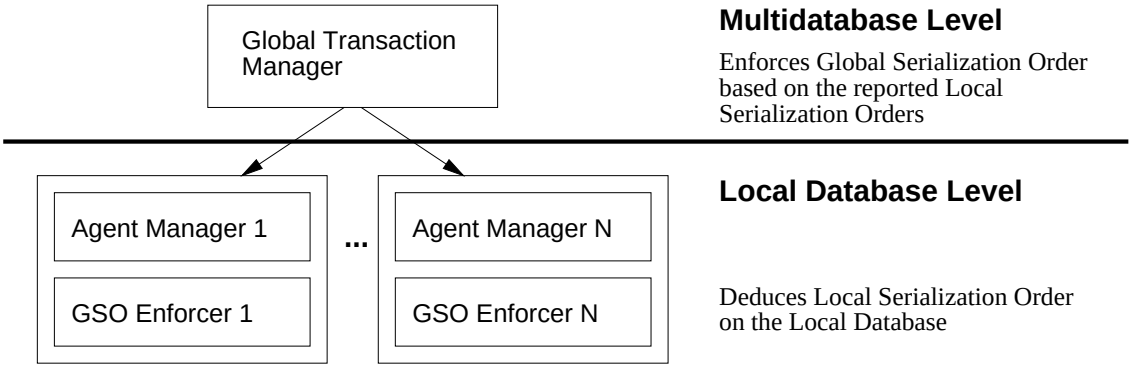


Figure 7.4: Modules used in ensuring a consistent serialization order for all local databases.

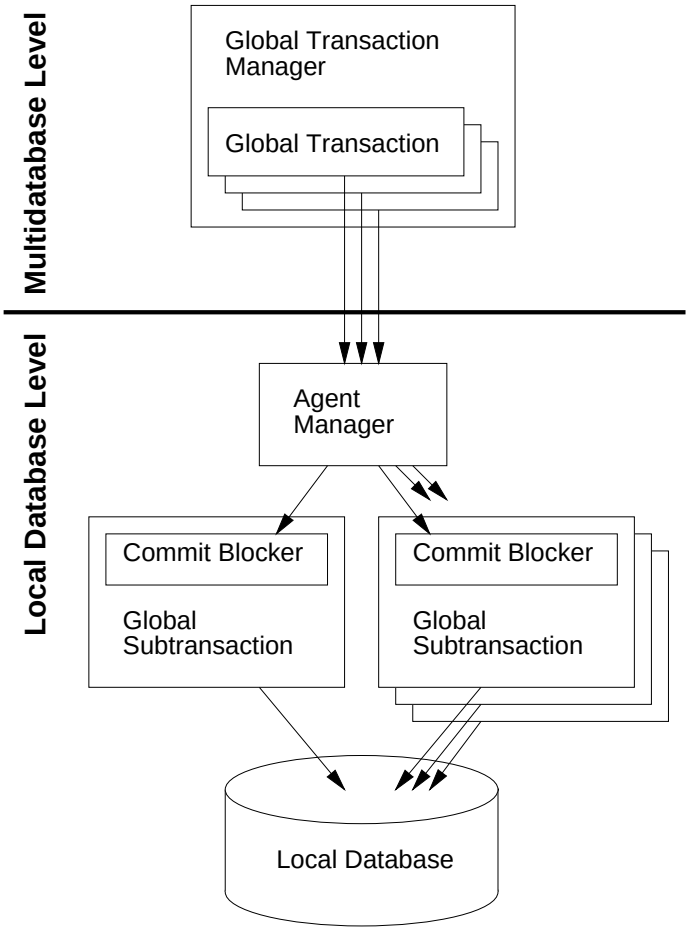


Figure 7.5: System modules for emulating two-phase commit

7.2.3 Emulating Two-Phase Commit

A second requirement identified in Section 4.1.6 for ensuring global transaction atomicity is that an atomic global commit protocol must be used. Emulating a two-phase commit protocol in a multidatabase is a harder problem than maintaining a consistent global serialization order, because it is not as easy to deduce the commit behavior of a local database.

In our prototype, we implement a local commit before global commit strategy. Figure 7.5 shows the required system modules. During global transaction execution, the Global Transaction module maintains the subtransaction information flow for the specific global transaction. The Global Transaction module also coordinates the two-phase commit process. It requests votes from each global subtransaction that participated in the global transaction, in the subtransaction information flow order.

The crux of any local commit before global commit strategy is that if a global subtransaction commits but its global transaction aborts, the committed global subtransaction must be semantically undone before other transactions read or overwrite its erroneously committed results. Because of this, the local commit before global commit strategies specify that all requests to the local database cease while the global subtransaction is in an emulated prepared state. Also, if problems occur that cause the global transaction to be fully or partially aborted, other transactions that have gained the global subtransaction's resources (e.g., because they were queued waiting for a lock) may need to be aborted in order for the global subtransaction to complete successfully.

Because the Global Subtransaction processes may need to block all traffic to the local database for a

period of time, each Global Subtransaction process contains a *Commit Blocker*. In the unblocked state, the Commit Blocker just processes all traffic received by the Global Subtransaction as usual. The Commit Blocker receives block requests from the Agent Manager, which puts the Global Subtransaction into the blocked state. In the blocked state, the Global Subtransaction queues all traffic received from the Global Transactions until an unblock message is received, then processes the queued messages in order.

When the Agent Manager receives a vote request for one of its global subtransactions from the Global Transaction Manager, it not only passes the vote request on to the appropriate Global Subtransaction Process, but it also sends a block message to all of the other Global Subtransaction Processes. Once the decision is known, the Agent Manager forwards an unblock message to all the Global Subtransaction Processes.

When an Agent Manager receives a "no" vote for some committed Global Subtransaction, it initiates a compensating subtransaction as defined in Section 7.5. If the compensating transaction fails to commit, the Agent Manager may direct Global Subtransactions that follow the failed one in the global serialization order to abort, so the compensating subtransaction can be successfully retried.

7.2.4 Dealing with Independent Transactions

Independent transactions execute directly on individual local databases, and consequently run outside the domain of the multidatabase. That is, their execution is autonomous from the execution of any transaction on the multidatabase. When we consider autonomy of the local databases, it is the execution of independent transactions that ultimately causes the most serious problems.

The autonomous execution of independent transactions can impact the atomicity of the global transactions. For instance, the execution of an independent transactions can cause two global subtransactions in the multidatabase to be serialized in the opposite order from their execution order. This problem can be solved by regulating the serialization orders of the global subtransactions on the local databases, as defined in Section 7.2.2.

A far more serious problem with the independent transactions is that they can impact the emulated two-phase commit process described in Section 7.2.3. Because we use a local commit before global commit strategy, we must be able to block all access to the local database while the global subtransaction is in the emulated prepared state. This includes blocking all traffic to the local databases initiated by the independent transactions as well as all the traffic initiated by other global subtransactions. Thus, with this strategy, the multidatabase must interfere with the control autonomy of the independent transactions, in that it must occasionally block their execution to ensure the correctness of its own transactions. Also, if some Global Transaction receives a "no" vote during the commit, and compensation fails, the Agent Manager may tell some independent transaction to abort.

In our prototype, we insert a local database interface into the connection between the independent transaction and the local database, as shown in Figure 7.6. Each independent transaction has its own Local Database Interface, which contains a commit blocker that behaves identically to the commit blocker in the Global Subtransaction Processes. Thus, the Local Database Interface has a connection directly to the Agent Manager, and receives directions to block and unblock the message stream between the independent application and the local database. In the unblocked configuration, the traffic passes directly through. In the blocked configuration, traffic to the multidatabase is queued for transmission once the Local Database Interface is unblocked.

Schemes for global transaction commitment that do not require the existence of a commit blocker for the independent transactions would be preferred for the sake of local database autonomy. However, it has been shown in [SKS91a] that such schemes cannot exist.

7.3 Step Execution

The global transactions in an Interaction are specified using *steps*, where there is some flexibility as to which steps are actually executed to accomplish the subtask of the particular global transaction. Each step runs on a single local database, and is represented as a call to some procedure in that local database's *Step Library*. The system modules required to execute steps on the local database are shown in Figure 7.7.

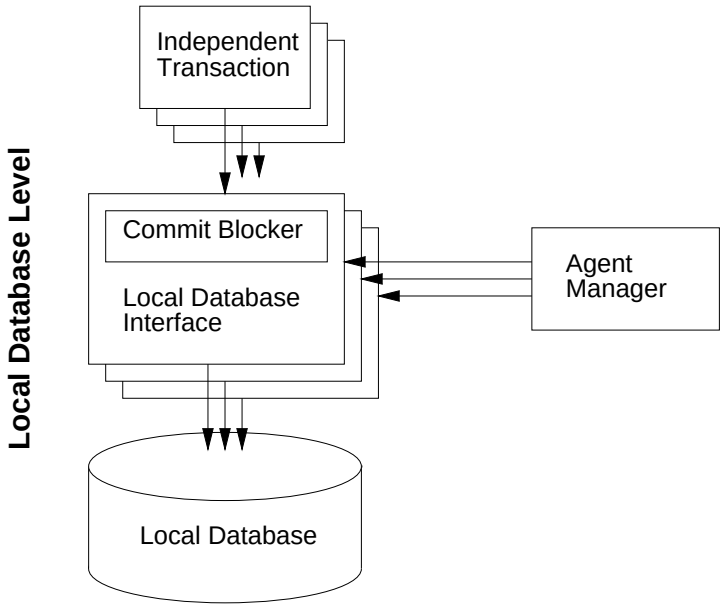


Figure 7.6: Independent transactions and their Local Database Interfaces

The following sections describe the step libraries and the step functions in them, and the mechanism for executing steps as a part of a global subtransaction.

7.3.1 Step Libraries

Associated with each local database is a standard interface of procedure calls that can be invoked by the multidatabase as it is running its global transactions. These are the procedures that are used by the multidatabase; in some sense, the procedure calls encapsulate the data in the local database in a similar manner to the way objects encapsulate their data representations behind an interface of method calls. This interface and the procedures that implement it are called the local database's *Step Library*.

Each local database makes public the procedure calls for the different steps in its Step Library, including the step's function name, its step number on the local database, the number and types of its arguments, and the type of its return value. This interface is accessible by the TaSL Interpreter as needed to type check the step calls. It also can be accessed by the Interaction definer as needed.

Since the steps translate a specific function into the local database's DML, they are also in a position to deduce the information required to compensate for themselves, as well as supplementary information such as the data items read and written (and values) required by the replacement algorithm. This may in some cases require supplementary queries to the local database beyond the minimum commands needed to actually execute the step. The step itself is responsible for returning this information as well as its own return values from executing its function on the local database. The compensating step information, and the read/write data items and values, are processed and logged by the global subtransaction object. Thus, a step call to do an update would cause a set of procedure calls like the one shown in Figure 7.8.

7.3.2 Executing Steps in Global Subtransactions

Interactions are defined through an interface to an interpreter for the multidatabase manipulation language TaSL. In TaSL, steps of global transactions are specified as procedure calls to routines in a step library, specified in the same thread as the global transaction is executing. The steps themselves are parsed by a *TaSL Interpreter* in the multidatabase application into a local database, a step number, and a list of arguments.

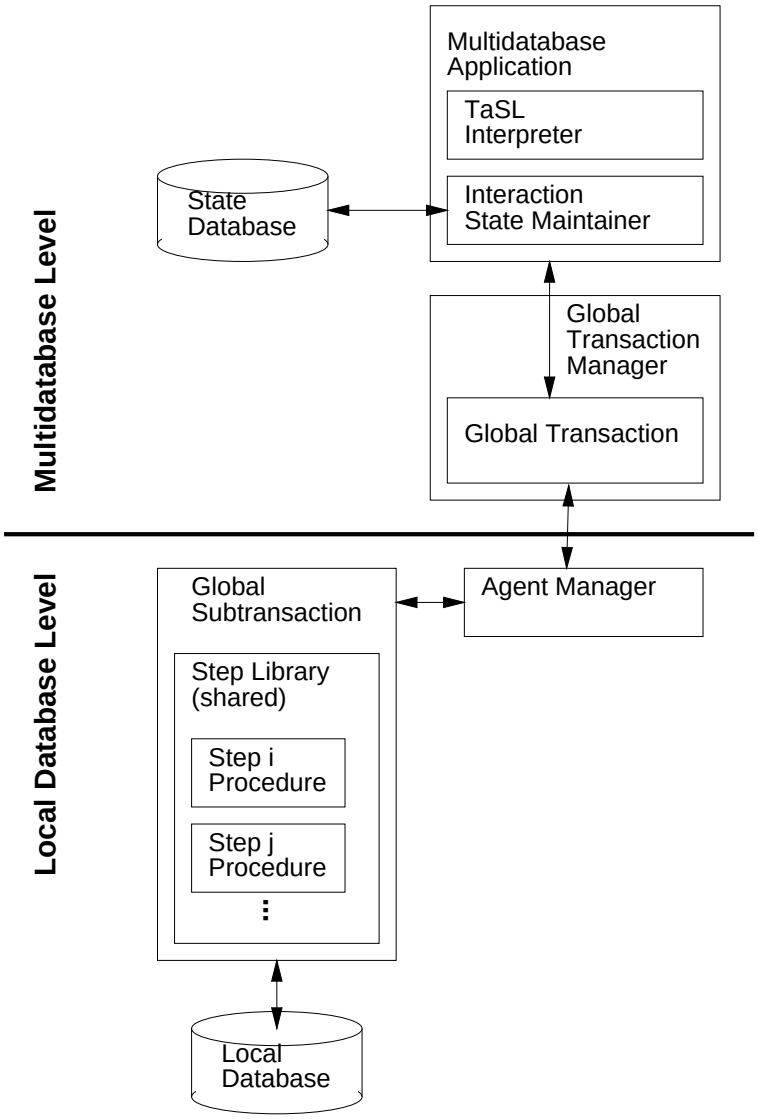


Figure 7.7: Steps and the Step Library

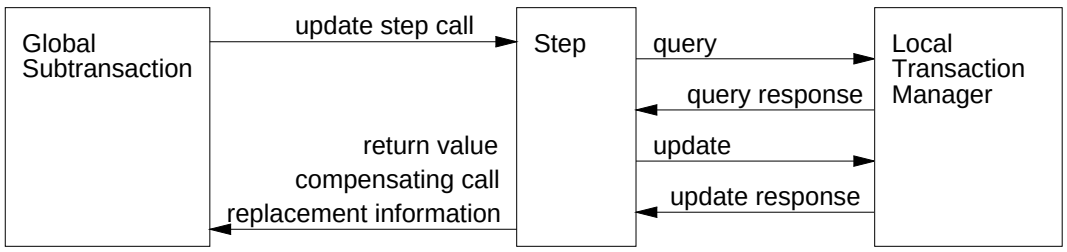


Figure 7.8: Procedure calls for a step that updates.

The TaSL Interpreter maintains a directory of active steps and their argument types for type checking. The types of the arguments in the list are typechecked based on the information in the directory, and the request to execute the step is passed down to the global transaction object only if the types are correct.

The request to execute the step is passed intact through the Global Transaction and the Agent Manager of the local database on which the step is run, to the Global Subtransaction process associated with the step's global transaction and its local database.

The Global Subtransaction Processes themselves make the call to the step's procedure in the shared Step Library. This is done by taking the step number, using it to access the function pointer to the step via a jump vector, and executing that function with the specified arguments. The call executes the step in the context of the already running transaction on the local database. The step itself generates some requests to the local database in the local data manipulation language. It formats the return value into the expected format, and returns the results as well as the other information to be logged. The Global Subtransaction issues the commands to log the step's call and return value, as well as the compensating step function and parameters, and the read/write data items and values required for compensation. It then passes the return value back up to the multidatabase application. Once the application has received the return value, the step is complete and the next step may be initiated, or the global transaction committed.

7.4 Managing Interactions

According to Definition 6.5.6 for correct Interaction Histories, a history of a set of concurrent Interactions is correct if the projection of each Interaction from the history is view equivalent to some compensation-correct execution of that Interaction. The global transactions of the different Interactions themselves may be arbitrarily interleaved in the history.

In this section, we show how the system generates correct executions for each Interaction. First we discuss how the multidatabase schedules global transactions for an Interaction, and how the dependencies among the different global transactions in that Interaction are maintained. Then we discuss how events are specified, detected, and reported. Last, we discuss how reactivity works to maintain the correct execution of that Interaction in the presence of a conflict that provokes an event.

7.4.1 Interaction Scheduling

Since Interactions can arbitrarily interleave at the global transaction level, merely ensuring that the global transactions are executed atomically is enough to ensure that the operations of different Interactions are interleaved correctly. The major problem with Interaction scheduling is ensuring that the different global transaction of a single Interaction are ordered (serialized) correctly in the execution.

The different modules at the global level associated with a single Interaction are shown in Figure 7.9. Each active Interaction's current execution is represented in the Interaction Manager as a single Interaction object. That Interaction has a set of global transaction objects representing its current set of uncommitted global transactions.

The effective execution order of the global transactions in an Interaction is specified in the multidatabase application. The order is represented within the Interaction as a set of *execution dependencies*. Execution dependencies are computed as the Interaction is executed by the part of the multidatabase application that parses the TaSL Interaction specification. Execution dependencies are computed in the following circumstances:

- If two global transactions T_a and T_b run sequentially in the same thread, then T_b has an execution dependency on T_a.
- If a global transaction T_c forks a new execution thread, with T_d being the first global transaction that is executed on the new thread, then T_d has an execution dependency on T_c.
- If a set of threads joins, and T_e is the first global transaction executed on the thread after the join, then T_e has execution dependencies on each of the last global transaction that executed on the joining threads.

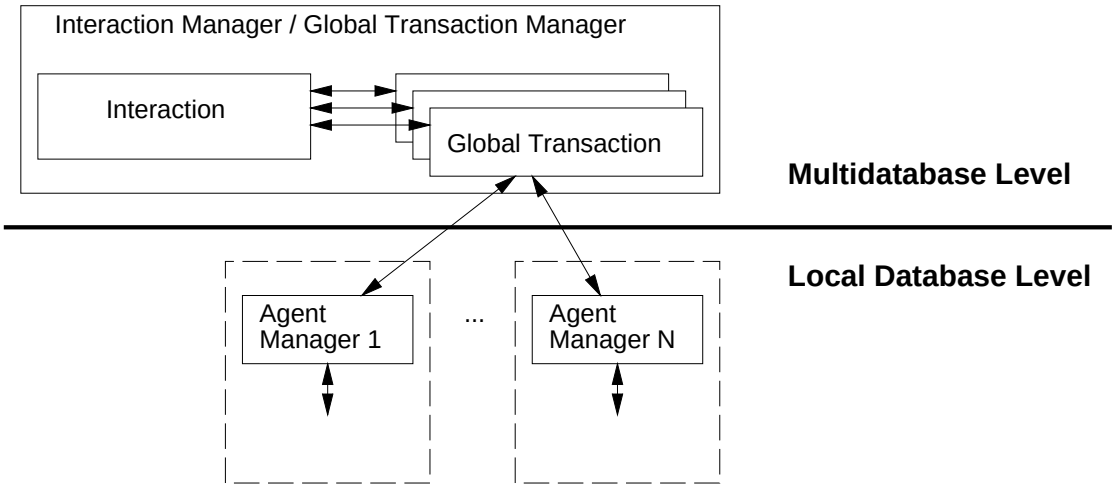


Figure 7.9: Modules involved in scheduling and executing global transactions in an Interaction.

When a transaction T_z has an execution dependency on some other global transaction T_y, this is reported to the Global Transaction Manager. It is the job of the Global Transaction Manager to ensure that T_y serializes before T_z in the global serialization order. Also, because we need to maintain the open nested recoverability property (Definition 4.2.1), we also constrain T_z to commit after T_y.

7.4.2 Maintenance of Interaction State and State Dependencies

Each Interaction has an internal state represented by a set of Interaction variables. The state also contains information on what the Interaction definition is, and what of that has been executed.

Since Interactions are intended to support very long-lived applications, it is not practical for a single process to maintain the state of an Interaction internally. This is because it is very likely that the process will fail for some reason before the Interaction completes. Also, different execution threads of the same Interaction may run concurrently but access the same state. Instead, we stably save the state for each Interaction in an Interaction state database. This portion of the system is shown in Figure 7.10.

The Interaction State Maintainer implements the functions in the Interaction Storage and Management layer. It ensures that the entire Interaction definition is stably stored. For each potential alternative global transaction (as specified by the multidatabase application), the Interaction State Maintainer also keeps track of its history, with a notation of which of the following categories this global transaction definition fits into:

- Was executed, but failed to accomplish its subtask.
- Was executed, and succeeded in its subtask.
- Was not executed.

For each variable in the Interaction state, the State Maintainer also keeps a copy of its name, its value, and the identifier of the global transaction that wrote the value to the variable. These can be organized in a per-Interaction basis, as no Interaction should ever access another Interaction's state.

State dependencies form between global transactions when one global transaction reads the Interaction's state as written by another global transaction. That is, they occur when some global transaction T_a writes a value to some state variable, and some other global transaction T_b in the same Interaction later reads the value that was written. These dependencies are detected by the Interaction State Maintainer. When a global transaction writes a state variable, its global transaction identifier is saved as well as the new value. When another global transaction reads that state variable, the State Maintainer detects the state dependency and reports it.

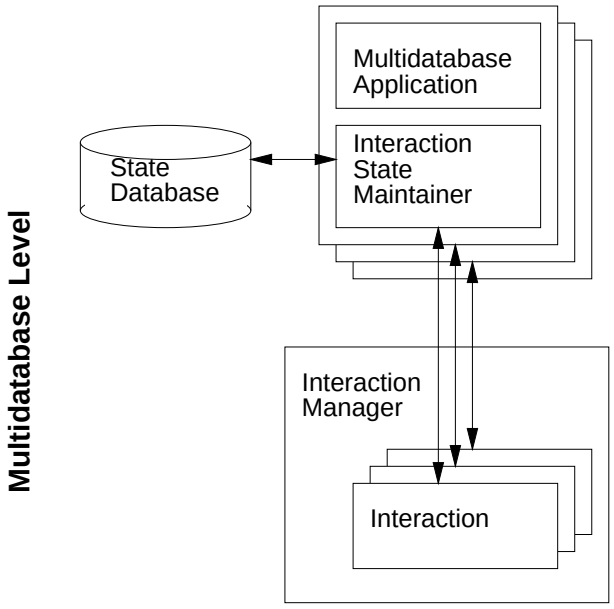


Figure 7.10: Maintaining Interaction State.

When some transaction T_z has a state dependency on some other global transaction T_y, the application reports the dependency to the Global Transaction Manager. The Global Transaction Manager ensures that T_y serializes before T_z in the global serialization order.

7.4.3 Specifying, Detecting, and Reporting Events

An event is either an update of some data item that violates some condition on the value of that data item, or a deletion of some data item. An event is specified as a get step (similar to the steps described in Section 7.3) to get the value of the item, and a condition which is either "exists" or a simple boolean condition on the value retrieved by the get step. If the condition is violated, the event is also tagged with a global transaction to roll back to.

The modules involved in event specification and detection are shown in Figure 7.11. Events are associated with specific updates done by global transactions, and the conditions that must be maintained for a global transaction's work to remain valid are set within that global transaction. When a global transaction specifies an event and a condition, it sends a message to the Activator associated with the local database that maintains the data item in question, via its Agent Manager.

The Activator maintains its own persistent *event table*, whose entries specify the name of a relation or set in the local database and the set of events concerning that relation that the Interaction Manager is currently interested in. When an Activator receives a message to begin monitoring the database for a particular event, it first determines what relation or set the event is associated with. For that relation, it then inserts information in the event table concerning the object and/or the condition that must be maintained on the object. The Activator also polls the database to ensure the condition is initially satisfied. If it is, then it returns an acknowledgment to the Agent.

When the Activator receives a message to stop monitoring the local database for some event, it removes that event's entry from its event table. All of an Interaction's events are removed when it terminates.

There are two algorithms we can use for determining when the Activator should check the local database for condition violations. The first method is a simple polling method: The Activator wakes up periodically and generates queries to get the values of the different data items in question.

The second method is a bit more complicated, but saves query traffic on the local databases. Call the collection of active Global Subtransaction Processes and Local Database Interfaces the *Local Transaction Client Processes*. In this method, we assume that each of the Local Transaction Client Processes makes a

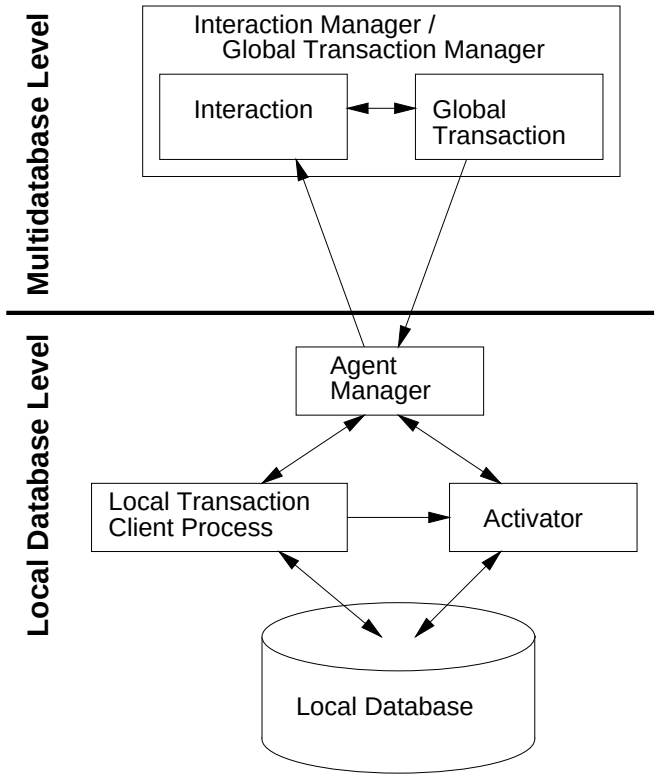


Figure 7.11: Modules involved in Event Specification and Detection.

copy of its traffic to the local database. This copy is then forwarded to the Activator. The Activator parses each message sent to the local database. The query messages get discarded, but the update messages are checked to see what collections/relations or objects are being updated.

The Activator determines whether or not an update operation may have caused an event by checking if the operation changed some object in a set of relation that has associated events in the Activator's event table. If so, it checks the local database to see whether or not constraints were violated by the update event. This is necessary because the update may not have violated the condition; also because the update operation may have been done by a transaction that later aborted.

Say the Activator parsed some update associated with some local database transaction T, and decided that the update may have triggered one or more of its events. The Activator then generates a read-only transaction R to read the information from the database that it needs to check whether the conditions still hold. If some condition does not hold, then the Activator notifies the Interaction Manager of the events associated with those conditions. In order for the check to be correct, R has to serialize after T in the local database's serialization order if R and T conflict. This happens by default, however. If there is an actual conflict on the local data item, then the conflict forces the serialization order. If there is no conflict, then the event could not have occurred.

This scheme has one disadvantage that it is impossible in the general case to determine for certain which transaction on the local database actually did the update that caused the condition to be violated. As this information would only be needed if we wanted to abort or semantically undo the transaction that did the update. However, since we place the burden of recovering from conflict on the transaction that was overwritten rather than on the overwriter, this is not a problem.

Once a value for a data item has been retrieved form the database (by either method) it is checked against the appropriate conditions. If some condition is violated, the Interaction that specified the event is notified. This message goes to the Interaction object, because the global transaction object probably no longer exists. Also, events (wait events or weak conflict events) really belong to the entire Interaction, as their scope is really outside that of an individual global transaction.

Events specifying weak conflict violations indicate that some reactive action should be taken. The application specifier is notified of the event, and it specifies a committed global transaction to roll back to. Exactly how this happens in the system is described in the next section.

7.5 Logging, Recovery, and Reactivity

With Interactions, both reactivity in the event of a weak conflict and recovery after a process, communication, or media failure require logged information. Both also operate by (possibly semantically) undoing and redoing work. Semantic recovery of open nested transactions in the case of failure has been defined in several contexts, e.g. [GS87]. However, using recovery techniques to evolve an Interaction from an inconsistent state to a consistent state after a weak conflict violation presents a different type of problem.

In this section, we define the logging and reactivity/recovery subsystems for Interactions in a multidatabase.

7.5.1 Logging Subsystem

In our multidatabase system, we maintain two levels of logging. At the global level, we maintain all of the information necessary to recover an Interaction correctly, including the information on the specific global transactions. At the local level, for each local database, we log information on the global subtransactions that is needed to deduce and execute the compensating subtransactions. These logs are dual-purpose. They serve to support both compensation-based failure recovery and the replacement algorithm invoked after a weak conflict violation.

7.5.1.1 Global-Level Logging

The global level log maintains the following information about each active Interaction:

1. When the Interaction begins.

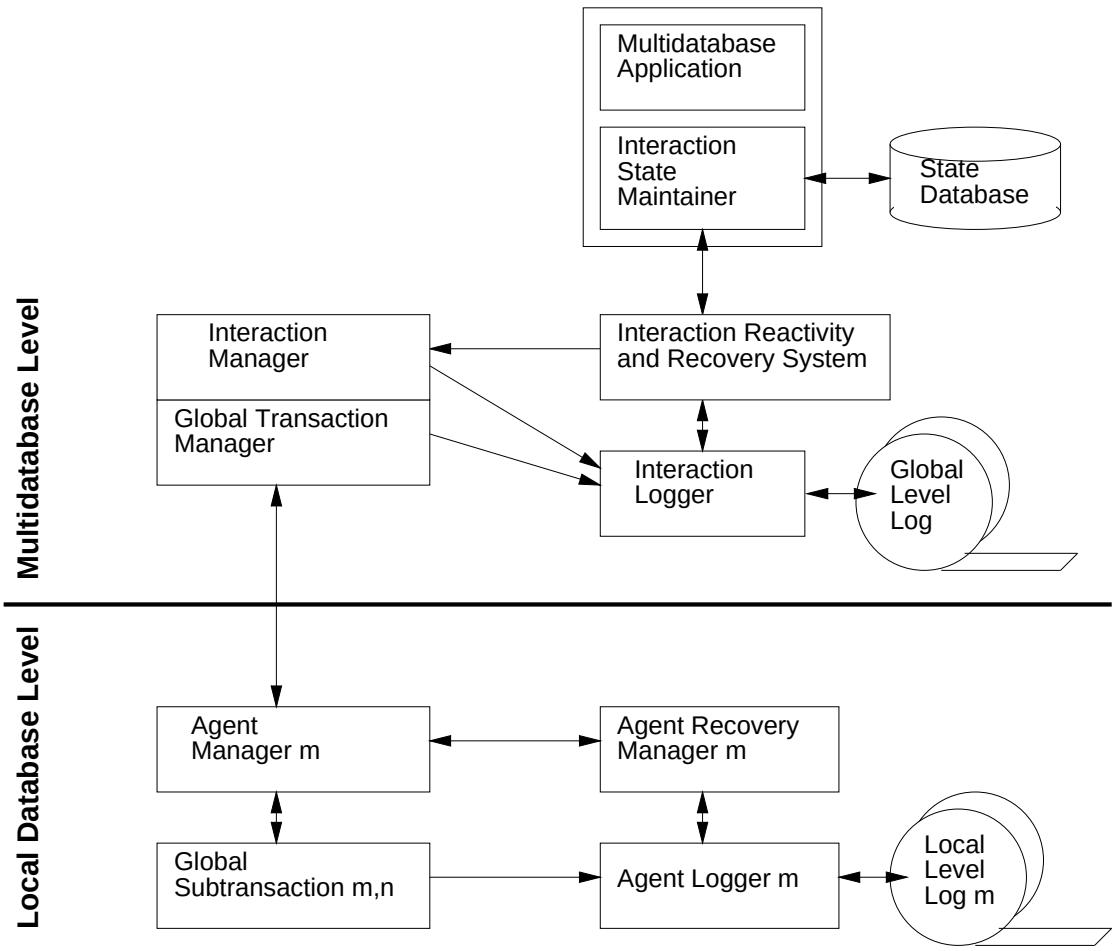


Figure 7.12: Modules used in logging, reactivity and recovery of Interactions.

2. The global transaction identifiers of the global transactions that participate in the Interaction.
3. The state and execution dependencies between the different global transactions in the Interaction.
4. Which events occur that invalidate parts of the Interaction, and when they occur in the execution.
5. When the Interaction terminates.
6. Whether the Interaction commits or aborts.

In addition, the global level log maintains the following information about each global transaction:

1. When the global transaction begins.
2. The local databases that participated in the global transaction.
3. For each local database, the log sequence number required to locate the records for its global subtransaction in the local log.
4. When the global transaction terminates.
5. Whether the global transaction commits or aborts.
6. Whether or not the global transaction has become invalid.

The information needed at the global level for reactivity and/or recovery includes information about each Interaction to be used during reactivity and/or recovery. The local databases maintain the real, definitive log used for failure recovery. The global level log (and the local level log) need only really maintain information required for reactivity, which is only needed for the lifetime of the Interaction. Therefore, each Interaction's global log entries can be garbage collected once the Interaction commits or aborts.

7.5.2 Local-Level Logging

The local level log contains information about the global subtransactions and their compensating transaction that is needed for reactivity and recovery. This includes the following information for each step called by each global subtransaction:

1. The arguments to the step.
2. The data items read and written by the step, along with the values read or written.
3. The results returned by the step.
4. Whether or not the step was compensating for some other step.
5. The name and arguments required to run the compensating step.
6. The projected read and write information for the compensating step, along with the projected values to read or write (if available).

7.5.3 Interaction Reactivity and Recovery Subsystem

The reactivity and recovery subsystem is invoked when a failure occurs or when an Interaction is partially rolled back as a result of some weak conflict violation. This subsystem operates in one of two modes. The *compensation mode* semantically undoes the Interaction, either partially or fully. This is the mode invoked when an Interaction is aborted, either by the user or because of some failure. The *replacement mode* partially or fully replaces the work done by the Interaction using the replacement algorithm. This is the mode invoked when a weak conflict is violated.

7.5.3.1 Compensation Mode

Following along in Figure 7.12, in compensation mode, the Interaction Reactivity and Recovery System (which handles both reactivity and recovery) reads the log to determine which of the committed global transactions need to be rolled back, and in what order. This can be reconstructed from the logged information from the global transactions and the Interaction's logged state and execution dependency information. The partial order of global transactions that need to be rolled back is called the *invalid portion*.

We know by the open nested recoverability property (Definition 4.2.1) that no uncommitted global transactions in the invalid portion have committed transactions with execution dependencies on them. Therefore, we begin the compensation process by aborting all uncommitted global transactions in the invalid portion. Once the uncommitted global transactions have all been aborted, the Interaction Reactivity and Recovery System then issues commands to execute compensating transactions for the remaining global transactions in the invalid portion.

Commands to run compensating global transactions are issued by the Interaction Reactivity and Recovery System in the inverse of the partial order that the original global transactions ran in. To run a compensating transaction, the Reactivity and Recovery System first tells the Interaction Manager to begin a new global transaction, with execution dependencies on the compensating transactions for the successors of its original transactions (or its original transaction, if there are no successors). Then the Reactivity and Recovery System parses the log record that was read for the original global transaction to get the list of local databases that participated in the global transaction and the logged local-level log sequence numbers for the subtransactions in the local-level logs. The Reactivity and Recovery System then issues a request to the Agent Managers of each of the local databases that participated in the original transaction to run a compensating subtransaction. Each request is forwarded to its Agent Recovery Manager via the new global transaction object in the Global Transaction Manager and the Agent Manager of the local database.

When the Agent Recovery Manager receives a command to run a compensating subtransaction, it first begins a new global subtransaction on the local database. Then, using the specified log sequence number, it reads the local-level log to get the sequence of compensating step calls for the (non-compensating) steps from the original global subtransaction. These compensating step calls are then issued to the compensating subtransaction's process in the inverse of the order of their corresponding original steps. However, less information needs to be logged for the compensating steps, because they themselves will never be compensated for.

Once all of the compensating steps have returned, the Agent Recovery Manager notifies the Global Transaction object for the compensating transaction that the subtransaction has finished its execution. When the Global Transaction object has received such notices from all of the local databases that participated in it, it enters the standard multidatabase two-phase commit (with the normal restrictions on commit order due to the need to maintain open nested recoverability), and terminates.

7.5.3.2 Replacement Mode

Replacement mode works similarly to compensation mode up to the point where the uncommitted global transactions have all been aborted. At this point, the Reactivity and Recovery System generates a projected compensation schedule for the remainder of the invalid portion, to use as a basis for replacement.

The first job of the replacement algorithm is to retrieve the projected read/write information, including the data items to be read or written and the values projected for those data items. This must be done for each compensating transaction in the projected schedule.

To retrieve the read/write information, the Reactivity and Recovery System iterates through each global transaction in the invalid portion, finding the local databases that it participated in and the log sequence numbers of the global subtransactions. It then sends requests down to the local databases' loggers, via the Interaction Manager, requesting the projected read/write information for each compensating subtransaction.

When the Reactivity and Recovery System has retrieved the projected read/write information and sorted it by global transaction, it can then begin the replacement process. The Reactivity and Recovery System follows the basic replacement algorithm defined in Section 6.6, generating either replacement transactions or compensating transactions as needed.

Compensating transactions are executed as they were in the compensation mode. Replacement transactions are a bit different, though. When the Reactivity and Recovery System begins a replacement transaction,

it first sends a request to the Interaction Manager to begin a global transaction. The elimination test issues some read operations to the local database to get the current values. If elimination fails, the replacement transaction is actually run. The compensating part of the transaction is run as with compensation mode, except the transaction is not committed. Then, the Reactivity and Recovery System hands control of the global transaction over to the Interaction State Maintainer for the Interaction, who can determine what to redo.

The job of the Interaction State Maintainer is to issue new steps to replace the compensated-for subtask. This is done by accessing the State Database associated with the application to get the original Interaction specification, and generating new steps as a part of the same replacement global transaction. These steps are executed normally, and logged as normal steps in the local-level log.

Once the Interaction State Maintainer completes the new steps, it passes control of the global transaction back to the Reactivity and Recovery System. The Reactivity and Recovery System then tests whether or not the replacement transaction view commutes to the front of the schedule, using the replacement transaction's actual read/write information requested from the local database's log. If the view commute succeeds, the Reactivity and Recovery System then enters into the standard two-phase commit. Otherwise it aborts the global transaction and does basic compensation following Algorithm 6.6.4 until it can resume the replacement process.

When all of the invalid steps have been compensated for (by steps in some compensating transaction, by steps in some replacement transaction, or by steps in some eliminated transaction) the Interaction is left in a state that is consistent both internally and with the multidatabase. However, the current execution state may be an earlier one than the one the Interaction was in when the weak conflict occurred. At this point, the Interaction State Maintainer for the Interaction needs to catch up by redoing the rest of the invalidated work.

7.6 Implementation

In this chapter, we gave an overview of the system design of our prototype multidatabase system, Mongrel. Mongrel provides the basic functionality to test the feasibility of Interactions as a multidatabase transaction model. This includes the following:

1. Basic support for atomic global transactions.
2. A set of local databases implemented over ObjectStore™, an object-oriented database, along with the support required to integrate them into the multidatabase.
3. Support for running Interactions, including a global-level database to store information on each Interaction, and a process to synchronize Interaction execution.
4. A shell that supports the definition of Interactions in TaSL.

The bulk of the code is written in C++ on Sun Sparc 10™ workstations running the UNIX™ operating system. The user interface portion of the TaSL shell is written using Motif™. Currently, the system implementation is about 57,400 lines of code. It supports basic reactivity using the compensation mode only.

There are very few multidatabase implementations, or even prototypes, in existence. A summary of existing multidatabase prototypes can be found in [ECB93].

Chapter 8

Conclusions

In this thesis, we have examined planning applications and their requirements on an underlying multidatabase. We defined a planning application as one that generates tasks that are long-lived, and that needs to respond to changes in the underlying database. We gave a variation of the standard travel agent example which demonstrates the needs of a planning application with respect to a multidatabase.

The user of a planning application defines a planning task to the multidatabase using an Interaction. Interactions follow a flexible, open nested transaction model. An Interaction is described as a partially-ordered set of global transaction specifications. The global transactions (or subsets thereof) can be combined in different ways to accomplish the planning task. Thus, unlike other flexible transaction systems, Interactions define their flexibility at the level of combining different sets of global transactions in their execution, rather than in allowing a flexible specification within a single global transaction.

An Interaction executes as a partial order of atomic, global transactions. Each global transaction coordinates the execution of a set of global subtransactions, one on each local database that it accesses. Each global subtransaction executes as a sequence of steps, as defined for the multidatabase in the local database's Step Library.

The principal new transaction concept defined for Interactions is *reactivity*. Reactivity allows the Interaction to respond and regain consistency when the underlying multidatabase evolves in a way that conflicts with the Interaction's work. Thus, in our model we have the option of allowing conflicting operations (rather than preventing them), and we also provide algorithms that are used within the multidatabase to recover the Interaction's consistency after the conflict. These algorithms use both the ability in our underlying implementation to automatically generate compensating transactions, and our ability to find a new scheme using the flexible definition.

Interactions use a notion of *weak conflict* to define their expectations on the state of relevant information in the database. If the database evolves in a way that violates an Interaction's weak conflict, then the Interaction is notified of the change and can react appropriately. For example, a person booked on a flight may need to know and respond to the fact that his flight has been canceled. A weak conflict is defined as an update in the database that causes some condition to be violated. When the event occurs, the Interaction Manager is notified, and can recover its consistency by identifying what is invalid and replacing that work.

We examine problems with using standard compensation to regain Interaction consistency when an event occurs in the multidatabase. We discuss a new scheme for recovering consistency called *replacement*. Replacement differs from previously proposed techniques, especially in the order semantic undoing and redoing is done and in the way steps are grouped together into atomic units. Replacement also takes advantage of the existing flexible Interaction definition and state to find automatically a new plan according to the specification given by the planning task. This new plan is both internally consistent and consistent with the current state of the database. It also differs from the original plan as little as possible.

The contributions of the replacement algorithm include:

- Undo and redo are executed as an atomic unit. In this way, the undo will not release a valuable resource that is needed in the redo.
- Steps in the current plan are always favored over other equivalent steps. This is accomplished by the

elimination process which avoids unnecessary work.

- It is not necessary to undo everything that the Interaction has done. In this way, we minimize the amount of work that must be redone.
- Redoing may begin early in the recovery process. It may begin as soon as a step is invalidated.

Because Interactions require as a basis the ability to support atomic global transactions on the multidatabase, we also explore in this thesis algorithms for enforcing a consistent global transaction serialization order across several types of local database. We discuss global atomic and semantically atomic commit. We develop a theory to describe when atomic commit (which blocks the local database operation) and semantically atomic commit (which blocks but not for the entire commit process) are possible.

The theory behind global commitment in a multidatabase is graph-based and independent of the specific commit protocol being emulated. A global transaction is represented as a graph with a single source and a single sink, whose nodes represent the global subtransactions and whose edges represent how information is transmitted from one subtransaction to another within the global transaction. Each global subtransaction is classified as (provisionally) compensatable and/or (provisionally) retriable, according to its syntactic structure. The theory enables us to decide the following things about when and whether a global transaction commit is possible:

1. If a specific subgraph (the *pivot subgraph's pivot)* contains more than one node, no atomic commitment is possible.
2. If a specific subgraph contains a cycle, no atomic commitment is possible.
3. If a specific subgraph (the *pivot subgraph*) contains at most one transaction, nonblocking semantically atomic commitment is possible. (see also [MRKS92]).
4. Given a committable global transaction, what is the minimal amount of blocking required?
5. Given a global subtransaction, at what point in the commit process can a global decision to commit be made and enforced?

We have defined and implemented a multidatabase system that supports Interactions as its access mechanism. It includes Interaction State Maintainers which store for each Interaction its entire definition, its execution state, and the state of its variables. The Interaction Manager coordinates the execution of the Interactions themselves, and calls upon the Global Transaction Manager to ensure that its global transactions are executed atomically.

Each local database has an Agent and an Activator. The Agent acts as an extension to the local database to provide the additional functions required by the multidatabase. The Activator monitors the local database for events of interest. The Activator and the Agent are both tailored to the design of the underlying database. It also has a Recovery Manager, which cooperates with the appropriate Interaction State Maintainer to execute the replacement algorithm when a weak conflict event occurs.

Each local database also defines its own *Step Library*, which is the interface that the multidatabase can use to access the information in the local database. The step approach differs from the two existing approaches for integrating local databases. It differs from the restricted approach in that each step is not itself a separate local database transaction. Rather, different steps on the same local database can be grouped into a single atomic global subtransaction. Steps on different local databases can be grouped into a single atomic global transaction. Thus, the step model is more flexible than the restricted model. The step model differs from the unrestricted model in that it does not allow arbitrary access to the local databases, only access through the defined steps. This restriction is useful because it provides a uniform way of accessing the multidatabase, independent of any local database's data manipulation language. It also allows the Step Library definer to associate compensating steps for each step, and to define how to derive the compensating step call during the execution of a step. During reactivity or recovery, these compensating step calls are used to generate compensating global transactions when needed for reactivity and recovery. The multidatabase user is freed from having to specify compensating subtransactions himself.

We have developed a language, TaSL, which planning applications can use to describe Interactions to the multidatabase. TaSL basically defines the control structure for executing *steps*. Steps are functions defined

on a single local database. Each step translates its function into local database DML, runs the ensuing queries and updates on the local database, and reformats and returns the result. Steps also derive and log the step-specific information needed to support compensation and reactivity during their execution.

TaSL allows steps to be grouped together by the Interaction definer into global transactions. These steps are ordered in the Interaction, similar to the way they would be ordered in any block-structured programming language. The definer also specifies alternative sets of global transactions that can be used to accomplish the Interaction's task. These are specified as prioritized alternative paths that can be taken from the "current" state of the Interaction execution.

Bibliography

[ANRS92] Mansoor Ansari, Linda Ness, Marek Rusinkiewicz, and Amit Sheth. Using flexible transactions to support multi-system telecommunications applications. In *Proceedings of the 18th VLDB Conference*, pages 65–76, 1992.

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

[BCEP93] O. A. Bukhres, J. Chen, A. K. Elmagarmid, and R. Pezzoli. InterBase: An execution environment for global applications over distributed, heterogeneous, and autonomous software systems. (to appear), August 1993.

[BGRS91] Yuri Breitbart, Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Abraham Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, September 1991.

[BGS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–239, October 1992.

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

[BÖH+91] Alejandro Buchmann, M. Tamer Özsu, Mark Hornick, Dimitrios Georgakopoulos, and Frank A. Manola. A transaction model for active distributed object systems. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications.* Morgan-Kaufmann, 1991.

[BS88] Yuri Breitbart and Avi Silberschatz. Multidatabase update issues. In *ACM SIGMOD*, pages 135–142, 1988.

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

[CBE93] Jiansan Chen, Omran Bukhres, and Ahmed K. Elmagarmid. IPL: A multidatabase transaction specification language. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, 1993.

[CR90] Panayiotis K. Chrysanthis and Krithi Rmamaritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *SIGMOD Proceedings*, pages 194–203, 1990.

[CR91a] Panos K. Chrysanthis and Krithi Ramamritham. ACTA: The SAGA continues. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications.* Morgan-Kaufmann, 1991.

[CR91b] Panos K. Chrysanthis and Krithi Ramamritham. A formalism for extended transaction models. In *VLDB 1991 Proceedings*, pages 103–112, 1991.

[Dav78] C. T. Davies, Jr. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

[DBC+88] U. Dayal, B. Blaustein, U. Chakravarthy, M. Hsu, R. Ladin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, March 1988.

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

[DEK91] Weimin Du, Ahmed K. Elmagarmid, and Won Kim. Maintaining quasi serilaizability in multidatabase systems. In *Proceedings of the 7th International Conference on Data Engineering*, pages 360–367, 1991.

[DHL90] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Organizing long-running activities with triggers and transactions. In *SIGMOD Proceedings*, 1990.

[DW91] Thomas L. Dean and Michael P. Wellman. *Planning and Control*. Morgan Kaufmann, 1991.

[ECB93] A. K. Elmagarmid, J. Chen, and O. Bukhres. A survey (taxonomy and analysis) of multidatabase systems. 1993.

[ED90] Ahmed K. Elmagarmid and Weimin Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of the 6th International Conference on Data Engineering*, pages 37–46, 1990.

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

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

[EM93] Ahmed K. Elmagarmid and James G. Mullen. Multidatabase atomic commitment protocols: A taxonomy and unified aproach. Technical Report CSD-TR-93-018, Purdue University Computer Sciences Department, March 1993.

[Fir89] R. James Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, 1989. (Also Technical Report YALEU/CSD/RR 672).

[Fis83] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Technical Report 273, Yale University Computer Science Department, June 1983.

[Geo87] Michael P. Georgeff. Plannning. Technical Note 418, SRI International, March 1987.

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

[GGK+91] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling long-running activities as nested sagas. *IEEE Data Engineering Bulletin*, 14(1):14–18, March 1991.

[GP86] Virgil Gligor and Radu Popescu-Zeletin. Transaction management in heterogeneous database management systems. *Information Systems*, 11(4):287–297, 1986.

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

[GRS91] Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Amit Sheth. On serializability of multidatabase transactions through forced local conflicts. In *1991 Data Engineering Proceedings*, pages 314–323, 1991.

[GRS93] Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Amit Sheth. Using tickets to enforce the serializability of multidatabase transactions. *IEEE Transactions on Knowledge and Data Engineering*, August 1993. (to appear).

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

[GSW87] Irene Greif, Robert Seliger, and William Weihl. A case study of CES: A distributed collaborative editing system implemented in Argus, 1987.

[HK86] Scott E. Hudson and Roger King. Cactis: A database system for specifying functionally-defined data. In *IEEE OODBS Workshop*, 1986.

[Kai90] Gail E. Kaiser. A flexible transaction model for software engineering. In *Proceedings of the 6th International Conference on Data Engineering*, pages 560–567, 1990.

[Kam90] Subbarao Kambhampati. A theory of plan modification. In *Proceedings of the 8th National Conference on Artificial Intelligence*, volume 1, pages 176–182, 1990.

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

[KRST92] Piotr Karychniak, Marek Rusinkiewicz, Amit Sheth, and Gomer Thomas. Bounding the effects of compensation under relaxed multi-level serializability. TM-TSV-021509/1, Bellcore, June 1992.

[LEB92] Yungho Leu, Ahmed K. Elmagarmid, and Noureddine Boudriga. Specification and execution of transactions for advanced database applications. *Informaton Systems*, 17(2), 1992.

[Leu91] Y. Leu. *Flexible Transaction Management in the InterBase Project.* PhD thesis, Purdue University, August 1991.

[LKS91a] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. An optimistic commit protocol for distributed transaction management. In *1991 ACM SIGMOD Proceedings*, pages 88–97, 1991.

[LKS91b] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. A theory of relaxed atomicity. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 95–109, 1991. (Extended Abstract).

[Lom92] David B. Lomet. MLR: A recovery method for multi-level systems. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 185–194, 1992.

[MJS] James G. Mullen, Jin Jing, and Jamshid Sharif-Askary. Reservation commitment and its use in multidatabase systems. (submitted to DEXA 1993).

[MKS92] James G. Mullen, Won Kim, and Jamshid Sharif-Askary. On the impossibility of atomic commitment in multidatabase systems. In *Proceedings of the Second International Conference on Systems Integration*, pages 625–634, 1992.

[Mos85] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, Massachusetts, 1985.

[MR91] Peter Muth and Thomas G. Rakow. Atomic commitment for integrated database systems. In *1991 Data Engineering Proceedings*, pages 296–304, 1991.

[MRB+92a] Sharad Mehrotra, Rajeev Rastogi, Yuri Breitbart, Henry F. Korth, and Avi Silberschatz. Ensuring transaction atomicity in multidatabase systems. In *PODS 1992 Proceedings*, pages 164–175, 1992.

[MRB+92b] Sharad Mehrotra, Rajeev Rastogi, Yuri Breitbart, Henry F. Korth, and Avi Silberschatz. The concurrency control problem in multidatabases: Characteristics and solutions. In *SIGMOD 1992 Proceedings*, pages 288–297, 1992.

[MRKS91] Sharad Mehrotra, Rajeev Rastogi, Henry F. Korth, and Abraham Silberschatz. Non-serializable executions in heterogeneous distributed database systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, 1991.

[MRKS92] Sharad Mehrotra, Rajeev Rastogi, Henry F. Korth, and Avi Silberschatz. A transaction model for multidatabase systems. Technical Report TR-92-14, University of Texas at Austin, March 1992.

[Nod91] Marian H. Nodine. Interactions: A non-serializable global transaction model for heterogeneous multidatabases. Technical Report CS-91-64, Brown University Department of Computer Science, December 1991.

[Nod93] Marian H. Nodine. Supporting long-running tasks on an evolving multidatabase using Interactions and events. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 125–132, 1993.

[PRR91] William Perrizo, Joseph Rajkumar, and Prabhu Ram. Hydro: A heterogeneous distributed database system. In *SIGMOD Proceedings*, pages 32–39, 1991.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

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

[Raz92a] Yoav Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource-managers. Technical Report DEC-TR 841, Digital Equipment Corporation, April 1992.

[Raz92b] Yoav Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers. In *VLDB Proceedings*, pages 292–312, 1992.

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

[SKS91a] Nandit Soparkar, Henry F. Korth, and Abraham Silberschatz. Failure-resistent transaction management in multidatabases. *IEEE Computer*, 24(12):28–36, December 1991.

[SKS91b] Nandit Soparkar, Henry F. Korth, and Abraham Silberschatz. Techniques for failure-resilient transaction management in multidatabases. Technical Report TR-91-10, University of Texas at Austin, April 1991.

[SKS92] Nandit Soparkar, Henry F. Korth, and Avi Silberschatz. Serializability among autonomous transaction managers. Technical Report TR-92-49, University of Texas at Austin, December 1992.

[SPSW90] Hans-Joerg Schek, Heinz-Bernhard Paul, Marc H. Scholl, and Gerhard Weikum. The DASDBS project: Objectives, experiences, and future prospects. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):25–43, March 1990.

[Wei91] Gerhard Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[WHBM90] Gerhard Weikum, Christof Hasse, Peter Broessler, and Peter Muth. Multi-level recovery. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 109–123, 1990.

[WR91] Helmut Waechter and Andreas Reuter. The ConTract model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan-Kauffman, 1991.

[WS91a] Gerhard Weikum and Hans-J Schek. Multi-level transactions and open nested transactions. *IEEE Data Engineering Bulletin*, 14(1):60–64, March 1991.

[WS91b] Gerhard Wiekum and Hans-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515–547. Morgan-Kaufmann, 1991.

[WV90] Antoni Wolski and Jari Veijalainen. 2PC agent method: Achieving serializability in presence of failures in a heterogeneous multidatabase. In *Proceedings of PARBASE*, pages 321–330, 1990.

[ZE93a] Aidong Zhang and Ahmed K. Elmagarmid. On global transaction scheduling criteria in multidatabase systems. In *Proceedings of the 2nd International Conference on Parallel and Distributed Systems*, pages 117–124, 1993.

[ZE93b] Aidong Zhang and Ahmed K. Elmagarmid. A theory of global concurrency control in multidatabase systems. *VLDB Journal*, 2(2), July 1993. (to appear).