Supporting Reactive Planning Tasks On An Evolving Multidatabase

Marian H. Nodine and Stanley B. Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-59
December 1992

Supporting Reactive Planning Tasks on an Evolving Multidatabase*

Marian H. Nodine and Stanley B. Zdonik
Brown University
Providence, RI 02912

Department of Computer Science
Technical Report CS-92-59
December, 1992

Abstract

In this paper we discuss our open nested multidatabase transaction model for multidatabases called *Interactions*. Interactions sit above a layer that provides atomic, serializable global multidatabase transactions. This model provides additional support for *planning applications*, which are long-running applications that may need to respond to changes in the underlying evolving multidatabase.

We begin by describing Interactions and the multidatabase architecture that they run on. We then discuss synchronization of Interactions. We describe the nature of *events*, which the applications can use to monitor changes in the information in the multidatabase.

When relevant events occur, we define how they are detected and how to respond to them. This response usually involves semantically undoing part of at least one Interaction, and then potentially re-executing the Interaction from that invalid point. We point out some problems with using standard compensation order (as defined in the Sagas paper [11]) when doing this type of recovery response. We then define a new type of recovery, called *replacement recovery*, that attempts to avoid these problems.

1 Introduction

A multidatabase is a collection of separate, autonomous local databases. A user of a multidatabase needs to access and/or update the information in these local databases in a consistent way to accomplish some common purpose, or *task*.

In this paper, we study how to support multidatabase applications that generate long-running tasks that need to react to relevant changes to the information that occur during their lifetime. These applications abound in the real world, including the widely-discussed travel agent applications as well as Artificial Intelligence applications for military and other operations. Because these applications are frequently associated with planning, we call them *planning applications*.

We define a planning application is one in which one or more users are concerned with accomplishing some well-defined goal. It is likely that the exact plan for how to accomplish this goal is not determined at the outset. Nevertheless, the goal can be stated clearly so that we know what needs to be done before we can successfully complete our task. For example, making travel plans for a business trip would be a planning

*This work was supported in part by ARPA order 8225, ONR grant N00014-91-J-4052, and in part by an IBM Graduate Fellowship.

task. At the outset, we know that we must be in a particular city for some number of days. We know that we need a hotel and some way to get around to the various meetings during our stay. We have 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. Furthermore, if we look at the structure of one of these activities, we will find that it typically consists of short activities separated by long periods of idle time. Because of their characteristics of longevity and burstiness, it is unreasonable to represent a planning task to the database as a serializable global transaction.

If a failure occurs, it is not always possible to roll back the result of an earlier activity. Consider the purchase of a non-refundable airline ticket. If the trip is canceled, we must either suffer the financial penalty associated with the ticket, or we could sell it ourselves. Either solution attempts to compensate for the earlier action.

When changes happen in the outside world that affect a planning task, the plan needs to be updated to take those changes into account. Thus, planning tasks must be written to be *reactive* to external changes. As they proceed, planning tasks accomplish subgoals. There are things about those subgoals that the planning task relies on as it proceeds. A planning task can explicitly specify these conditions along with steps to react to their retraction. This policy explicitly violates the classic transaction criterion of isolation.

In a reactive task, when some condition associated with some subgoal of a planning task is violated, the steps associated with that event are executed so the task can regain a consistent state. Our approach to regaining consistency is primarily through semantic undoing and working forward. By semantic undoing we roughly mean compensation, and by working forward we mean continuing the planning task from some consistent internal state.

Suppose that we achieve the subgoal of making an airline reservation as a part of our trip to San Francisco. We proceed with this assumption as we try to make a car reservation. If the flight is canceled, we must reconsider the car reservation. 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.

1.1 Travel Agent Example (With a Twist)

An example of a planning application is a travel agent application. 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 a single data repository, called the *multidatabase*. The multidatabase provides the travel agent with a global interface to this set of databases to use in booking trips for his customers.

In this example, let us examine one specific task that the travel agent needs to do. In this task, a travel agent is booking a single-day business trip for a customer, which requires booking a round-trip plane flight 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 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.

5. Notes the car rental information in the travel agent database.

6. Sends the itinerary to the customer.

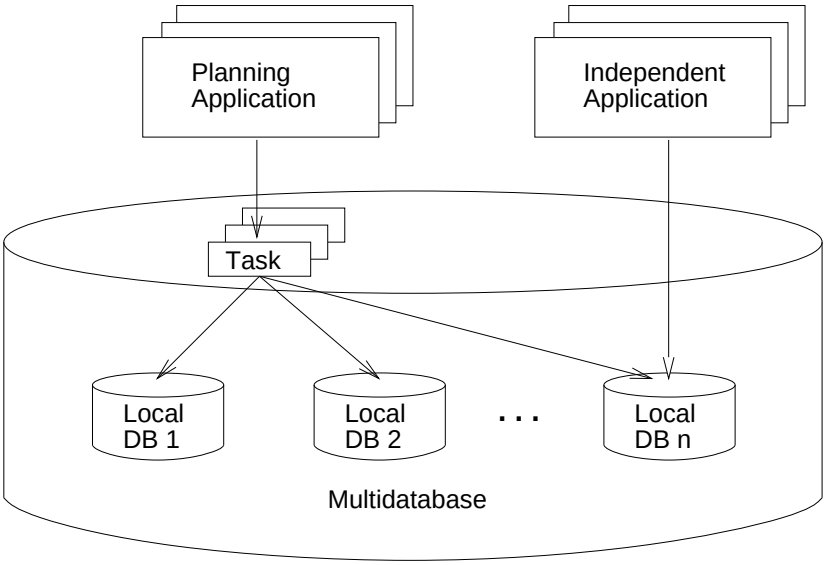


Figure 1: Multidatabase Environment

Later, the customer pays for the tickets and the travel agent issues them to the customer.

Now, provided nothing interferes, the trip should go smoothly. Everything is all set up for the customer. However, in this example, consider what happens if the airline cancels the flight 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 traveler so that he can *react* by replanning the invalid part of the task and making the plan consistent again. In this case, the traveler can try 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. Notifying the customer of the changes.
6. Getting a refund for the canceled flight.
7. Paying for the new flight.

If all of this is possible, then the trip can continue as planned. However, different things may go wrong; for instance, no alternative flight may be available.

1.2 A Working Environment

The basic multidatabase environment in which the planning application executes is shown in Figure 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.

This work makes certain assumptions concerning the multidatabase environment and the relationship between the local databases and the multidatabase. These assumptions include the following:

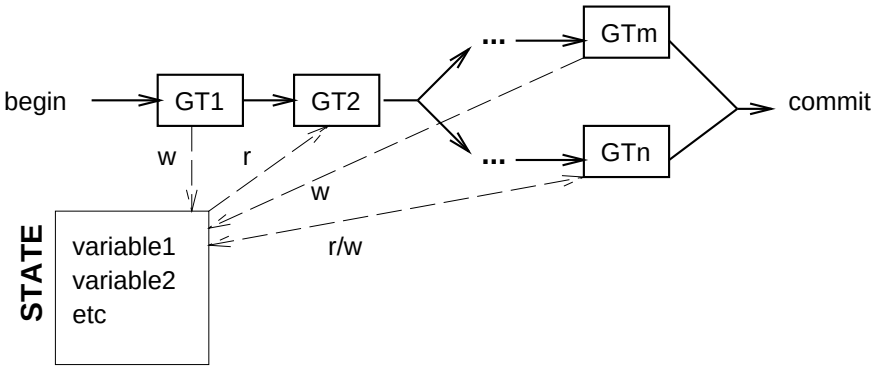


Figure 2: Basic structure of an Interaction.

- *Heterogeneity* – The different local databases may support different concurrency control protocols, data models and data manipulation languages.
- Access to the information in the multidatabase is provided through some *uniform interface*.
- *Design and execution autonomy* – No modifications need to be made to the local database functions or protocols to support the multidatabase.
- *Distribution* – The databases may be distributed in any way.
- *Flexible transaction model* – The function or structure of the tasks in the multidatabase may vary depending on the content or availability of information in the multidatabase.

In addition to these relatively standard multidatabase requirements, the multidatabase also supports *reactive* tasks – that is, tasks which need to be notified of interesting events in the database. While reactive tasks can be supported by active databases [5, 3, 14], 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. Also, when a constraint is violated, it is *the database's* responsibility to restore the consistency of its own data.

2 Interactions

An Interaction on a multidatabase is a partially-ordered set of atomic global transactions that share some state. In other words, Interactions follow a two-level open nested transaction model. The partial order of the global transactions ($<_I^T$) reflects the execution dependencies among the global transactions (i.e. places where one global transaction must serialize before a second) 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*.

2.1 Interaction Structure

The structure of an Interaction is shown in Figure 2. 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.

While we do not study in detail how to enforce the atomicity of the global transactions in this work, we do make some assumptions about them. Each global transaction consists of a set of *global subtransactions*, each of which executes on a single local database. No two global subtransactions of the same global transaction execute on the same local database.

Each global subtransaction is a sequence of *steps*. The Interaction programmer defines the Interactions in terms of these steps, and indicates how the steps are grouped into global transactions. The global subtransactions commit atomically with the global transaction commit.

2.2 Atomicity, Visibility and Conflict

The global transactions in an Interaction commit atomically once all of their steps have executed. In this sense, Interactions are like Sagas [11], in that this global transaction commitment makes its effects on the local databases visible both to other global transactions and Interactions as well as to independent transactions. 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. Weak conflicts are stability conditions on the values of individual data items 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.

Weak conflicts are used by a planning application to specify when a database should notify it 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 plans are made and their effects 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 respond to the weak conflict violation and 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 itself can go away.

2.3 Interaction Specification

In this section, we give a brief description of how Interactions are specified. An Interaction is programmed in terms of *actions*, that dictate what needs to be done to accomplish its task. These actions are grouped together into blocks that are executed together as atomic global transactions. As with Flex Transactions [8], the Interaction can specify different ways to accomplish the task, depending on the multidatabase state.

2.3.1 Actions, Global Transactions and Steps

An *action* is a partially ordered set of steps each of which accomplishes the same objective. Exactly one step must succeed for the action to succeed. The steps defined in the partial order are functionally equivalent, but may execute on different databases. The partial ordering of the steps indicates the preferences of the

task definer. The steps are attempted in some order consistent with the partial order defined by the action. If an action may be optional, the task definer can express *NULL* as the last alternative action.

A global transaction is a set of actions that must be executed as a single atomic unit in the multidatabase.

A *step* defines a cohesive unit of operations on a single local database. Each step is implemented as a procedure call on the Agent's *Step Library*.

2.3.2 Events

An *event* is something that happens in the multidatabase that influences the past execution of the Interaction. We define an event as an update operation on a specific data item that causes some condition in the database to be violated. The event is usually provoked by some other transaction or Interaction on the local database.

Events can be used in two ways in an Interaction definition. The first way is to delay the Interaction until the event has occurred. For example, the travel agent may wish to delay the termination of an Interaction until the traveler has completed his trip, or delay payment for a flight until a certain date. The second way events are useful is in noting when weak conflicts are violated.

2.3.3 Weak Conflicts

Weak conflicts indicate the conditions that should be preserved in the database during a specific execution span of an Interaction for its overall effects in the multidatabase to remain consistent (with respect to completing its own task). In addition, a weak conflict defines the steps needed to recover when its conditions are violated. In other words, weak conflicts do not indicate something to prevent, but rather when defined recovery measures must be taken.

Weak conflicts are expressed as an execution span over which an ECA-type rule is enforced. ECA rules [6] specify an event to monitor for, conditions that must be checked if the event occurs, and actions to take if the condition is violated. With Interactions, we define the event as an update operation on a specific data item in some local database. The condition in the rule is a condition that must hold on the value or existence of that data item after the update has been executed. The only variable allowed in the condition is the data item specified in the event.

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 invalidation of some committed action in the Interaction. Therefore, the action (event handler) defines at a minimum what part of the Interaction must be semantically undone.

2.4 Example

Figure 3 shows the process flow for the example task from Section 1.1, with the added caveat that the traveler may wish to rent a car and drive if he cannot get flights at the appropriate times. This flow diagram is represented as a modified finite state automaton (FSA). The transitions represent atomic global transactions. Figure 3 also shows an example of a weak conflict for the defined Interaction. The span during which the conflict holds is indicated by a dashed box. The label associated with the box specifies the conflict's ECA rule. In the figure, the weak conflict on the outgoing flight ensures that the reservation stays around until the traveler actually takes the flight.

3 Multidatabase Architecture

Figure 4 shows the architecture that we use to support planning tasks on a Multidatabase

Interactions are specified by the different multidatabase applications. The work done by an Interaction 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 ensure that these global subtransactions work together to do the work defined by the Interaction and to preserve each Interaction's isolation requirements, despite any activity by the independent transactions.

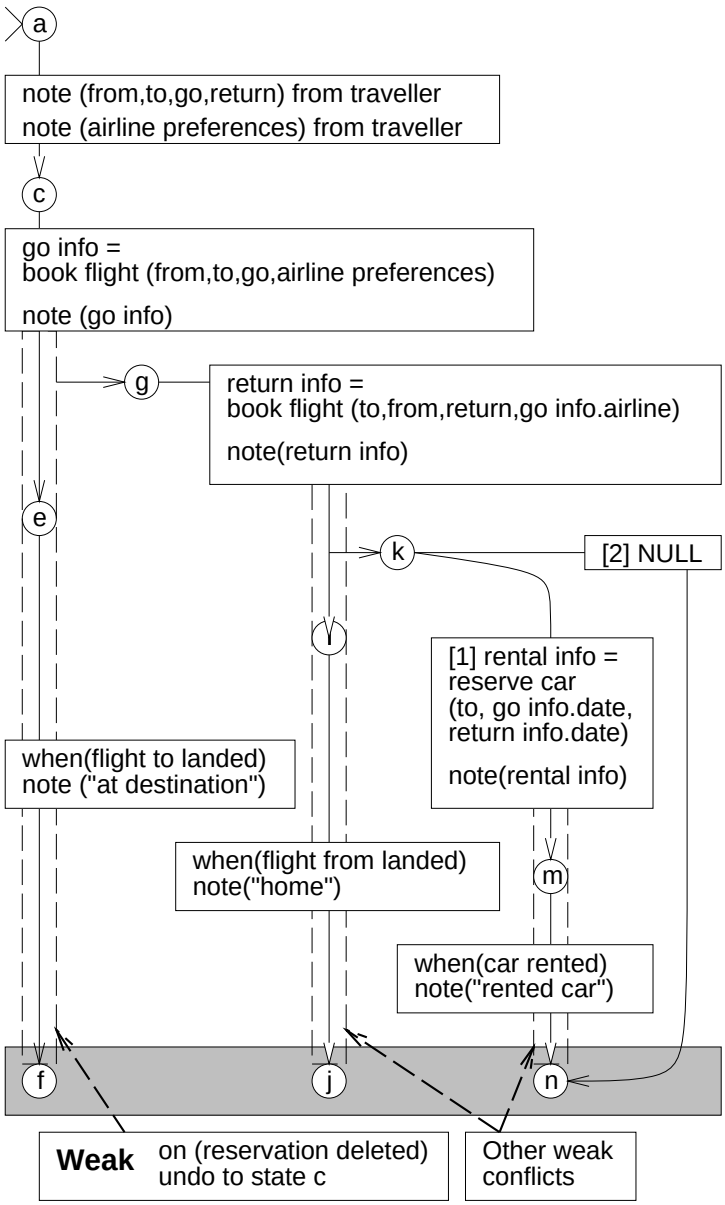


Figure 3: Task flow for a trip reservation. Example weak conflict is shown as a labeled dashed box. {f,j,n} is the final state set. Numbers in brackets indicate the priority of alternative actions.

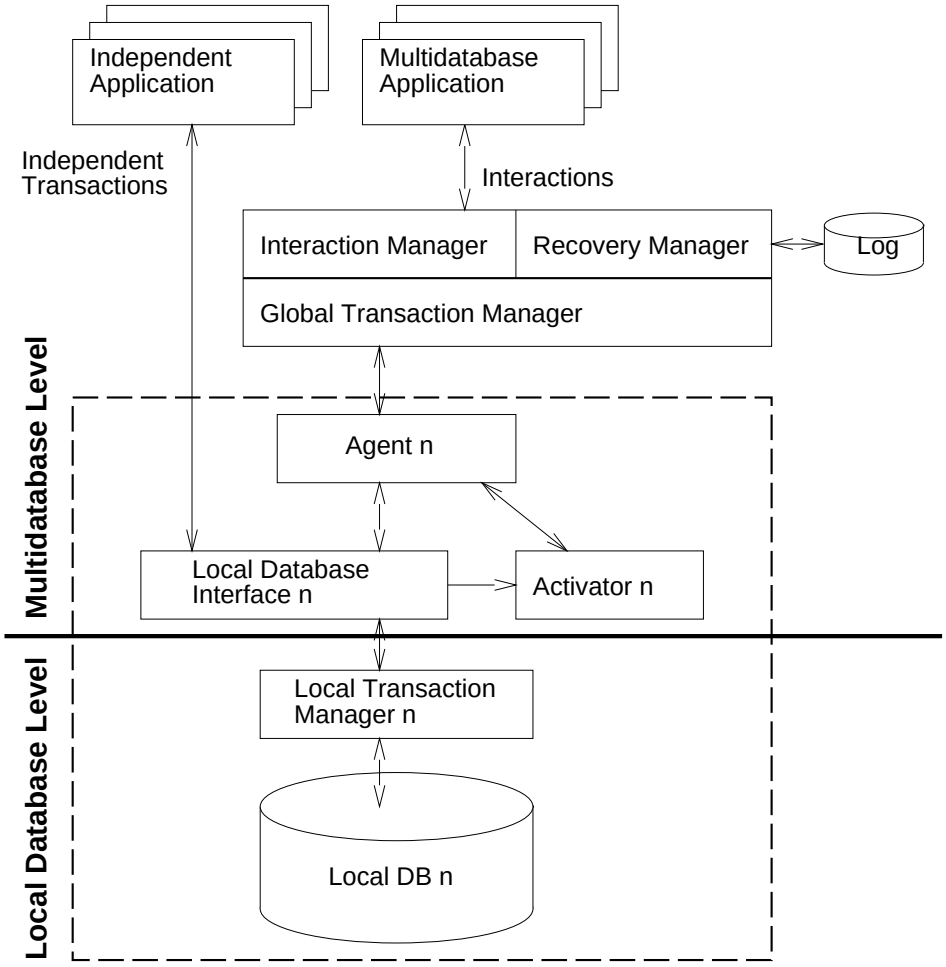


Figure 4: Multidatabase Architecture.

An Interaction is broken up into atomic *global transactions*, each of which encompasses several of the global subtransactions. Ensuring that the global transactions are executed atomically requires the coordination of these subtransactions at the global level. This work is done by the Global Transaction Manager at the request of the Interaction Manager.

One Agent is associated with each local database. Its primary job is to help the Global Transaction Manager ensure that the resulting history containing all global subtransactions and independent transactions is serializable. There are several known algorithms for this [23, 12, 17]. The Agent also maintains a Step Library that contains code to translate the steps of the different global subtransactions from the multidatabase's DML to the local database's DML. It then issues the translated subtransactions to the Local Database Interface, where they look just like the independent transactions issued by applications outside of the multidatabase.

The Agent's Local Database Interface co-opts the actual interface to the local database. It receives transactions in the local database's DML from both the Agent and from the independent transactions. Its sole purpose is to make a copy of those transaction command streams and pass them to the Activator.

The job of the Activator is to detect weak conflict violations. This involves detecting events (update operations) in the local database that are of interest to the active Interactions in the multidatabase, checking whether the associated conditions still hold, and informing the relevant Interactions when a condition is violated.

The Activator receives a specification of events and conditions to monitor for from the multidatabase application (via the Interaction Manager and the Agent). It also receives from the Local Database Interface a copy of the input stream to the local database that it partially parses 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 multidatabase application specifying the weak conflict that was violated. The multidatabase application can then tell the Recovery Manger what steps to take to recover.

The Recovery Manager maintains a log of information needed when an event causes some part of an Interaction to become invalid. The Recovery Manager also controls the recovery process when a semantic undo and re-execute are actually invoked.

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.

4 Interaction Synchronization

The Interaction Manager is responsible for ensuring that the execution of the Interactions in the multidatabase is correct. This section discusses how the Interaction Manager schedules the steps of the active Interactions, how it begins and ends the local transactions and sets up the Agent processes to ensure correctness, and how it deals with events in the database and their control over the timing of the execution of specific Interactions.

4.1 Ensuring Atomic Global Transactions

There have been many ideas proposed in the literature for enforcing atomicity (or at least, semantic atomicity) of global transactions in a multidatabase. For the purposes of this paper, we assume that the Global Transaction Manager dictates some serialization order for the global subtransactions, and this order is enforced using the forced local conflict scheme [12]. That is, a "ticket" is kept in each local database for use by the multidatabase. The ticket is successively incremented by each global subtransaction, with the increment order following the dictated serialization order. Other possible schemes include [24, 23, 17]. For atomic commitment, we follow the "local commitment before global decision" algorithm defined in [20]. Another commit scheme is defined in [16].

4.2 Scheduling Actions and Steps

At any specific time, the different in-process Interactions in the database have one or more concurrent threads executing the different tasks and subtasks. For scheduling purposes, the Interaction Manager 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 depth-first 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 action of some Interaction:

1. Choose the highest-priority step in the Interaction that has not yet failed. If none exists, place the Interaction in the *waiting* set, and allow the user to choose among the following:
   (a) Semantically undo the entire Interaction if all alternatives have been explored, and indicate to the user that it failed.
   (b) Semantically undo only the step that accomplished the "parent" action, and indicate that that failed. This causes alternative actions to those attempted at an earlier point to be explored.
   (c) Attempt a newly-specified alternative step.
2. If no global transaction is currently running, begin one.
3. Try the chosen step. 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. Go to step 1.
   (b) The step is not aborted, but the results are that the step failed to accomplish its subtask. 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.
4. If the step/action succeeds, then the results are returned and also logged in some stable location. If this is the last action defined for the current global transaction, commit it.

4.3 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 $T_c <_I^T T_d$.

When the new thread executes a new 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 5(a). In this situation, if the forking global transaction 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:

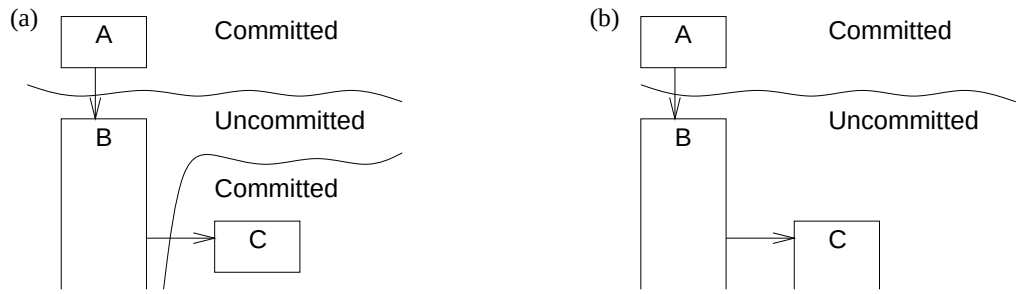


Figure 5: Various forms of current global transaction executions in an Interaction: (a) Uncommitted Global Transaction B has forked a second execution thread, and has a successor committed Global Transaction C. (b) Same Interaction execution, but in this case, Global Transaction C is restricted to not commit until Global Transaction B commits.

Definition 4.1 (Open Nested Recoverability Property) *For an Interaction to maintain the* open nested recoverability property*, no global transaction can commit until all global transactions that precede it in* $<_I^T$ *have committed.*

Given that this property is enforced by the Interaction Manager, situations such as the one in Figure 5(a) would be avoided, some delays in commit processing would occur, and the analogous executions such as the one in Figure 5(b) would be produced instead.

At any point in its execution, and Interaction thread may specify other threads to join with it. These joining threads must be threads that the thread has in the past (transitively) forked, thus forks and joins for an Interaction can be represented in a nested way as well. The joining threads must all have completed their execution before the join itself can be executed. Thus, at the time the join takes place, the joining threads must have no active global transactions.

5 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). It is the job of the Activator to detect when an event occurs, and to notify the Interaction Manager.

As an Interaction is executed, specifications of events of interest are sent to and deleted from the Activator. There are typically different purposes for these events, including:

1. *Wait events.* These are events on the local database that are explicitly being waited for (i.e., those specified in a *wait* command). Wait events are removed from the Activator's list of "interesting events" as soon as they occur.

2. *Conflict events.* These are events on the local database that indicate that a weak conflict has been violated (i.e., those specified in a current *on* statement). Conflict events are associated with *event handlers* (the *action* in the ECA rule), and these handlers are run by the Interaction Manager once the event is detected by the Activator. These events are removed explicitly from the Activator's list of "interesting events" once they are no longer relevant to the Interaction or when the Interaction terminates.

The Activator is responsible for noting when events on data items in its local database occur. It receives messages from the Interaction Manager via its Agent. This includes messages indicating events, and the times when these events are interesting to some Interaction. The conditions are optional. Each event is tagged as either a *wait event* or a *conflict event.*

The Activator maintains its own *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.

The Activator also receives an asynchronous copy of the actual input message stream to the local database. The messages in this stream are in the local database's DML. The Activator knows how to parse these messages to determine first, which ones update the local database, and second, what relations or sets in the database they update.

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 weak conflicts associated with the failed 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 can be done by the Agent as it processes the Activator's query.

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, and we disallow this as a tactic to use during recovery, this should not be a problem.

6 Recovery Problems

There are several problems with applying the compensation-based recovery methods as they have evolved over the years to the Interaction model (or any other flexible, open nested transaction model). First is that the constraint of "persistence of compensation" [16] cannot be guaranteed except in restricted environments. A second problem is that with the newer, flexible transaction models, exact compensation is certainly possible, but the re-execution may diverge radically and/or unacceptably from the original execution. Third, in open nested transaction models where each transaction maintains its own internal state, care must be taken to ensure that certain ordering constraints are maintained if concurrent compensating transactions are attempted. And last, from a semantic standpoint, compensation may cause work to be undone and redone unnecessarily.

In this section, we give examples of these problems and examine them in more detail. In the next section we outline the replacement recovery scheme, which avoids some of these pitfalls.

6.1 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 [11]. 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 if the open nested transaction is fully or partially rolled back.

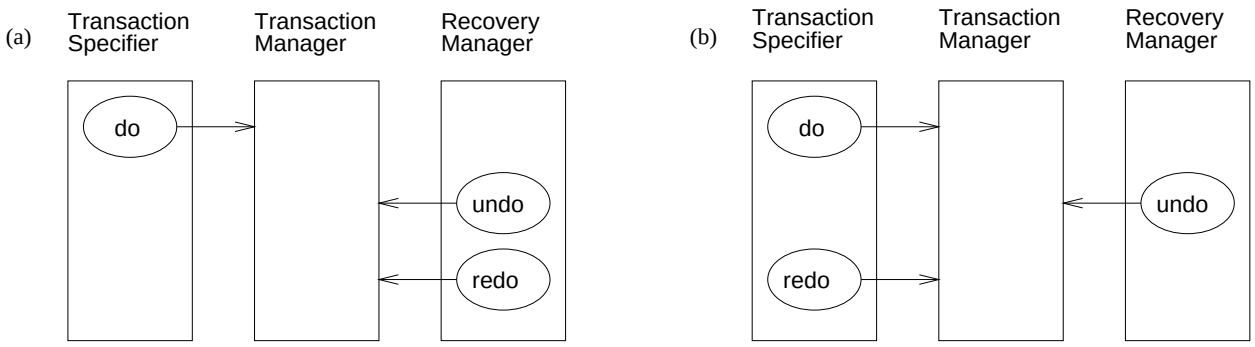


Figure 6: (a) Distribution of functions in traditional transaction and recovery management. (b) Distribution of functions when managing the execution and recovery of flexible transactions.

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.

In the early work on Sagas [11], a concept called *recovery blocks* was proposed. A recovery block provides an alternate way of compensating a particular action, under the assumption that the first method 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 subtransactions be specified as a prioritized set of guarded code blocks. The top priority code block exactly semantically undoes the original transaction. Lower-priority code blocks could take alternative, automated steps if certain conditions hold true. 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 to do less-than-optimal compensation steps we call "sloppy compensation".

6.2 Dealing with Flexible Transactions

Flexible transaction models such as [10, 8, 25, 22] 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.

With flexible transaction models, it is still possible to enable the database to determine compensating steps for the ones executed, and thus to automatically create compensating subtransactions as needed. The real problem occurs with the re-execution. If the original transaction failed due to some real problem, then the re-execution path will likely 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. Trying this alternative is a fundamentally different process from just retrying a set of steps that have already been executed once.

The solution to this problem is to give the specifier of the code for the transaction the responsibility to decide what to re-execute if and when a flexible transaction must be redone. 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(a).

With a flexible transaction model, the re-execution should choose among the different alternatives specified in the original transaction, using the same priorities. 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 specifier of the original transaction, which must save *all* of the alternatives originally specified, including

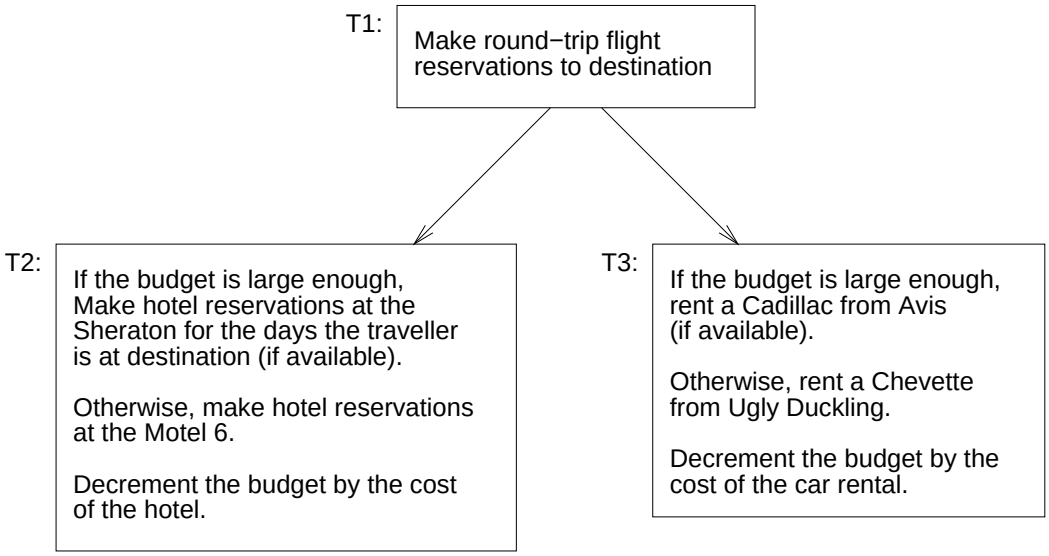


Figure 7: Does the traveler get a cheap hotel and an expensive car, or an expensive car and a cheap hotel?

the ones that originally failed, the one that did succeed, and the ones that were 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.

6.3 Managing Concurrency and State During Recovery

Allowing concurrent subtransactions to run within a single open nested transaction, and allowing open nested transactions to maintain internal state both create new problems in determining what to compensate at a given time, and in what order. Concurrency within an open nested transaction is usually specified explicitly using constructs equivalent to *fork* and *join* [11, 21]. 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) [25]. 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 open nested transaction shown in Figure 7. 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 open nested transaction must maintain enough information about its structure to be able to determine exactly what compensating subtransactions to issue and in what order during a partial semantic undo of the open nested transaction. This information includes maintaining 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.

6.4 Eliminating Unnecessary Work

During compensation work may unnecessarily be undone and redone, causing various undesirable scenarios. Consider again the open nested transaction from Figure 7. 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 section defines in detail the *replacement recovery* 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 recovery instead of using pure compensation.

7 Replacement Recovery Correctness

The basic idea behind replacement recovery is to try to *replace* any work in an open nested transaction that has become invalid without disturbing unrelated work. If we consider an open nested transaction 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.*

If we examine the flow of compensation-based recovery in the partial order of subtransactions that defines an open nested transaction, we see that recovery 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 recovery is always done in the original partial order. Starting with the invalid subtransaction, replacement recovery works *forward* in the order, replacing each function as needed. In the plane and hotel example from section 6.4, compensation-based recovery would compensate for the hotel, compensate for the plane, then re-execute some plane reservation subtransaction followed by some car rental subtransaction. Replacement recovery would compensate for the old plane reservation and make a new one, then if needed, it would compensate for the hotel reservation and make a new one, etc.

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.

7.1 View Commutation in a Projected Recovery History

In order to determine if a replacement history is correct, we first need to define some equivalence class 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 recovery, and affects how the recovery operations are scheduled. It also helps us to determine when certain recovery 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 7.1 *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 during compensation-based recovery.*
- *T' be the sequence of steps in T^C concatenated with the sequence of steps in T^R, executed atomically (the* replacement *transaction).*

Definition 7.2 *For any sequence of steps S, let*

- *$R_V(S)$ be the set of variables internal to the Interaction 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 variables internal to the Interaction 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 7.1 *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[1] over some projected subtransaction T_A.

Theorem 7.1 (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.*

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 7.1). 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 different values, causing T_A to read different values. However, as long as the values read by T_A remain the same, its outcome will not be affected, by Fact 7.1. 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 7.1.

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. □.

[1] 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.

7.2 Elimination of Work

In addition to commuting adjacent subtransactions in projected recovery histories, we also need to be able to determine when a subtransaction should not change the state of the database. *Elimination* during replacement recovery is the process of determining what of the subtransactions that are considered during recovery still stands, regardless of the fact that earlier subtransactions of the partial order have been replaced. For example, again using the plane and hotel example from Section 6.4, 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 recovery, the steps involved in re-executing a subtransaction T immediately follow the steps involved in compensating for T. Thus, the replacement subtransaction $T' = T^C|T^R$[2]. 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.

We base our test for elimination on the following theorem:

Theorem 7.2 (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 open nested transaction.

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 as T had. Similarly, the read-set of T^C is a subset of the read-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 7.1. 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.

[2] Here the | means that the steps of T^R immediately follow those of T^C.

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. These two sets of conditions, combined with the fact that $W(T) = W(T^C)$ combine to 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 re-execution paths are valid.

7.3 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 7.3 (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 7.4 *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 7.5 (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 the previous two sections.*

Definition 7.6 (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.*

8 The Replacement Algorithm

The replacement recovery algorithm attempts to replace as much of the work done by the invalid portion of an open nested transaction execution as is possible, while defaulting to compensation-based recovery when replacement is not possible. In the worst case, a schedule for replacement recovery is identical to the compensation schedule. In the best case, the schedule for replacement recovery 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 recovery 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. To ensure this, we restrict the commit order of the subtransactions to be consistent with the partial order of the open nested transaction; thus, no uncommitted subtransaction can have a committed subtransaction as its descendant in the partial order.

We also assume the presence of a log. The log contains the read- and write- sets (and their values) for each subtransaction. Also, we assume that the read- and write- sets (and some of the values they expect) can be deduced for the compensating subtransactions. They are logged as well.

8.1 Pseudocode

Algorithm 8.1 (Test Elimination) *Input: Read- and write-sets (and values) from T's execution. 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 $T^R = T$) 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 8.2 (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 8.3 (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 8.1. If so, do the following:*

   (a) *Using Algorithm 8.2, 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 process that originally specified the Interaction (the* transaction specifier*).*

   (b) *The transaction specifier 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 8.2, 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 open nested transaction to continue its execution from the current state.*

Algorithm 8.4 (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 recovery algorithm falls into compensation recovery when replacement fails. Compensation recovery (including, where necessary, sloppy compensation) always succeeds. Therefore, replacement recovery should always succeed as well.

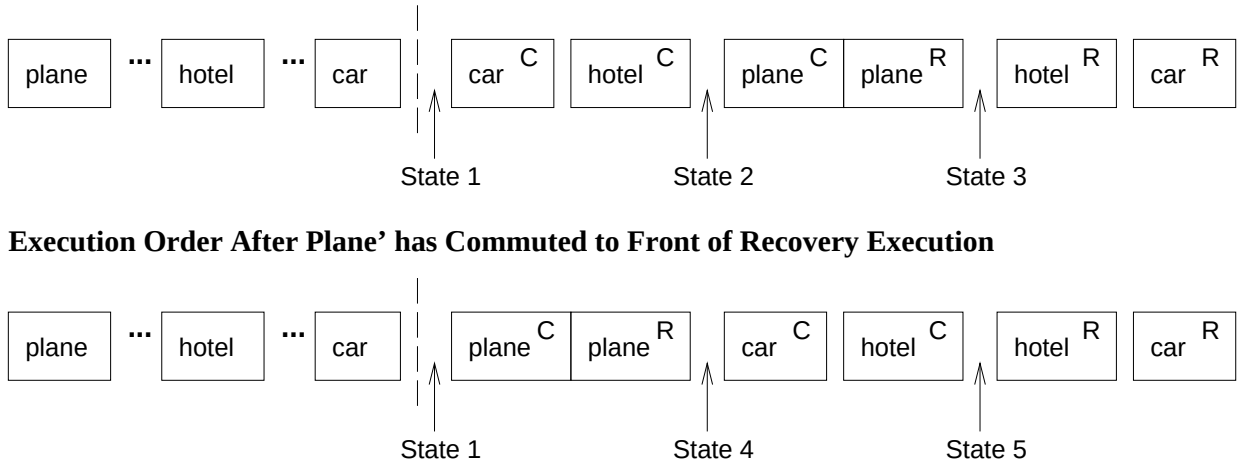


Figure 8: Replacement recovery and state.

8.2 Context Management

In commutation-based recovery, the recovery starts with the context and database in the state at the time that the abort or failure occurs. At each compensation step, the database and the internal variables are carefully backed up to a state that is semantically equivalent to the state that they were in just before the original global transaction was executed. Then, when the compensation phase is complete, the re-execution proceeds smoothly forward from a state that has all the effects of the original global transactions semantically removed.

When replacement is attempted, context management is not so obvious. The steps that replace the functions of the original global transactions are executed from a database and internal variable state that is not consistent with respect to some valid partial execution of the open nested transaction. To really ensure correctness, we need to show that the effects of the remaining compensating global transactions do not affect the execution of the current global transaction being replaced.

Let us examine the execution shown in Figure 8. Here, we have an open nested transaction that is to be compensated for and then redone. With replacement-based recovery, we try to commute $plane'$ over all of the compensating steps $car^C, \cdots, hotel^C$. We examine the case where the commutation succeeds. What we need to show is that state 1 is equivalent to state 2 with respect to the execution of $plane'$. Also, we need to show that state 4 is equivalent to state 1 with respect to the execution of $car^C, \cdots, hotel^C$. Then we need to show that state 3 is equivalent to state 5, regardless of whether $plane'$ is executed first or last.

We can divide up the objects and internal variables touched by $plane'$ and $car^C, \cdots, hotel^C$ as follows:

1. Items that will only be read by any of those global transactions.
2. Items written by $car^C, \cdots, hotel^C$, but not read by $plane'$.
3. Items written by $plane'$, but not read by $car^C, \cdots, hotel^C$.
4. Items written by $plane'$ and read by $car^C, \cdots, hotel^C$. We know that $car^C, \cdots, hotel^C$ reads the same values regardless of which of the two execution orders is chosen, because $plane'$ backwards commuted over $car^C, \cdots, hotel^C$. Therefore, we can treat these as if they are in the first case.

No other cases exist, because otherwise $plane'$ would not have backwards commuted over $car^C, \cdots, hotel^C$.

We can see from this list that (effectively) $plane'$ and $car^C, \cdots, hotel^C$ never read from each other; they only overlap in the sets of database objects and internal variables that they access is never written by either. Therefore, state 2 is equivalent to state 1 from the point of view of $plane'$, and state 4 is equivalent to state 1 from the point of view of $car^C, \cdots, hotel^C$.

Now let us compare states 3 and 5, to see whether they differ in how they might affect the re-execution (here represented as $hotel^R, \cdots$ though the actual execution may diverge from this). Because they commuted, we know that if $plane'$ and $car^C, \cdots, hotel^C$ overwrite each other, they both write the same value. Therefore the database state after both of them are executed is not dependent on their execution order, and the value of any variable or database object that $hotel^R, \cdots$ reads from either is the same in either state.

We see then that we do not need to worry about context when we reorder $plane'$ with respect to $car^C, \cdots, hotel^C$. Therefore, replacement recovery can begin with the current context, and will generate the same correct recovered state that would be reached using compensation-based recovery.

9 Examples

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

9.1 Plane, Hotel, and Car Example

Let us examine how the situation described in Section 1.1 would manifest itself in the multidatabase, and how it would be handled. In this example, assume that both flights and the car rental have been booked, but the traveler has not left yet. Thus, in the process flow diagram of Figure 3, the current state of the Interaction is represented by the state set $\{e, i, m\}$. Furthermore, the weak conflicts on the existence of the plane and car reservations are still 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 Manager of the violation of the weak conflict.

5. The Interaction Manger notifies the multidatabase application that the weak conflict has been violated.

6. The multidatabase application responds by telling the Recovery Manager associated with the Interaction Manager to roll back the Interaction through the plane reservation and re-execute it.

7. 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 multidatabase application books a new flight and gets new go_info. 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.

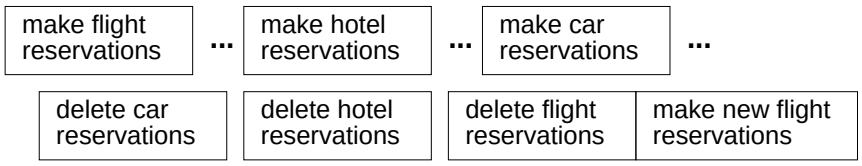
8. The Recovery Manager tries to view commute the replacement transaction for the flight to the front of the projected history. The view commute succeeds because the flight dates did not change. The Recovery Manager then commits the replacement transaction.

9. Because the new flight is on the same date as the canceled flight, the replacement transactions for the rental car and the hotel can be eliminated from the recovery execution. So the interaction's state once again reaches the state set $\{e, i, m\}$.

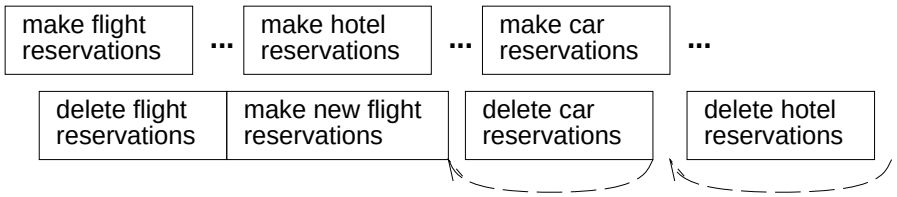
This entire process is shown in Figure 9.

Eventually, the customer takes the outgoing flight. At this point, the Interaction Manager should see that the flight has been taken, and continue executing the Interaction as appropriate. This is done as follows:

(a) Project the beginning of the execution with compensation−based recovery



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

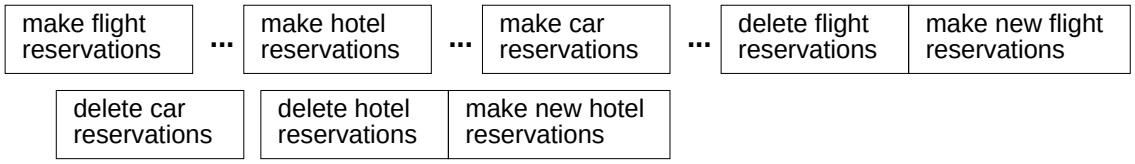


(c) Try to eliminate flight reservation changes.

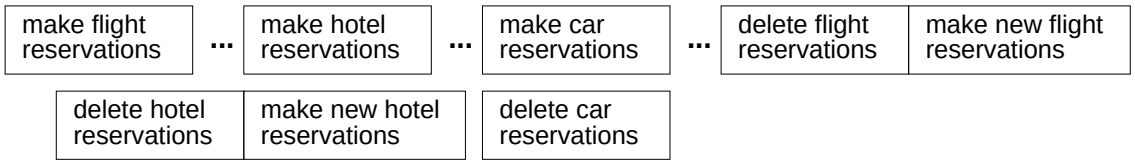
This does not work because the original flight reservations were cancelled.

(d) Execute the flight reservation changes.

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

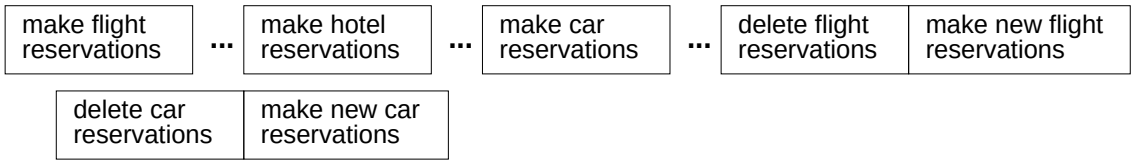


(e) Commute the hotel reservation changes to the beginning of the projected execution (if possible)



(f) Try to eliminate hotel reservation changes.

This works because the travel dates did not change



et cetera

Figure 9: The replacement recovery process for the plane, hotel, and car example. In each stage, the end of the current history is shown on the top row, and the projected execution is on the row underneath.

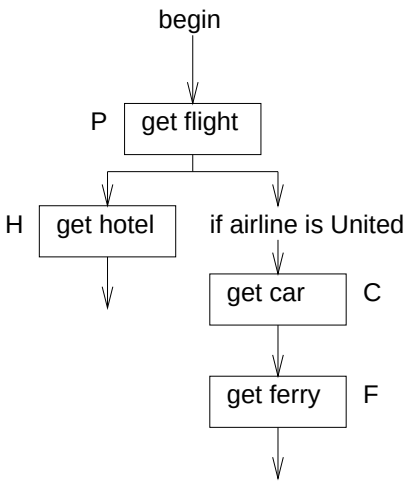


Figure 10: Example: Frequent flyer miles give traveler a free car rental, requiring ferry reservations to be made for the rental car.

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 Manager. The Activator then removes this event from the event table.
4. The Interaction Manager receives the notice, and continues executing the appropriate Interaction thread. This causes the travel agent's database to be updated. In the process flow diagram, there is a state transition from state e to state f, making the current state set $\{f, i, m\}$.
5. The Interaction Manager cancels the weak conflict on the outgoing flight reservation.

9.2 Ferry Example

A second example is shown in Figure 10. 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.

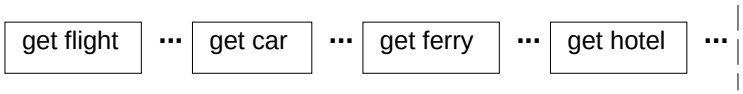
The original execution dependencies are reflected in the arrows between the atomic blocks in the figure. The "get hotel" block is dependent on the airline flight dates. The "get car" block is dependent on both the flight dates and the airline. The "get ferry" block is dependent on the existence of the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

In this example, the initial history before the airline flight is canceled is shown in Figure 11(a). Figure 11(b) shows the projected compensation-based recovery execution; the actual recovery execution should be equivalent to this one.

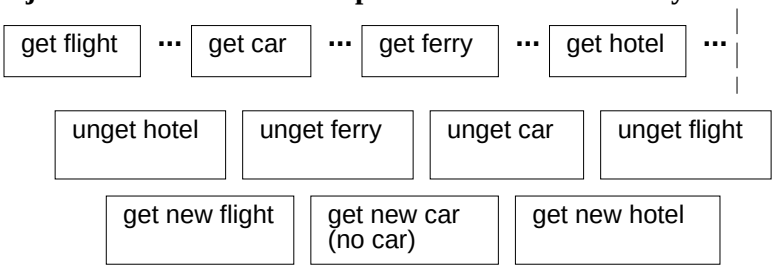
At the point in the scheduling shown in Figure 11(c), the flight has been successfully replaced by another flight on a different airline, and replacing the rental car is considered. However, since not enough frequent flyer miles are available for the free rental car (because of the change of airline), 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 11(d), the ferry needs to be dealt with. The compensating block for the initial ferry reservation commutes over the hotel reservation, since the two have nothing to do with each

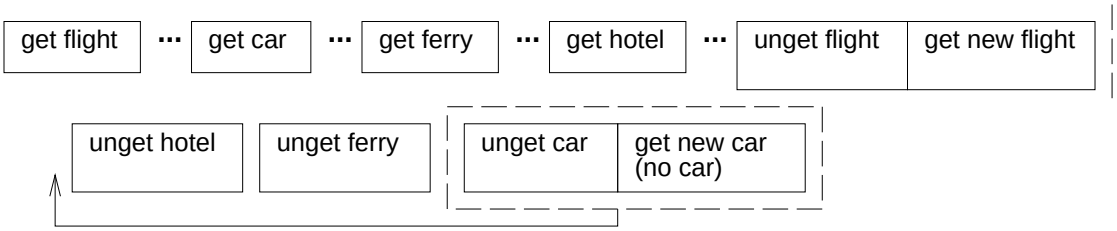
(a) Original history (before flight is cancelled)



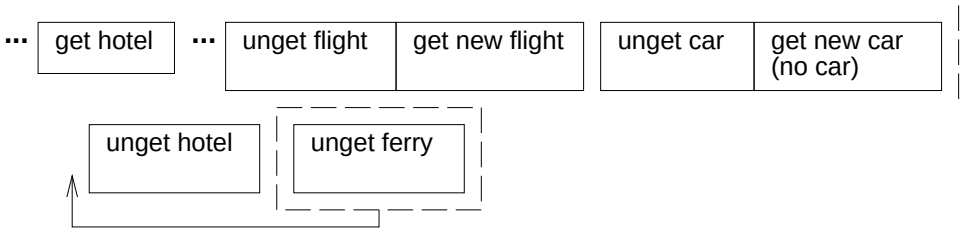
(b) Projected execution with compensation−based recovery



(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

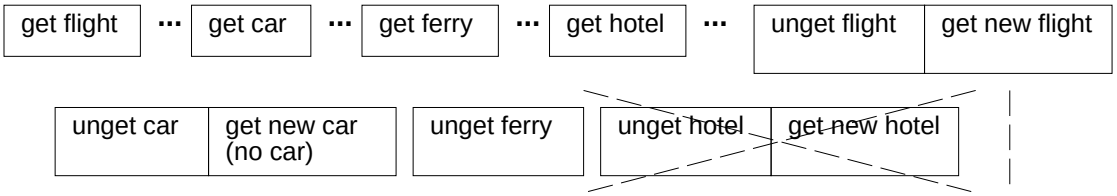


Figure 11: Steps in scheduling recovery for example.

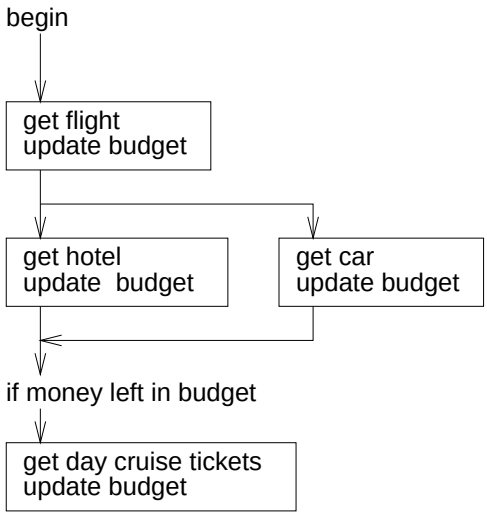


Figure 12: Example: Keeping a trip within budget causes recovery to degrade to a totally compensation-based strategy.

other. Then, just after this, the recovery process notes that the hotel blocks would not change anything, and thus they can be eliminated from the recovery history. The final history after recovery completes is shown in Figure 11(e).

9.3 Budget Example

Another example where recovery is constrained by data access is shown in Figure 12. In this figure, a careful budget is being kept, and the trip must cost less than the budget. In this situation, even though the block to get the hotel and the block 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 during recovery. This is because every original block, every compensating block, and every re-executed block read and change the budget variable. Thus, each such block must read from the previous such block. Therefore, no block executed during recovery can commute over the previous block, and the replacement recovery order deteriorates to be identical with the compensation-based recovery 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.

10 Related Research

10.1 (Almost-) Serializable Models

This work assumes an underlying multidatabase transaction manager that enforces the atomicity or semantic atomicity of global transactions on the multidatabase. This problem is currently receiving much scrutiny. Gligor et.al. [13] were the first to really explore issues concerning multidatabase transaction serializability, and they came up with a list of requirements that need to be met by the global transaction manager. A thorough survey of the approaches to these problems can be found in [2]. Specific work includes [20, 16, 24, 23, 12, 17]. Elmagarmid and Du [9] explore how different local concurrency control models affect their serialization order.

Schemes that do not necessarily enforce serializable global transaction executions, but try to approximate it, include *quasi-serializability* [7] (also called multidatabase serializability [1]) and *two-level serializability (2LSR)* [18]. Both QSR and 2LSR are easy to enforce in a multidatabase, but they need supplementary restrictions on transaction structure to ensure consistent execution at the global level.

10.2 Open Nested Models

Open nested transactions differ from standard nested transactions [19] in that the children of a parent operate under different spheres of control [4]. Unlike multilevel transactions [26], open nested transactions are not restricted to conform to a consistent number of levels. Because Interactions are defined for a multidatabase environment, we restrict our nesting to two levels, although the model does not preclude extension to deeper levels of nesting.

An early open nested transaction model was Sagas [11]. A Saga is a sequence of independent atomic transactions. This work was extended to the nested Saga model [10]. In a nested Saga, a child can be either another Saga or a transaction.

Nested Sagas, ConTracts [25], and Flex transactions [8] extend the open nested transaction model to allow flexibility in execution depending on the database state. With flexibility, different ways of achieving the same goals are specified. Which way is actually taken depends on the state of the underlying database at the time the actual steps of the transaction are attempted. Some types of flexibility that are supported include:

- 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., rent a car if one is available.
- Allow divergent sets of steps; e.g. either book a flight and rent a car at the destination, or rent a car at the source and drive the whole way.

The open nested transaction model presented in ConTracts is very similar to the Interaction model. However, while Interactions are responsible for their own consistency, ConTracts place the responsibility of dealing with conflicting operations in the database itself.

One other transaction model that incorporates open nesting is the extended nested transaction model from [6]. This model assumes an underlying database that is active.

10.3 Recovery Approaches

Our work differs from other work in open nested transactions because of the types of failure we support. All work, including ours, needs to deal with open nested transaction failures when a system failure occurs. Also, all work on aborting deals 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 abort.

All of the models described in the previous section use a form of recovery called *compensation*. This form of recovery was first defined in detail by Garcia-Molina and Salem [11]. 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 [10] require that the compensating Sagas be invoked recursively.

The ConTracts work [25] 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. In this work, we have taken this one step further, as *view commutation* defines when this concurrency can occur.

Because open nested transactions make partial results visible early, recovery of one transaction can cascade to others. ConTracts address this problem by explicitly maintaining all dependencies. However, this maintenance is not always possible. For example, dependencies cannot be explicitly computed in a multidatabase.

Korth et.al. [15] describe 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.

Since our open nested transaction model is *reactive*, in addition to failure recovery and complete transaction abort we support conflict- or event-driven partial rollback of committed portions of an open nested transaction when a conflicting operation has occurred. We do not know of any other work that examines this form of recovery.

11 Conclusions

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 an open nested transaction model. They also 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 respond appropriately. For example, a person booked on a flight may need to know and respond to the fact that his flight has been canceled.

An Interaction defines a weak conflict as an update in the database that causes some condition to be violated. When the event occurs, the Interaction Manager is notified, and applies the user-defined *event handler* to recover from the problem.

We define a multidatabase architecture that uses Interactions as its access mechanism. It includes a centralized Interaction Manager that coordinates the execution of the Interactions themselves. 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.

We examine problems with implementing compensation-based recovery when an event is detected by the Activator. We discuss a new compensation-based recovery scheme called *replacement recovery* that addresses some of these problems. This scheme differs in many significant respects 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.

The contributions of replacement recovery 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.

We are currently implementing a prototype of this model, called Mongrel.

References

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

[2] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–239, October 1992.

[3] 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.

[4] Jr. C. T. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

[5] U. Dayal, B. Blaustein, U. Chakravarthy, M. Hsu, R. Ledin, 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.

[6] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Organizing long-running activities with triggers and transactions. In *SIGMOD Proceedings*, 1990.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] Dimitrios Georgakapoulos, Marek Rusinkiewicz, and Amit Sheth. On serializability of multidatabase transactions through forced local conflicts. In *1991 Data Engineering Proceedings*, pages 314–323, 1991.

[13] Virgil Gligor and Radu Popescu-Zeletin. Transaction management in heterogeneous database management systems. *Information Systems*, 11(4):287–297, 1986.

[14] Scott E. Hudson and Roger King. Cactis: A database system for specifying functionally-defined data. In *IEEE OODBS Workshop*, 1986.

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

[16] 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.

[17] 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.

[18] 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.

[19] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.

[20] Peter Muth and Thomas G. Rakow. Atomic commitment for integrated database systems. In *1991 Data Engineering Proceedings*, pages 296–304, 1991.

[21] 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.

[22] 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*, 1993. to appear.

[23] William Perrizo, Joseph Rajkumar, and Prabhu Ram. Hydro: A heterogeneous distributed database system. In *SIGMOD Proceedings*, pages 32–39, 1991.

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

[25] Helmut Waechter and Andreas Reuter. The ConTract model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan-Kauffman, 1991.

[26] 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.