InterActions: Multidatabase Support for Planning Applications

Marian H. Nodine

Technical Report No. CS–91–64
Dec ember, 1991

InterActions: Multidatabase Support for Planning Applications

Marian H. Nodine
Brown University
Providence, RI 02912

Department of Computer Science
Technical Report CS–91–64
December, 1991

Abstract

Multidatabases attempt to provide a unified method of accessing information currently stored in multiple, autonomous, existing databases. This paper describes a global transaction model, *InterActions*, that supports access to a multidatabase. An InterAction is a unit of work that accomplishes a specific, coherent task. Unlike traditional transactions, an InterAction is not assumed to be atomic. Instead, it allows the user to define for the InterAction not only the steps that it takes to accomplish its task, but also the extent to which the InterAction must be isolated from other InterActions. It also allows the user to define what to do when the underlying multidatabase evolves in some way that affects the InterAction.

This paper presents a set of requirements for tasks that access and/or update information in a multidatabase. A model for defining these tasks, called InterActions, is presented. This includes specifying a language for defining InterActions, and shows how to map an InterAction specification onto a set of transactions on the local databases that form the multidatabase. In addition, it presents a definition for correct operation of a multidatabase that supports InterActions.

1 Introduction

A multidatabase provides a global, uniform interface to 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 order to accomplish some common purpose, or *task*. As with a specific, single database, the multidatabase user also requires certain guarantees to be made about the effects of his specific task, including how those effects are reflected in the information stored in the local databases, and how well those effects persist in the event of system, media, and/or process failure.

1.1 An Example (The Travel Agent Example)

An example of a multidatabase 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. The multidatabase provides the travel agent with a global interface to this set of databases that he can 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, he uses the customer's preferences to determine which airlines to try to book first.
3. Notes in the travel agent database what needs to be paid by the customer, and by what date.
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 in the travel agent database the results of the car rental.
6. Notifies the customer about the specifics of the flight and the car rental.

Later, the customer pays for the tickets. At this point, the travel agent issues the tickets 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, before the day of the trip, the airline cancels the flight. In this situation, it is not necessarily desirable to abort the trip plan and start all over. Rather, the travel agent should be told that the flight is no longer available, and then should try to book another flight. 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 payment of 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 that day.

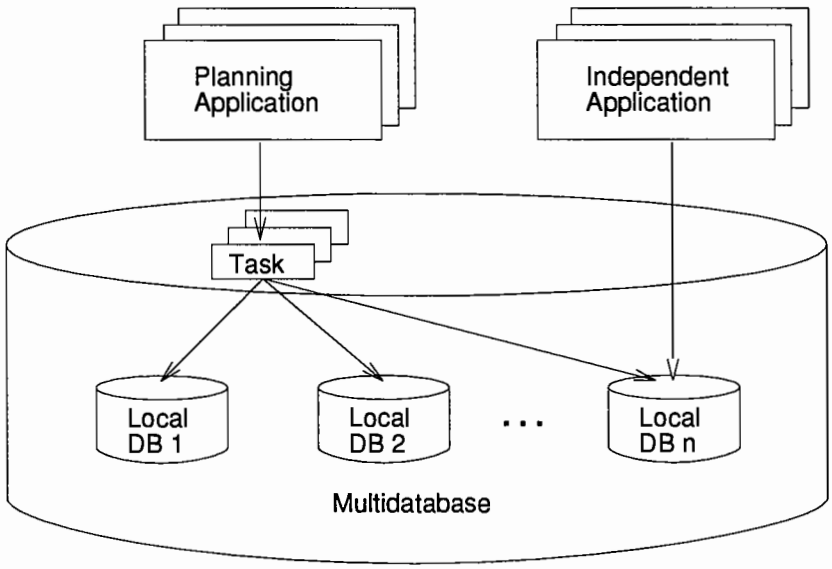


Figure 1: Multidatabase Environment

1.2 A Working Environment

The basic multidatabase environment in which the travel agent 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:

- The different local databases may support different concurrency control protocols, data models and data manipulation languages *(heterogeneity)*.
- Access to the information in the local databases is provided by the multidatabase through some *uniform interface*.
- No modifications need to be made to the local database functions or protocols to support the multidatabase *(autonomy)*.
- The databases may be distributed in any way *(distribution)*.
- The function or structure of the tasks in the multidatabase is not restricted in any way; e.g. read-only.

A detailed description of these assumptions and their justification is given in Section 2.

1.3 A Task Definition

Different tasks that execute in a multidatabase may affect different objects in the local databases. Tasks may interact with each other, usually because they deal with the same object(s). These interactions happen naturally, and may or may not interfere significantly with the involved tasks' progress. Additionally, independent applications may execute transactions that interfere with the progress of some task. In the travel agent example, the airline initiated a transaction on the local database (canceling the flight) that interfered with the travel agent's task (booking the trip) because they both interacted with the same object (the customer's reservation for the flight). This interference was desirable from the point of view of the airline, because there was probably a very valid reason for the cancellation of the flight. However, it is also desirable from the point of view of the travel agent. Admittedly, it interfered with the progress made so

1. A task has its own unique identitiy.
2. A task has a clearly defined procedure.
3. A task operates on a multidatabase that evolves over time.
4. A task specifies its own correctness conditions:
    a. What must be done, and in what order. (pattern)
    b. What parts must be executed in isolation. (strong conflict)
    c. What to do when events interfere with its progress. (weak conflict)
5. A task should not be left partially done.

Figure 2: Task Characteristics

far in giving the customer a valid flight reservation for his trip. However, provided that the travel agent is watching for such a development, he can recover from the interference before the customer actually arrives at the airport to take the canceled flight.

When a task is done, the person doing the task is responsible for ensuring (as much as possible) that other tasks do not interfere with the accomplishment of its objective. In this case, the travel agent is responsible for ensuring that the customer has valid bookings and that his trip goes smoothly. This includes noting when the flight cancellation occurs and working towards recovering from that situation.

Note in this example that it would be really undesirable for the airline to be unable to update its database to reflect the canceled flight because of some action by the travel agent. In particular, the travel agent should not be holding a lock on the flight reservation, denying access to the task that is updating the database to reflect the cancellation. The plane is not going to fly regardless of the state of the airline's database. Thus, there also needs to be a notion of times when a task cannot allow another task to interfere at all with its objective.

Some of the requirements for task functionality in the travel agent's multidatabase are evident from this example. Each task has a clearly-defined procedure. Also, the task of booking a trip starts when the travel agent first talks to the customer about the trip, and doesn't really end until the customer returns from the trip [Gra81]. Because the trip involves a single person, and because the various subtasks such as booking the flight and reserving the rental car are related, the trip should really be treated as a self-contained unit. Thus, the task should always be identifiable using a single, unique identifier.

Thirdly, tasks operate on a multidatabase that evolves over time, and tasks need to take into account the evolution of the database as they work towards their objective. Tasks may interact if they operate on the same information in the multidatabase. There also may be interactions between tasks and independent transactions on the local database. The task needs to be able to define when interactions can occur and to manage any desirable or undesirable interference between two tasks. The ability to completely back out of a specific task is also needed (provided that the individual steps taken towards completing the task can each individually be backed out of). This would occur, for instance, if the customer decides not to take the trip after all.

The characteristics of these tasks are summarized in Figure 2.

1.4 Approach to Task Execution

This paper explores how to manage tasks appropriately in a multidatabase. Specifically, we examine the special requirements of tasks initiated by long-term planning applications requiring information from multiple databases. Most multidatabase transaction management work up to this point [Pu88, EH88, BST90, DE89] explores how to define and implement tasks which are atomic and executed in the multidatabase serializably or almost serializably. However, we believe that tasks may need to interact with each other in certain circumstances. Planning tasks, for instance, need to be cognizant of the evolution of the information in the

underlying multidatabase they access, and therefore cannot run in isolation. As a consequence, the underlying model for defining and executing tasks in the multidatabase should allow for controlled interaction among the tasks and with the independent transactions in the local databases. Specifically, tasks may be allowed to see changes in the multidatabase, including the effects of other tasks they may have yet to commit.

In this paper, we define a framework for executing tasks in multidatabases, called the *InterAction Model.* Each *InterAction* defines how a specific task is to be executed in the multidatabase. InterActions are not like traditional database transactions. Traditional transactions specify the steps required by a particular task. However, they are forced to operate with the same isolation constraint (serializability). Serializability forces an execution where any given transaction can only see the results of the transactions that precede it in some serial execution. InterActions allow the task definer not only to specify steps, but also to specify the task's isolation and non-isolation requirements explicitly. That is, InterActions specify both how to execute the tasks on the database and how the task should isolate itself from undesirable interference from other tasks. They also specify how the task should recover from certain types of desirable interference from other tasks.

Because InterActions are not necessarily atomic, they must themselves specify their notion of correct operation, and this notion must be enforced using some form of synchronization protocol. We assume that an InterAction definition includes some specification of the sections of its execution that cannot be interfered with by other transactions on the local databases. This is loosely related to the notion of *conflicts* defined in [NZ90].

An InterAction consists of a program of atomic steps. A step is an atomic unit of work that is executed on a single local database. Consecutive sets of steps may be grouped together into blocks; these blocks are executed on the multidatabase as if they were self-contained, traditional atomic transactions. That is, the atomic blocks define sets of steps that must run in isolation. However, these blocks are related to one another, both because they execute as a part of the same InterAction, and because as a consequence information can flow internally from one atomic block to another. Thus, there are constraints on the order in which these atomic blocks are executed, and on when the effects of the atomic blocks can be committed in the multidatabase. However, these constraints can be imposed on the InterAction history directly by the program that synchronizes the InterAction execution.

Preserving the atomicity of both the steps and the atomic blocks requires the cooperation of the local databases. This is because a step or atomic block may conflict not only with other InterActions and their steps, but also with independent transactions on the local database. Unfortunately, since the local databases are autonomous, we cannot directly change them to work with the multidatabase. Instead, we need to use the local databases in a way that ensures that these atomicity constraints are maintained. In other words, we need to ensure that conflicting steps and atomic blocks are scheduled by the local databases in a way that preserves their atomicity.

The actual work that needs to be done by the task is accomplished by running atomic database transactions *(global subtransactions)* on the local databases that make up 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. To enforce isolation, global subtransactions are used to hold resources. Thus, all resources touched by any step in an atomic block must be held by some global subtransaction until the execution of the atomic block completes. This approach requires that the transactions on the local database preserve the ACID properties (atomicity, consistency, isolation, durability). We also assume that atomicity is preserved in the local databases by using a concurrency control protocol based on serializability.

Because the effects of an InterAction persist, issues of recovery when different types of failures occur must also be addressed. As the InterActions are executed, information about that execution and about the dependencies between different steps is logged in the database. When a failure occurs, the log is used to determine which steps are affected. Because InterActions may be long and a failure may affect only a small portion of the task, forward recovery is preferable. This means that the recovery manager tries to preserve as much of the tasks' completed work as possible, and work forward from the point of failure. Also, because some of a task's work may be committed when the failure occurs, compensation [GS87] should be the primary means of removing the effects of an invalid step from the database.

1.5 Road Map

The rest of this paper is organized as follows: Section 2 gives a general description of multidatabases, describes the specific multidatabase architecture used in this research. It also describes some basic requirements that planning applications place on a multidatabase. Section 3 gives a basic definition of InterActions, including the properties required of any language for defining InterActions, and a specific InterAction Definition Language. Sections 4 through 6 give a basic overview of how to define correctness and give algorithms to ensure correct operation. Section 7 gives an overview of related research. Finally, we conclude in Section 8.

2 Multidatabases

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

A traditional database makes certain guarantees about the information it stores. It ensures that it remains *consistent (correct)*, provided that the transactions that access and/or update the information in isolation from other transactions. Also, it guarantees that the information is *persistent*. That is, the information in the database should be available outside the scope of the process that created it, and after that process has terminated. This should be true even when some failure has occurred that could cause the data to be corrupted or lost. For a multidatabase to conform to the expectations of its users, it also needs to provide guarantees concerning the correctness and persistence of the information stored in its local databases.

2.1 Multidatabase Structure

A multidatabase has two logical levels. The *local* level consists of the set of local databases. All of the information is stored in the multidatabase at the local level. The information in the local databases is accessed and updated using *global subtransactions*. Each global subtransaction does some part of the InterAction's task that requires accessing and/or updating information in a single local database. The local databases are assumed to be autonomous, and to make specific guarantees about the correctness and persistence of the information they store. Specifically, these include the ACID properties (atomicity, consistency, isolation, durability). We also assume that the concurrency control protocol for the local databases is based on serializability.

In this multidatabase model, a task accesses and/or updates information at the *global* level using *InterActions*. An *InterAction Manager* is responsible for coordinating the InterActions and for making specific guarantees about the correctness of the execution of the InterActions. Obviously, the InterAction Manager cannot make guarantees about the correctness of a transaction execution or the persistence of its effects that are stronger than those made by its local databases.

Figure 3 shows the basic multidatabase architecture used in this work. The multidatabase user interacts with the multidatabase by submitting InterActions to the InterAction Manager. The InterAction Manager in turn executes the InterActions by submitting global subtransactions to the local databases. Currently, the InterAction Manager is centralized. Since it could easily become a bottleneck, care must be taken to ensure that it can be distributed if the need arises.

Since this model of multidatabase does not restrict the activity of the local databases, the local transaction managers can choose to execute local transactions submitted from outside the context of the multidatabase. These are *independent transactions*.

Each local transaction manager has an *Agent*. The Agent operates on top of the local transaction manager, and all transactions on the local database (including both those submitted by the global transaction manager and those submitted independently of the multidatabase) pass through it. The Agent acts on behalf of the InterAction Manager, and serves as a local helper to enable the InterAction Manager to provide correct isolation between InterActions and the transactions submitted to the local databases that are not initiated by the InterAction Manager. For example, the Agent may monitor the database for activities that interfere with some InterAction's objective. Also, if two InterActions access the same set of databases, the InterAction Manager cooperates with the Agents to ensure that their global subtransactions are executed in a way that preserves correctness in the multidatabase.

2.2 Integration Assumptions

While many approaches to multidatabases have been described in the literature, no consistent model or integration strategy has been defined yet. Özsu and Barker [OB90] have defined a simplified taxonomy of integration strategies for multidatabases, categorizing them according to the degree of heterogeneity, autonomy, and distribution of the different local databases that comprise the multidatabase.

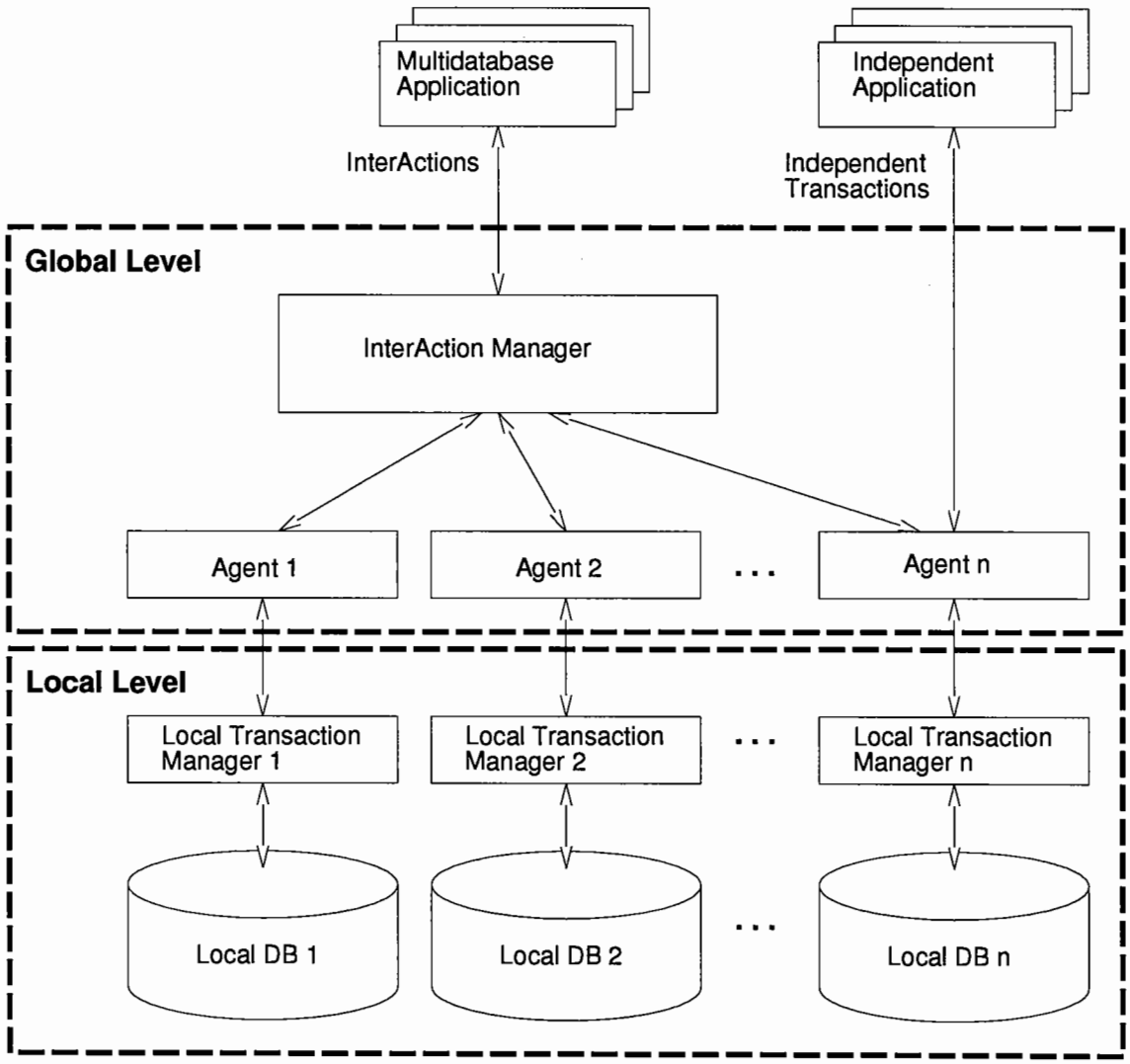


Figure 3: A Multidatabase Architecture

There are several factors in addition to the ones used by Özsu and Barker that can affect the character of a multidatabase. These include how the data is accessed, and what kind of functionality is allowed in the transactions on the multidatabase. This section describes various properties of multidatabases. Some of these are described in [DE89, OB90]. For each property, we also describe and defend the assumptions made in this paper.

2.2.1 Heterogeneity

In the multidatabase itself, *heterogeneity* refers to the extent to which the different local databases can differ in terms of their transaction management strategies, data models, and data manipulation languages. Other types of heterogeneity may be present; for example, the hardware that supports the different local databases and the communications protocols used in the underlying network. However, these are not directly relevant to the operation of the database, and therefore can be ignored.

In this research, no assumptions are made concerning the data models or data manipulation languages. Instead, we assume that the specifics of accessing the local databases are hidden in the actual implementation of the atomic steps that serve as the units of work in the InterActions. As stated previously, we do assume that the underlying transaction management strategies of the local databases support ACID transactions and serializable transaction executions.

2.2.2 Uniform Access

Uniform access ensures that the user of the multidatabase can access all of the information it stores using a single database access model, and without having to learn the local data models or to know in which local database the information is stored. We provide uniform access by providing a global interface to the multidatabase that is based on object-oriented views of the relevant data in each local database with methods to access the information in those views. The details of the implementation are hidden in the procedures. The methods themselves are called from tasks specified in our task definition language, TaSL.

An alternative to this approach is to allow the multidatabase user to access the data using his own local DML. We do not support this option. Instead, we abstract the issues of data model translation away from the issues of global transaction management, and hide them in the code that implements the methods.

2.2.3 Local Database Autonomy

Local database autonomy refers to the degree of control that the multidatabase has over the local databases. If the local databases are fully autonomous, they know nothing about the multidatabase or the global transaction manager. Neither do they know anything about the existence of the other local databases in the multidatabase. This means that strategies for synchronization, access control, and recovery in the multidatabase cannot assume the cooperation of the local databases. Neither can they expect features in the local databases that are not already provided.

The basic definitions of autonomy from [DE89] are summarized here.

Design autonomy concerns the assumptions that the global transaction manager makes about the data models, data manipulation languages, and transaction strategies of the local databases. We assume that all transactions on the local databases are atomic, serializable, and recoverable. We make no assumptions about the underlying data models at all; thus, the local databases may support different data models.

Communication autonomy concerns the assumptions the global database makes concerning the direct interactions among the local databases. We do not assume that the local databases can communicate with each other; they need not even know about each others' existence. Thus, we have full communication autonomy. We also make no assumptions concerning how the local databases are distributed, either geographically or with respect to the enterprises that are managing them.

Execution autonomy concerns any assumptions on how the local databases execute their transactions. We assume that the local databases can execute transactions in any way, provided that they maintain serializability and recoverability. Furthermore, we assume that the local databases can execute transactions that are initiated outside of the multidatabase. We do, however, assume that we can examine each operation that is about to be submitted to a local database, and possibly delay the submission of that operation based on the requirements of the multidatabase.

2.2.4 Distribution

Distribution characterizes how the location of the local databases is spread out. This includes the physical location of the databases. Are they within the same building? City? Continent? Also, it concerns the number of enterprises managing the different databases. Must the databases all be controlled by one company? Many different companies? Companies in different countries under different legal systems?

In this work, we make no assumptions at all about the distribution of the local databases. We do realize, however, that distribution can impact the efficiency of operation of the multidatabase.

2.2.5 Functionality

Functionality refers to the type of constraints that the multidatabase transaction model places on the expressiveness of the global transaction models. For example, early multidatabases such as Multibase [SBD+81b] were read-only – they did not allow multidatabase updates. Quasi-serializability [DE89] and multidatabase serializability [Bar90] both assume that the global subtransactions which run on behalf of a single global transaction have no data dependencies among them.

In this model, tasks are executed on the multidatabase in steps, where each step is an atomic piece of work on a single local database. The task defines which steps are required and which are optional. It allows alternative steps to be defined, which can also be used in completing the task if the preferred step fails. These features are similar to those provided by other transaction models, for example [GGK+90, ELLR90].

A task also defines how the steps are ordered. Steps may access information in the local database, update that information, or both. In addition, steps may depend on other steps, in that they read information that was retrieved from another local database by another step.

In addition to defining the steps, tasks also have an associated conflict definition, which expresses the task's isolation requirements. That is, a task defines what parts of its work must be done in isolation from the effects of other InterActions and independent transactions. This is required because these tasks are not forced to conform to a uniform isolation requirement such as serializability or quasi-serializability.

2.3 Multidatabase Access Assumptions

Tasks to be executed on a multidatabase do not necessarily have the same database access requirements as the type of task that is implemented as a transaction on a typical database. This section explores some of the requirements of tasks within the multidatabase, and why traditional transaction models do not suffice to fill these requirements.

2.3.1 Non-Atomic

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

Unfortunately, there are cases where multidatabase applications need to store, access and update persistent information, but where the the constraints of atomicity and serializability for the database transactions are too strict [Gra81]. For example, if a task reads a piece of information from the database, the multidatabase may not guarantee that result will be the same if the task reads the same piece of information again *(repeatable read)*. There is an instance of this in the travel agent example, where a flight is initially available and therefore a seat can be booked for the travel agent's customer. Later on, the flight is canceled, so the travel agent's query concerning the flight should reflect the change. In this situation, it is desirable to allow a non-repeatable read. Thus, multidatabases require other approaches to expressing and/or enforcing correct

database operation, and to constraining the execution history of the transactions on the multidatabase so that they are correct according to the new form of correctness specification.

2.3.2 Procedural

In a traditional database, tasks are identified with transactions. The code of the transaction specifies the flow of the task – what needs to be accomplished to complete the task, how the smaller subtasks are ordered, and what information passes among the smaller subtasks and between the user and the task as it progressed. Since all tasks are executed in total isolation from each other, this information is completely contained in the transaction definition.

In this multidatabase, the task definer describes the flow of the task, but the task is carried out ultimately using operations on the local databases. The definer needs to be able to describe different parts, or *steps* of the multidatabase transaction, where each step is a piece of the task that is executed on a single local database. The definer also needs to be able to describe how the steps must be ordered to complete the task, and to define local variables to hold intermediate information as the task progresses.

The flow of the task may be guided not only by the information it is given initially by its definer, but also by the information retrieved from the local databases. Thus the task can do things such as retrieve a set of available seats, sort them by increasing fare, and try to make reservations in that order.

Tasks can be described using a simple programming language where the statements are the steps. The programming language used to describe patterns is defined in Section 3.

2.3.3 Interactive (Cooperative)

Interaction among tasks is implicitly allowed any time one task makes its results visible to another task before it commits, thus allowing the second task to act on those results before the first task is committed. In some cases, this interaction is desirable. For instance, if a travel agent has booked a flight that is canceled, it is desirable for him to find out about that cancellation as soon as possible, to give him the best chance of making other reservations. However, other interactions may be undesirable. For example, if the travel agent has found an available flight for his customer, he does not want the last remaining seat to be booked by some other person before he can actually make the reservation.

Tasks are *cooperative* when they are allowed to interfere with each other when the effects of one task are helpful for the second task in accomplishing its objective. Cooperation between tasks occurs in a database when two tasks share some data, and one task can see the changes made by a second task before the second task has actually committed. In other words, cooperation can occur any time the effects of a task become visible before that task has completed.

In multidatabase environments, the results of one task become visible to other tasks when its subtransactions on the local databases commit. This is because the information in the multidatabase is directly managed by the local databases, and only indirectly managed by the multidatabase. The multidatabase has no ability to control the accessibility of a piece of information outside of the scope of a transaction that it is running on the local database that stores that information.

This type of cooperation is different from the type of cooperation defined in cooperative transaction hierarchies [NRZ91]. There, patterns are used to *structure* the cooperation explicitly, so that the sequence of operations in the history reflects an explicitly-defined interleaving of operations by different transactions. In a multidatabase, a task can allow specific kinds of interference at different periods during its execution, but there is no real way of forcing another task to actually *do* something at any point. However, these two types of cooperation may be combinable to form a transaction management strategy for cooperative planning applications.

2.3.4 Adequately Isolated

Because tasks in the multidatabase can be executed concurrently, there can be undesirable side-effects if different operations associated with two different tasks interleave their operations on the same information inappropriately. The problem of isolating the tasks' effects from each other has been studied extensively in the database literature [BHG87], and these studies form the basis of the various theories about serializability and the ensuing practices of timestamp, lock-based, and other forms of concurrency control.

Even though we believe serializability is not an appropriate approach to concurrency control in a multidatabase, it is necessary for tasks to be adequately isolated from each other to prevent undesirable side-effects. Each task should be able to specify and enforce its own isolation requirements. This can be done by defining the periods during the task's execution where conflicting operations cannot be tolerated. These conflicts constrain the histories both in the local database and the multidatabase.

An example of this can be seen in the travel agent example. Once the travel agent is assured that a specific seat is available on the United flight, he should be able to reserve that seat for his traveler without interference from others. Thus, he wants the execution beginning with the time he first determines that the seat is available until at least the point where he has claimed the seat for his traveler to be executed in isolation.

2.3.5 Recoverable Only Using Compensation

In a multidatabase environment, tasks tend to be long. Therefore, as discussed in the previous section, the task may not want to hold its resources until it terminates. Also, tasks do not necessarily commit atomically, so a global subtransaction is likely to commit in the local database before its global transaction commits in the multidatabase.

In these situations, the real problem occurs when some transaction on a local database commits, and then its multidatabase transaction fully or partially aborts. Garcia-Molina and Salem [GS87] argue that in these situations, the only way to back out the changes made by the transaction on the local database is to ensure that a *compensating transaction* runs to completion. A compensating transaction is designed to reverse the effects of the initial transaction. For example, if the initial transaction is the booking of a particular airplane reservation, the compensating transaction would be to cancel the reservation.

Compensation-based recovery is a sloppier form of recovery than the traditional *undo* for several reasons. First, there is a window of time between the point where the original transaction commits and the point where the compensating transaction begins. During this time, other transactions may read the effects of the original transaction and use that information in their own execution. These transactions then are ultimately dependent on the effects of some transaction that, in some sense, didn't run [KLS90]. While these dependencies may be difficult to compute, it is important to maintain them for correct recovery in a system that allows this type of dependency to develop in the first place. Thus, in addition to maintaining the traditional logs and defining recovery procedures that use those logs, we need to define a complete scheme for specifying, computing, and maintaining these dependencies.

One place this problem becomes apparent is when other transactions do operations that prevent a compensating transaction from completing. For instance, this would occur if the traveler pays PanAm for his reservations, and his reservation is later canceled because PanAm filed for bankruptcy. In this situation, PanAm may not be able to pay the traveler back immediately.

2.3.6 Partially Undoable

In the case where some problem occurs, it is not necessarily desirable to abort the whole task and start all over again. For example, if a traveler's flight is canceled, it may not be necessary to touch his reservation for his rental car. Also, tasks can be of long duration, and aborting a long task may involve undoing large amounts of work.

When problems develop because some already-completed work becomes invalid, the ensuing recovery process should have as little impact on the rest of the task as possible. Thus, the recovery manager needs to determine which other parts of the task are no longer valid, and "undo" those parts using compensation. Then, the task manager needs to do something else that accomplishes those pieces of the task in some other way.

2.3.7 Identifiable

A multidatabase task is really a grouping of related operations on different databases that accomplish a single, specific purpose. It may not be wise to view different parts of the task as completely different pieces, because relationships form between those parts, and these relationships must be maintained. For example, if the customer misses his flight on his business trip, all of his travel plans may need to be adjusted. The

example of Section 1.1 notes that all of the plans including the car rental may be affected if the customer's flight is canceled. Thus, it is important to be able to make conclusions about what to do about the car rental based on the status of the flight.

Another reason that a task requires a unique identity has to do with resource management. For example, even if compensation is the basic recovery strategy, the task may want to keep track of the status of the resources it needs for the correct compensation in case the task is aborted. If something happens to some required resource, the task needs to know that compensation is now no longer possible. Another case where the task needs to manage resources is when it is watching for operations that interfere with its work. For example, once the travel agent has made the flight reservation, he needs to monitor that reservation to ensure that it is not canceled. If the flight is canceled, then the task needs to do some work to recover its state. Clearly, the local transaction that made the reservation has committed, because the flight reservation is available for the airline to do the cancellation. However, there is something (the task) that is still responsible for initiating the watch on the resource and for receiving and processing any notification that the reservation has gone away.

3 InterActions

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

An InterAction is specified by some application (such as the travel agent application) as a program or command sequence. Information is requested from and sent to the application using message-passing (notifications).

The execution of all of the InterActions in the multidatabase is controlled by a single InterAction manager. The InterAction manager uses the definitions of the InterActions currently in process to ensure both that each InterAction's steps are done in an appropriate order and that each InterAction's isolation requirements are met.

3.1 InterAction Structure

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

The parts of the program that actually interact with the local databases are called *actions*. Each action defines a set of functionally equivalent steps, one of which must be taken for the action to complete. Each step is a piece of code that is executed atomically on a single local database to accomplish some part of the InterAction's task.

The program specifies how the different actions can be combined to accomplish the task correctly. This includes specifying alternative actions if the preferred action is not possible. For instance, if there are no more flights from point *from* to point *to*, the traveler may wish to rent a car and drive to his destination, rather than fly and rent a car at the destination.

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

In addition to defining the flow of a particular task, an InterAction also must define the task's isolation requirements. Isolation requirements are defined over spans of the execution of the InterAction.

This model allows two forms of isolation. The first, called a *strong conflict*, defines what absolutely cannot happen during a specific span. With a strong conflict, the job of the InterAction manager is to ensure that there is no way that the conflicting operations in the local databases can occur.

The second form of isolation is called a *weak conflict*. It defines a set of conditions which should be preserved for a specific span of the InterAction's execution. If these conditions are violated, then the weak conflict also specifies what the contingency plans are. For example, a weak conflict might be specified that, once the traveler's flight is booked, that reservation stays around until he actually flies. If the reservation goes away, for instance if the flight is canceled, the weak conflict specifies how to recover.

3.1.1 Example

Figure 4 gives a flow diagram 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).

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

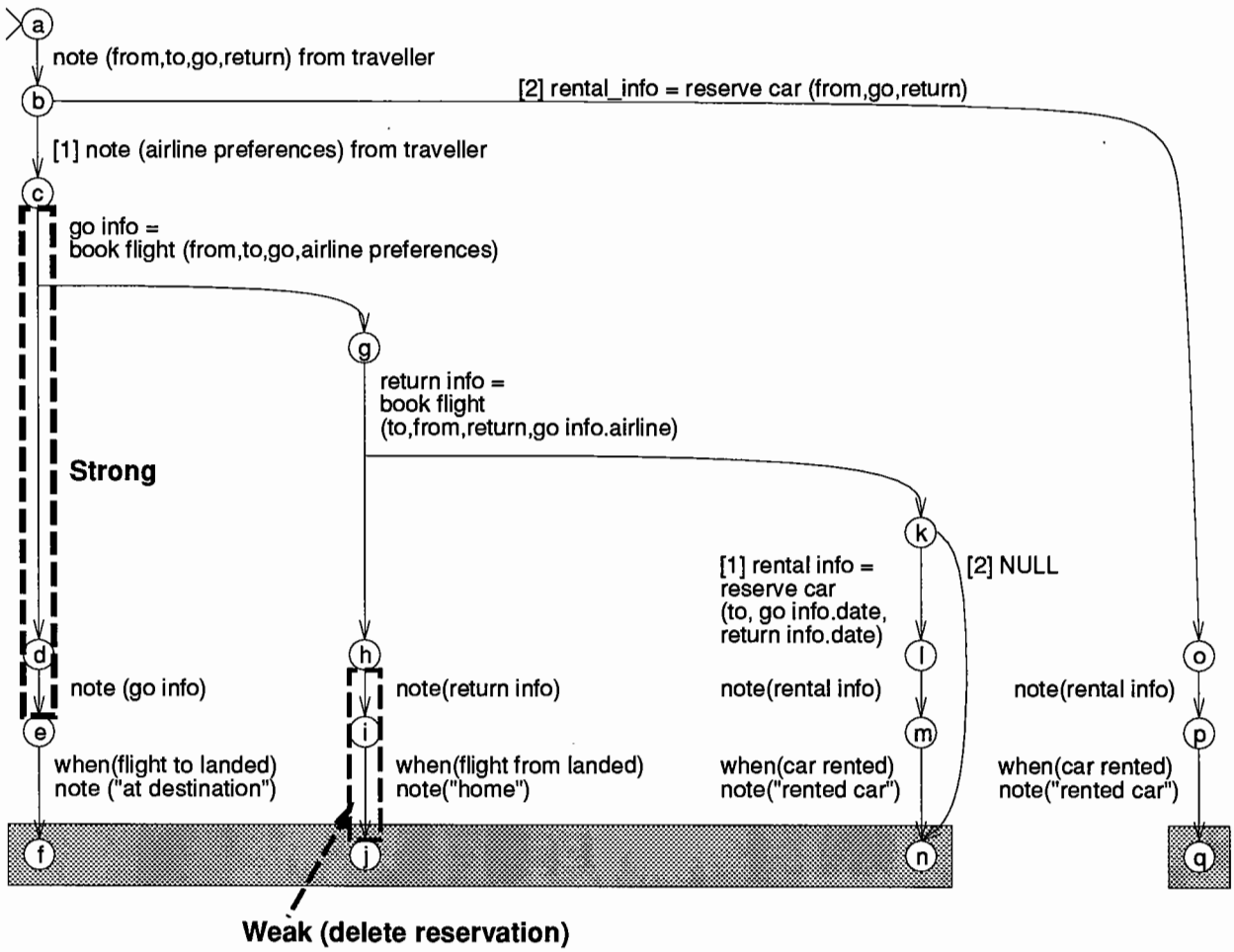


Figure 4: Task flow for a trip reservation. Example conflicts are shown in dashed boxes.

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

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

Figure 4 also shows examples of a strong and a weak conflict for the defined InterAction. The duration of the conflict is indicated by a dashed box. The label associated with the box indicates whether the conflict is strong or weak. If it is a weak conflict, the label also indicates the operation that cannot occur during the span delimited by the box. In the figure, the strong conflict ensures that, once an open seat on the outgoing flight is found for the traveler, that seat remains available until the actual reservation is made and the travel agent has a confirmation. The weak conflict on the homeward flight ensures that the reservation stays around until the traveler actually takes the flight.

3.1.2 Actions

An *action* is a partially ordered set of steps. The steps defined in the partial order are functionally equivalent. For example, one of the steps may reserve a flight on United Airlines. Another step may reserve a comparable flight on TWA, and a third step on PanAm. Any one of these steps would accomplish the same objective: Get the traveler from point *from* to point *to* at about the time *go*.

The partial ordering of the steps indicates the priorities of the task definer. For example, the traveler may prefer to travel on United, because he has a first-class upgrade and the trip is a long one. However, if United is not available, either TWA or PanAm will do. This action would then be expressed as follows:

{
    { *book_United_flight (from, to, go)* },
    { *book_TWA_flight (from, to, go)* or *book_PanAm_flight (from, to, go)* }
}

The InterAction manager attempts the steps in some order consistent with the partial order defined by the action. In the task flow example of Figure 4, this action returns *go_info*, the information on the outgoing flight (e.g. airline, flight number, time, airfare).

Actions and sets of related actions may be *vital* or *non-vital*. A vital action (or set of actions) must be accomplished. A non-vital action (or set of actions) does not need to be accomplished. Non-vital actions are expressed using the *NULL* action as an alternative to the non-vital action or action set. In Figure 4, the set of actions associated with renting the car at the destination *(from)* is non-vital, so there is an alternative *NULL* action (with priority [2]) from state k to state n. Thus, the car rental is tried first. However, if the car rental did not succeed, the *NULL* action is tried next, and succeeds automatically.

In other work [GGK+90, ELLR90], actions were explicitly tagged as vital or non-vital. If a non-vital action could not be accomplished, it still succeeded. We instead provide the *NULL* action as an explicit alternative, because the non-vital construct is ambiguous in the presence of prioritized alternatives. For example, consider the case where there is a higher-priority non-vital alternative and a lower-priority vital alternative. If the first alternative fails, do you assume it succeeds or do you try the second alternative? If you try the second alternative, what do you do if it fails? Also, there is the case where there are two equal-priority non-vital alternatives. If both fail, what state does the task end up in?

3.1.3 Steps

A *step* defines a cohesive unit of operations on a single local database. In the task flow, an example of a step is *book_United_flight (from, to, go)*. Its function is to check the database for United airlines to see if they have an available flight at the specified time, and make a reservation if one is found.

In this work, each step is implemented as a procedure call on the InterAction Manager's *Step Library*. Because compensation is the primary form of recovery, each step also has a compensating step defined for it in the Step Library.

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

Note that, because there are no guarantees concerning the atomicity of a set of steps, it is not necessary to execute two steps of the same InterAction on the same database as a part of the same global subtransaction. One global subtransaction could execute one step, then commit. When the next step on the same local database is reached, a new global subtransaction must then begin. We use strong conflicts to express situations where two different steps in the same local database must be executed within the same global subtransaction.

3.1.4 Events

An *event* is something that happens externally to the InterAction, but which influences the execution of the InterAction. An event here is defined as a specific operation on a specific data item by some other InterAction or some independent transaction on a local database.

There are two useful ways to use events in an InterAction definition. The first way is to delay the execution of the InterAction until some event has occurred. For example, the travel agent may wish to delay the commitment of an InterAction until the traveler has completed his trip. The second way events are useful is in noting when weak conflicts are violated. A weak conflict specification defines some event to wait for. The rest of the weak conflict effectively specifies the event handler for that event. This is explored in more detail in the section on weak conflicts. Events also can be used to monitor the state of the database to ensure that compensating steps will succeed if specific steps abort.

3.1.5 Spans

Spans define periods of execution of an InterAction. A span is defined by a start and end state. During execution, the span begins when an action is executed that leaves the start state (such as making a reservation). It ends when an action is executed that places the InterAction in the end state (such as taking a flight).

Spans are used to define the execution periods over which strong and weak conflicts are in effect; that is, in defining an InterAction's isolation requirements. When the InterAction is executed, the strong or weak conflict takes effect atomically with the action that begins the span, and is removed atomically with the success of the action that ends the span.

3.1.6 Strong Conflicts

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

For correct operation in the multidatabase, the InterAction manager must ensure that strongly conflicting spans of code are executed in isolation in the multidatabase. Each strong conflict must be enforced within a single transaction on the local database. This transaction prevents other transactions on the local database (either global subtransactions or independent transactions) from inappropriately accessing the resources that need to be preserved for the strong conflict to be maintained. For example, since a strong conflict is defined between the time the travel agent finds an available reservation for the traveler and the time the travel agent actually makes the reservation, there must be a single local transaction on the airline's database that finds the empty seat, blocks access to the seat until the reservation is made, and ultimately makes the reservation.

A strong conflict is expressed as a span over which other operations are blocked. In particular, no global subtransaction is allowed to commit during the span defined by the strong conflict.

3.1.7 Weak Conflicts

Weak conflicts indicate the conditions that should be preserved in the database during a specific execution span of an InterAction for it to be able to continue executing from its current state. In addition, a weak conflict defines what is necessary to recover when its conditions are violated. In other words, weak conflicts

do not indicate something that should be prevented necessarily. Rather, a weak conflict indicates conditions that must be preserved, or the task requires recovery. Thus, the violation of a weak conflict should cause the InterAction Manager to be notified, and the defined recovery measures to be taken.

Usually, a weak conflict violation indicates the full or partial invalidation of some step that has already been executed. Thus, the violation of a weak conflict could be treated analogously to the explicit aborting of some completed action in the InterAction.

Weak conflicts are not necessarily enforced by holding locks or any other equivalent function in the local database. For example, in the case where the plane flight is canceled, the traveler obviously does not want to hold the lock on his reservation. That would block his notification of the flight cancellation. Instead, weak conflicts may be allowed to occur, but their occurrence means that the InterAction Manager needs to institute some form of recovery for the InterAction.

Because of the autonomy requirements on the local databases, the enforcing of weak conflicts needs to be done by the InterAction Manager directly. This is because most local databases do not support the ability to detect when certain conditions are violated, while at the same time allowing the operations that cause the violation to continue. Instead, the InterAction manager must monitor the local databases for weak conflicts itself. This can be done using a variety of techniques, including periodic polling and explicitly monitoring the input command streams to the local databases.

Weak conflicts are expressed as a span over which an ECA-type rule is enforced. ECA rules [DHL90] specify a specific event to monitor, conditions that must be checked if the event occurs, and actions to take if the condition is violated.

3.2 InterAction Definition Language

This section defines a language, TaSL (for Task Specification Language), for defining InterActions. TaSL is different from the above specification in that it is more closely related to a programming language. TaSL is modeled primarily after block-structured programming languages.

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

The InterAction definition language requires facilities to express actions, variable assignment and use, strong and weak conflicts, parallelism, and alternatives. The following sections describe the language facilities available to describe these functions. The travel agent example task is given as an example TaSL definition.

3.2.1 Actions

An action ultimately is a partially-ordered sequence of procedure calls to steps. It is specified in the following form:

$$steplist_1\ steplist_2 \ldots steplist_m$$

where each step list is an unordered list of calls to specific step procedures, as follows:

$$step_A \text{ or } \ldots \text{ or } step_N$$

Within the code for the InterAction, actions may be specified directly within an *execute (<action>)* statement. An action may also be kept in a variable (of type *action*), and executed using the execute statement. Alternatively, an action could be represented as a procedure call, similar to the step procedure call, which takes a set of parameters, and attempts the steps directly. Actions also may be labeled. In particular, actions will be labeled if they have to be referred to by other statements in the language, such as *abort* statements. A labeled action has the form *<label>: <action>*.

3.2.2 Variable Declaration, Assignment, and Use

Variable declarations are of the form *<type> <name>* or *<type> <name$_1$>, ..., <name$_n$>*. A type is either the standard *int*, *char*, etc. that are used in C or some object view of information on some local database. In addition, there is the type *action*, which is a partial ordering of procedure calls of the form

defined in the previous section. At any time, the results of a step can be assigned to an internal variable. This is done with a simple assignment statement:

$$< variable >=< result > .$$

The type of the result must match the type of the variable.

Variables are used in the normal way. As in C, the components of the variable can be addressed using a dot notation, e.g. *go_info.airline*. When passed as a parameter, all variables use call-by-value semantics.

3.2.3 Alternatives

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

if not { $<action_1>$ }
then if not { $<action_{2a}>$ *or* $\cdots$ *or* $<action_{2m}>$ }
then if not ...
then { $<action_n>$ }

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

3.2.4 Events and Waiting

Events in TaSL correspond to the execution of specific operations on specific data items. They are specified as tuples

$$< local\_db,\ operation >$$

where *local_db* is the database on which the data item resides, and *operation* is the conflicting operation.

When the event takes place, the InterAction Manager notifies the relevant InterActions by sending it the event information including the identifier of the InterAction did the conflicting action (if it was another InterAction) or a note that it was done by some independent transaction.

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

3.2.5 Strong Conflicts

In TaSL, strong conflicts are expressed as blocks of code that are atomic. This is defined using the *atomic* construct, which is of the form *atomic* { <*atomic_block*> } What this means is that the set of InterAction statements in *atomic_block* cannot be interrupted by *any* conflicting operation. Effectively, this is the same as saying that any resource taken during the execution of an atomic piece of code must be held until the atomic code completes its execution.

3.2.6 Weak Conflicts and Aborting

Weak conflicts are problematic in that they do not fit well into the block structure of the language. In particular, one possible use for a weak conflict is to guard some specific change made for the benefit of the user, such as a flight reservation. The specific nature of the weak conflict is not determined until the actual reservation is made, and the flight, date, and time is known. However, there may be a window of time between the point where the reservation is made and the point where the weak conflict is put into place. If this window exists, then there may be a point where the reservation could get canceled, but the weak conflict would not notice. To close this window, the weak conflict must be in place before the global subtransaction

that makes the reservation releases its resources. This means that the weak conflict must be set before the strong conflict ends, and thus the weak conflict is set within the same atomic block as the step that makes the reservation. However, the weak conflict needs to persist for a longer time, and thus most certainly will be released outside of that atomic block. This situation may be seen in the example at the end of this section.

Because of this, TaSL treats weak conflicts more like exceptions than like locks. It uses the ECA paradigm [DHL90] to define the exception. The event and the condition define the signal to wait for, and the action defines the signal handler. However, the weak conflict may be released at some point of time that is not the end of the InterAction. Thus, there must be an option to specify the ending time of the weak conflict. This is done using an *until* statement, which specifies the label indicating the statement that ends the weak conflict.

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

$$[until < label >] on < event > [if < condition >] \{< handler >\}$$

where square brackets indicate optional parts of the construct. If the *until* statement occurs, there must be some statement "downstream" from the on condition that has that label. During execution, the weak conflict is released atomically with the execution of the labeled statement.

Typically, the event in the *on* statement is some operation on some local database. They event can also be specified as a procedure that determines what events to watch for at execution time. The condition in the *on* statement is some boolean algebraic expression. The variables in the expression may consist of constants, information returned from the event (such as the ID of the InterAction that provoked the event), or values read from the local database in which the event occurred. The handler typically specifies a step to abort. This does not necessarily mean that the global subtransaction that executed that step must be aborted. Likely, that transaction is no longer running. Rather, the effects of that step have been undone and recovery should be initiated.

An abort statement has the form *abort (<label>)* where *label* is the label of the statement that should be aborted. The handler is not limited to just an abort statement, however. [GSW87] claims that, in the implementation of the collaborative editor CES, it would have been useful for the user to have some explicit way of specifying user-oriented recovery actions. Thus, the handler may include clean-up actions in addition to undoing the work that is now invalid. However, to prevent circular triggering of handlers in the database, the handler may not modify information in the multidatabase.

3.2.7 Concurrency

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

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

In this case, a concurrent thread named *name* is created with the same context as the current thread, and that thread executes the code in *body*. The concurrent thread completes at the end of the code in *body*. If the fork statement occurs within an atomic block, the new thread is assumed *not* to be a part of the same atomic block as the current thread. Instead, the new execution thread executes outside the scope of that atomic block, while the continuing thread continues executing within the atomic block.

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

$$join\ (< name_1 >, \ldots, < name_n >)$$

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

3.2.8 The Trip Example

Figure 3.2.8 shows the trip example from Figure 4 with its associated conflicts, defined in TaSL. In this example, *note* is the action that places information in the travel agent's private database.

```
begin_InterAction (trip, traveller)
    place from, to;
    date go, return;
    list airline_preferences;
    data go_info, return_info, rental_info;

get_travel_information (&go, &return, &from, &to);
note ("flying from ", from, "to ", to, "on date ", go);
note ("returning from ", to, "to ", from, "on date " return);
if not {
        airline_preferences = get_airline_preferences (traveller);
        atomic {
            flight_to:  go_info = book_available_flight (from, to, go, airline_preferences);
            fork (T1) {
                atomic {
                    flight_from:  return_info =
                        book_available_flight (to, from, return, go_info.airline);
                    fork (T2) {
                        if not {
                            atomic { rent_car: rental_info =
                                        reserve_avis_car(to, go_info.date, return_info.date);
                                note ("Car rental infromation:  ", rental_info);
                                until (car_rented)
                                    on (rental_info.local_db, rental_info.undo_operation)
                                    { abort (rent_car); };
                            }  /* end atomic */
                            wait (rental_info.local_db, rent(rental_info.car));
                            car_rented:  note ("Rented car in ", in);
                        }  then NULL;       /* non-vital action */
                    }  /* end fork (T2) */
                    note ("return flight information:  ", return_info);
                    until (flew_from)
                        on (return_info.local_db, return_info.undo_operation)
                        { abort (flight_from); };
                }  /* end atomic */
                wait (return_info.local_db, landed (return_info.flight));
                flew_from:  note("Trip complete");
            }  /* end fork (T1) */
            note ("flight information:  ", go_info);
            until (flew_to)
                on (go_info.local_db, go_info.undo_operation)  { abort (flight_to); };
        }  /* end atomic */
        wait (go_info.local_db, landed (go_info.flight));
        flew_to:  note ("Flew to ", in);

}  /* end first alternative */

then {   /* other alternative */
        atomic { rent_car: rental_info =
                    reserve_avis_car(to, go_info.date, return_info.date);
                note ("Car rental infromation:  ", rental_info);
                until (car_rented)
                    on (rental_info.local_db, rental_info.undo_operation)  { abort (rent_car); };
        }  /* end atomic */
        wait (rental_info.local_db, rent(rental_info.car));
        car_rented:  note ("Rented car in ", from);

}  /* end second alternative */
commit_InterAction
```

Figure 5: TaSL code for the trip reservation example.

4 Multidatabase Correctness with InterActions

In Section 2.3.1 we argued that transaction atomicity and serializable histories may not be an appropriate approach for defining correctness in multidatabases. Instead, we examine other ways of expressing the correctness of a multidatabase history, and for enforcing that notion of correctness during the operation of the multidatabase. This section discusses InterAction correctness, and presents some requirements for enforcing the correct execution of a set of concurrent InterActions on the multidatabase and independent transactions on the local databases.

In this scheme, each InterAction defines its own expectations concerning correctness. This includes a program (pattern) to express both the steps that need to be taken for the InterAction to complete and the ordering constraints on those steps. It also includes isolation (conflict) information, which expresses what sets of steps must be done in isolation from other steps. At any point in the multidatabase execution, the pattern and conflict information for all the active InterActions is used to constrain the order of steps in the multidatabase history and in the local database, to ensure that the overall operation of each InterAction conforms to its pattern and conflict requirements.

In this section, we define correctness criteria for executing programs with strong conflicts only, and we ignore the presence of weak conflicts. This is because violating a weak conflict is essentially the same as aborting a step. Consequently, maintaining correct operation with respect to weak conflicts means correctly detecting when the weak conflict is violated, and correctly recovering after the abort. We address both weak conflicts and recovery in later research.

4.1 Definition for Correctness

The programs used to define transactions on centralized databases define the pattern information for those requirements. The isolation requirements in centralized databases are fixed by the database: Every transaction is executed in such a way that the database guarantees that it is executed atomically, operates on a consistent database state, is isolated from the effects of concurrently-running transactions, and can assume that its effects are durable once the transaction commits (the *ACID* properties). Usually, transaction atomicity and isolation are enforced by guaranteeing that the database execution is *serializable*.

With InterActions, the unit of isolation is not the transaction, but rather is the atomic block (or the step if it occurs outside the scope of any atomic block). Ignoring the issues of durability (which will be addressed in the work on recovery) we have the following intuitive definition for correct execution of atomic blocks in an execution of concurrent InterActions:

Definition 4.1 *An InterAction history is* correct *if its atomic blocks are executed* serializably, *and each InterAction manages its information* consistently.

Serializability is a known correctness criterion. Consistent management in centralized transactions is based on serializability – effectively, no other transactions interfere with a given transaction because the execution conforms to some serial order. However, the "lifetime" of a particular piece of information in an InterAction may span multiple atomic block executions. Therefore, the consistency requirements must be made explicit in the correctness definition.

The following section lays some groundwork for defining correctness more rigidly. We then proceed to formalize our correctness requirements in Section 4.3.

4.2 Some Definitions

4.2.1 Multidatabase

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

$$MDB = \cup_i LDI_i$$

where LDI_i is some local database in the multidatabase.

The set of information that a multidatabase can provide access to is the union of the sets of information available in its current collection of local databases. Note that while the multidatabase does not store extra

4 Multidatabase Correctness with InterActions

In Section 2.3.1 we argued that transaction atomicity and serializable histories may not be an appropriate approach for defining correctness in multidatabases. Instead, we examine other ways of expressing the correctness of a multidatabase history, and for enforcing that notion of correctness during the operation of the multidatabase. This section discusses InterAction correctness, and presents some requirements for enforcing the correct execution of a set of concurrent InterActions on the multidatabase and independent transactions on the local databases.

In this scheme, each InterAction defines its own expectations concerning correctness. This includes a program (pattern) to express both the steps that need to be taken for the InterAction to complete and the ordering constraints on those steps. It also includes isolation (conflict) information, which expresses what sets of steps must be done in isolation from other steps. At any point in the multidatabase execution, the pattern and conflict information for all the active InterActions is used to constrain the order of steps in the multidatabase history and in the local database, to ensure that the overall operation of each InterAction conforms to its pattern and conflict requirements.

In this section, we define correctness criteria for executing programs with strong conflicts only, and we ignore the presence of weak conflicts. This is because violating a weak conflict is essentially the same as aborting a step. Consequently, maintaining correct operation with respect to weak conflicts means correctly detecting when the weak conflict is violated, and correctly recovering after the abort. We address both weak conflicts and recovery in later research.

4.1 Definition for Correctness

The programs used to define transactions on centralized databases define the pattern information for those requirements. The isolation requirements in centralized databases are fixed by the database: Every transaction is executed in such a way that the database guarantees that it is executed atomically, operates on a consistent database state, is isolated from the effects of concurrently-running transactions, and can assume that its effects are durable once the transaction commits (the *ACID* properties). Usually, transaction atomicity and isolation are enforced by guaranteeing that the database execution is *serializable.*

With InterActions, the unit of isolation is not the transaction, but rather is the atomic block (or the step if it occurs outside the scope of any atomic block). Ignoring the issues of durability (which will be addressed in the work on recovery) we have the following intuitive definition for correct execution of atomic blocks in an execution of concurrent InterActions:

Definition 4.1 *An InterAction history is* correct *if its atomic blocks are executed* serializably, *and each InterAction manages its information* consistently.

Serializability is a known correctness criterion. Consistent management in centralized transactions is based on serializability – effectively, no other transactions interfere with a given transaction because the execution conforms to some serial order. However, the "lifetime" of a particular piece of information in an InterAction may span multiple atomic block executions. Therefore, the consistency requirements must be made explicit in the correctness definition.

The following section lays some groundwork for defining correctness more rigidly. We then proceed to formalize our correctness requirements in Section 4.3.

4.2 Some Definitions

4.2.1 Multidatabase

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

$$MDB = \cup_i LDI_i$$

where LDI_i is some local database in the multidatabase.

The set of information that a multidatabase can provide access to is the union of the sets of information available in its current collection of local databases. Note that while the multidatabase does not store extra

information independent of the local databases, it may choose *not* to provide information that is stored in some local database.

4.2.2 InterAction

The execution of a specific InterAction is represented by a partial order:

$$I = (S_I, <_I^S),$$

where

- S_I is the set of steps that are executed as a part of I, plus the delimiters *begin_InterAction(I)*, *commit_InterAction(I)* and *abort_InterAction(I)*,
- $<_I^S$ is a partial order of those steps as executed, where $S_a <_I^S S_b$ in the history if S_a and S_b share the same execution thread, and S_a is executed before S_b,
- for a specific InterAction I, $S_I <_I^S$ *begin_InterAction(I)* for any $S_i \neq$ *begin_InterAction(I)*, and
- for a specific InterAction I, if S_z is either a *commit_InterAction(I)* or an *abort_InterAction(I)*, then for any $S_i \neq S_z$, $S_i <_I^S S_z$.

Adjacent steps of an InterAction may be grouped together to form *atomic blocks*. The global subtransactions that execute the steps in some atomic block *participate* in that atomic block. In the InterAction model, we restrict global subtransactions such that a single subtransaction can only participate in a single atomic block: Therefore, the lifetime of an atomic block encompasses the lifetime of each of its global subtransactions.

4.2.3 Global Serialization Order

The *global serialization order* is the order in which the atomic blocks are serialized in the multidatabase. The global serialization order specifies the order in which the atomic blocks' global subtransactions should be serialized in the local databases. That is, for any two InterActions T_1 and T_2 that access information in one or more of the same local databases, then all such local databases either serialize T_1's subtransaction before T_2's or T_2's subtransaction before T_1's.

As an example, consider what happens when two InterActions submit global subtransactions to two or more of the same local databases at approximately the same time. For example, say InterActions I_1 and I_2 both have steps to make reservations on the same flight, and to rent a car at the destination. If both are executing their steps concurrently, then it is desirable for one InterAction's steps to consistently occur before the others across all databases. Consider the case where there is exactly one reservation left on the flight, and exactly one car. If a consistent ordering of the InterActions does not occur, then the following effective execution order is possible:

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

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

Now, in this example, if the two InterActions do not have their steps in a single atomic block, then this behavior is reasonable. However, if the steps are contained within atomic blocks, this behavior is not acceptable. This is because the two atomic blocks' results are affected by each other. I_1's atomic block fails because of the execution of I_2's, and vice versa. Thus, the expected execution of these InterActions in this

case depends on the steps for the two InterActions being executed in some consistent order. That is, I_1's steps are always executed before those of I_2, or vice versa.

Formally, the global serialization order is a partial order

$$GSO = (B, <_S^B)$$

where

1. B is the set of atomic blocks in the execution, and
2. if atomic blocks B_a and B_b in B access one or more of the same local databases, then either $B_a <_S^B B_b$ or $B_b <_S^B B_a$.

4.2.4 Information Flow Order

The *information flow order (IFO)* within an InterAction is the partial order in which uncommitted information can flow from one step to another. This occurs through the variables internal to the InterAction itself. An InterAction can read and/or modify information in some local database in one step, and in some other step use that information as it modifies information in some other local database. However, this information flow always occurs through some variable local to the InterAction, as the flow spans steps and, therefore, program statements.

A second way information could flow from step to step in an InterAction is if one step executed by some global subtransaction T_1 writes to some local database, and some other step of the same InterAction executed by some *other* global subtransaction T_2 reads that same data. However, in this case the information written by the first step has already been committed to the local database, therefore we do not need to consider it when we consider the flow of uncommitted information.

When information generated by step S_a is placed in some internal variable, and that variable is read by step S_b, we call S_a the *producer* of the information, and S_b its *consumer*.

We define the information flow for an InterAction I as a partial order:

$$IFO_I = (S, <_F^S)$$

where

1. S is the set of subtransactions associated with InterAction I, and
2. if some of the information written by step $S_a \in S$ is stored in some some variable V internal to I, and step S_b reads V, then $S_a <_F^S S_b$.

In the InterAction model, information always flows through variables. In particular, we never use message passing or synchronization primitives to pass information between two parallel threads of the same InterAction. Therefore, information is kept in the context of the processes that are running the InterAction. Because of this, we have the following lemma:

Lemma 4.1 *For any InterAction execution,* $<_F^S \subseteq <_I^S$.

Given the definition of some InterAction, its information flow can be determined at the latest as it is being executed.

4.2.5 InterAction Manager History

The InterAction Manager's history contains an interleaving of the steps of a set of concurrent InterActions. Intuitively, an InterAction is represented in the history as a partial order of steps, beginning with a *begin_InterAction()* delimiter and ending with either a *commit_InterAction()* delimiter or an *abort_InterAction()* delimiter. The part of the history where the InterAction was active is exactly the part between the two delimiters. The history itself is an interleaving of the steps of its active InterActions. These steps must be ordered correctly in terms of the InterAction definitions.

An additional constraint on the order of the history is specified by the conflicts defined by the active InterActions. If an InterAction contains an atomic block B, then for all other steps S that conflict with that block, either S must happen before any step in B, or S must happen after any step in B.

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

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

where

1. $HE_I = \cup_{I \in A} S_I$,
2. $\cup_{I \in A} <^S_I \subseteq <^S_{IH}$, and
3. for any set of steps B in the same atomic block and any step S executing on the same local database as some step in B, either $S <^S_{IH} B$ or $B <^S_{IH} S$.

Note that the global serialization order is derivable from the information in $<^S_{IH}$. In particular, consider two atomic blocks B_i and B_j, with steps $S_i \in B_i$ and $S_j \in B_j$. If $S_i <^S_{IH} S_j$ then $B_i <^B_S B_j$.

4.2.6 Local Database History

We define a local database history in a manner similar to that of [BHG87], page 29. A local history contains a set of n local transactions $T_1 \ldots T_n$. Each transaction T_i is represented in the local history by a partial order of local steps $<_i$. These steps form a partial order

$$LH = \left(HE_L, <_{LH}\right),$$

where

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

The local database history is assumed to be serializable. Its *local serialization order* is the partial order of the equivalent serial execution. We can derive the local serialization order for a local database from its history, by examining how the transactions relate based on the relationships between the transaction operations on the local database. This is because conflicting operations define where conflicts between transactions occur. Formally, the local serialization order is a partial order:

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

where

1. T is the set of all transactions (global subtransactions and independent transactions) in the local database, and
2. if $\{S_i, S_j\} \subseteq HE_L$, $S_i \in T_i$, $S_j \in T_j$, and $S_i <_{LH} S_j$, then $T_i <^T_{LS} T_j$.

4.2.7 Merged History

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

$$MH = \left(HE_M, \rightarrow_{MH}\right)$$

where

1. $HE_M = \cup_{L \in \mathrm{MDB}} T_L$ (where T_L is the set of transactions that ran on local database L),

2. if $T_a <_{LH} T_b$ in some local history LH, then $T_a \rightarrow_{MH} T_b$, and

3. if T_i and T_j are global subtransactions from respective atomic blocks B_i and B_j in IH and the steps of B_i precede those of B_j in $<^S_{IH}$, then $T_i \rightarrow_{MH} T_j$.

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

Note that $\rightarrow_{MH}$ is not necessarily a partial order, because if two steps are executed in the same local database, their execution order in the local history may not be the same as the completion order of their steps reflected in the InterAction Manager history. We presume that a correct merged history should not contain this kind of anomaly. Intuitively, this is because $\rightarrow_{MH}$ should define the global serialization order of the atomic blocks in the merged history.

4.3 Correctness Requirements

Theorem 4.1 *A set of atomic blocks and independent transactions in a merged history for some multidatabase is* serializable *if and only if the following conditions hold:*

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

2. *All subtransactions involved in the same atomic block are committed atomically.*

PROOF: Define the *merged serialization graph* as the graph constructed by taking the union of all of the (acyclic) local serialization graphs and merging all nodes that represent global subtransactions in the same atomic block. The resulting graph contains one node for each atomic block in some InterAction, and one node for each independent transaction in some local database. It contains all the information concerning serialization restrictions from all the local databases.

First, consider what happens if an atomic block does not commit atomically. Assume atomic block B accesses data in two local databases X and Y using subtransactions X_a and Y_a, respectively. Assume without loss of generality that the commit of X_a happens before the commit of Y_a. Y_a may then later abort, ultimately causing X_a to abort as well. However, the results of X_a were visible in X for a span of time, violating its atomicity constraints, and therefore also violating serializability.

Now, assume all atomic blocks are committed atomically. Examine the merged serialization graph. We know[1] because the local histories are serializable that no cycles are formed in the merged serialization graph that are induced by a single local database history. Therefore, they must involve multiple local database histories. However, if all atomic blocks are serialized on the local databases in the same order (the specified global serialization order), then all paths spanning multiple atomic blocks in the merged serialization graph must reach those atomic blocks in an order consistent with the global serialization order. Therefore, there are no cycles in the merged serialization graph that were induced by the merging process. Therefore, the merged serialization graph is acyclic, and the merged history is serializable.

Now, assume that there is a cycle $N_a, N_b, \ldots, N_n, N_a$ in the merged serialization graph. This cycle must involve edges induced by multiple local histories. For each local database whose transactions are involved in the cycle, find a path in the graph induced by its history that contains all nodes representing its transactions in the cycle. This path traverses the nodes in the local serialization order for that database. Because there is a cycle, we know that at least one of these paths traverses at least two nodes in the opposite order from the global serialization order. Therefore, the local database represented by that path did not serialize its global subtransactions in an order consistent with the global serialization order. □

Lemma 4.2 *In a serializable merged history, $\rightarrow_{MH}$ forms a partial order.*

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

PROOF: The potential cycles in $\rightarrow_{MH}$ occurred because the local serialization order $<_{LS}$ for some local database did not order some set of global subtransactions in the same way the steps of their respective atomic blocks were ordered in $<^{S}_{IH}$. This does not occur in a correct merged history, because the merged serialization graph is acyclic. The direction of the arrows between nodes reflects the effective serial execution of the operations on the local database. Thus, $<_{LS}$ for each local database is derivable from the unique partial order $<^{S}_{IH}$, because it reflects the serialization order of the blocks in the InterAction Manager History. Therefore no cycles can form in $\rightarrow_{MH}$. □

Theorem 4.2 *For consistent information flow, the serialization order of the atomic blocks in an InterAction must be consistent with the information flow order of its steps. That is, if* $S_a \in B_a$ *and* $S_b \in B_b$ *and* $S_a <^{S}_{F} S_b$, *then either* $B_a <^{B}_{S} B_b$ *or* B_a *and* B_b *are the same atomic block.*

Since atomic blocks commit in their serialization order, this means that, no step that consumes information commits before the step that produces that information.

PROOF: If we examine this requirement in terms of ACTA dependencies [CR91], we note that if the consumer commits, the producer must commit also. This is because the consumer uses information generated by the producer, and its results are correct only if the producer's results are correct. Therefore, the consumer has an abort dependency on the producer, requiring the consumer to commit before the producer. □

Note that, for recoverability in case of failure, if information produced by one atomic block is consumed by another (i.e., the producer commits before the consumer), the InterAction must stably store the produced information.

5 Atomic Block Serializability and Commit

5.1 Problems with Enforcing Atomic Block Serializability

In the InterAction Model, a correct history for a multidatabase requires that the atomic blocks be executed serializably. In Theorem 4.1, we noted that if the merged history is serializable, then the atomic blocks must commit atomically, and the local database histories must all serialize the subtransactions that execute the different steps in an order consistent with some unique global serialization order.

In this section, we address issues of enforcing serializability in a merged history. To do this, we require the Agents to be able to examine, delay, and reorder all operations entering their respective local databases. These functions are not done capriciously; rather, they are done based on supplementary commands from the InterAction Manager. Because of their detailed spying capability, we call this type of agent a *Snoopy Agent (SA)*.

5.2 Enforcing Global Serialization Order

We have already noted that correct operation of the multidatabase requires that the subtransactions of the different atomic blocks be executed in a consistent order on all of the local databases. We enforce this property by having the InterAction Manager determine a global serialization order for the atomic blocks. It communicates this order to each of the Snoopy Agents for each of its local databases. Each Snoopy Agent is then responsible for maintaining this serialization order in its local database.

In this section, we define the various schemes that can be used by the Snoopy Agents to ensure that the serialization order of the global subtransaction in each of the local databases is consistent with the global serialization order dictated by the InterAction Manager. Note that since the multidatabase is heterogeneous, different data items in the multidatabase may be protected using different concurrency control schemes. Therefore, different types of Snoopy Agents may coexist in the multidatabase, one for each concurrency control strategy used by some local database. However, since the purpose of the Snoopy Agents is to enforce a consistent serialization order, and since we have shown that this (along with the need for atomic commit) is a sufficient requirement for a correct merged history (see Section 4), then the diversity of concurrency control schemes should not impact the ultimate consistency of the global database.

The approach we use for enforcing global serialization order in the local databases is the one used by Elmagarmid and Du in [ED90b]. The basic idea is to determine for each global subtransaction what its *serialization point* is. This is the point at which the transaction's position in the local database's serialization order is fixed by its concurrency control protocol. For example, strict 2PL transactions are serialized in commit order, so the serialization point for a transaction is identical to the commit point. We then use the Snoopy Agents to delay commands that would cause a global subtransaction to reach its serialization point until all subtransactions associated with atomic blocks preceding its atomic block in the global serialization order have reached their serialization points.

In the remainder of this section, we examine first a general approach that works on all types of concurrency control, but is very inefficient. We show that no more efficient general approach exists. Then we examine specific concurrency control strategies to see whether some tailored approach works for any of them. The different concurrency control strategies we examine here are described in [BHG87]. We first examine the strict variants of 2PL, timestamp ordering, and serialization graph testing. Then we show how these need to be modified for the basic variants (which are, in general, *not* recoverable), and for the relevant conservative variants (which predeclare their read and write sets). We make some comments about certifiers. Finally, we make some general statements concerning a scheme that works regardless of the underlying concurrency control mechanism, albeit at the expense of executing the global transactions worse than serially.

5.2.1 A General, but not Optimal, Approach

With any concurrency control mechanism, the global serialization order can be enforced by not allowing any subsequent global subtransaction to begin until after the current one has reached its serialization point. A general scheme would delay beginning a subsequent transaction until after the current one would have reached its serialization point *in any local database concurrency control scheme.* Thus, we can trade off concurrency of global subtransactions for generality.

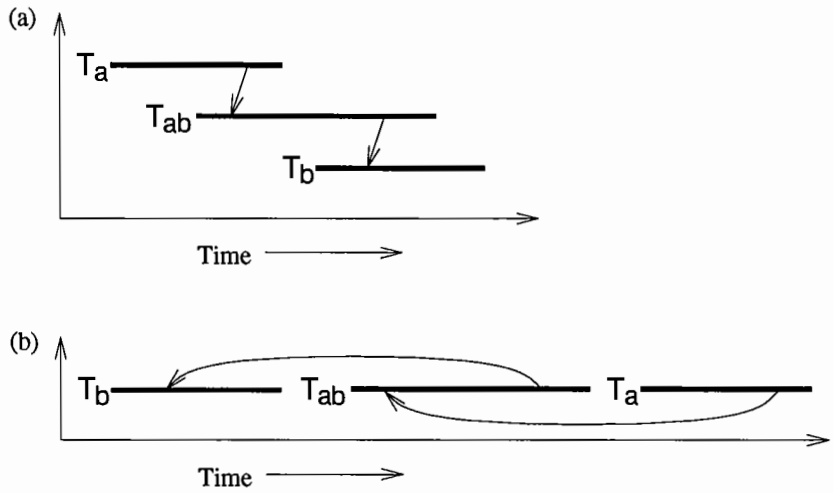


Figure 6: (a) Execution order. (b) Serialization order that differs from the execution order.

Unfortunately, the real problems occur when the local concurrency control schemes normally allow more concurrency, such as with serialization graph testing or timestamp ordering. We can show the following general theorem:

Theorem 5.1 *Given a sufficient supply of information in the database, an execution of any arbitrary number of transactions can be constructed in which the serialization order is inverted from the execution order.*

Proof: We will show this by induction. In the base case, suppose that we have an execution of three transactions T_a, T_b, and T_{ab}, where T_a commits before T_b begins. Let T_{ab} be a transaction whose execution overlaps the execution of both T_a and T_b.i This situation is shown in Figure 6(a); If the local database's concurrency control protocol allows T_{ab} to read from T_a, and similarly allows T_b to read from T_{ab} (for example, if it is based on SGT), then the serialization order would be T_b T_{ab} T_a, even though T_a completely precedes T_b in the execution order. In Figure 6(a) these reads-from relationships are indicated with arrows. The equivalent serialization order with the reads-from relationships still shown is in Figure 6(b).

Now for the induction step. Suppose we have a chain of transactions $T_a \ldots T_m$, all of whom ran serially with respect to each other, and have committed in the local database. Say their serialization order has been inverted by the execution of other transactions in the manner described in the base case. Now, say another transaction T_n begins, and there is also a transaction T_{mn} whose execution span overlaps those of T_m and T_n. Then if T_{mn} reads from T_m, and T_{mn} reads from T_n and *no other transactions in the execution*, then T_n will precede T_m in the serialization order. □

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

1. The previous global subtransaction has committed.

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

To ensure that these lulls occur, once a global subtransaction has committed, the Snoopy Agent may delay passing on any command that begins any new independent transaction or global subtransaction until all transactions that were concurrent with the global subtransaction have committed or aborted.

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

5.2.2 Strict 2PL Databases

In strict 2PL, a transaction takes locks on the objects it accesses up to the commit point, and all locks are released during the commit itself. Each strict 2PL transaction reaches its commit point at the end of its growing phase, when all of its lock requests have been granted. However, in practice, it is difficult to observe when a transaction takes its last lock. The Snoopy Agent's job is made simpler by observing that if two transactions overlap in the time they both have been granted all of their locks, their serialization order must be indeterminate. Therefore, we can use any point between the time the last lock is granted up to and including the time the transaction commits as the effective commit point. In strict 2PL, we choose the commit point, because it is easy to observe. Thus, we have that with strict 2PL, multiple atomic blocks can be running subtransactions in the same local database at the same time. However, if the commits are requested in the opposite order from the global serialization order, then the Snoopy Agent must hold the commit request for the later global subtransaction until the earlier one commits.

Note that if all of the local databases in the multidatabase use strict 2PL as their concurrency control strategy, then the InterAction Manager does not need to explicitly specify a global serialization order. Rather, it can enforce a global serialization order by the order that it commits the atomic blocks. As long as two atomic blocks that access the same local database do not commit at the same time, the multidatabase generates a global serialization order consistent with the commit order of the atomic blocks.

As a side note, one additional benefit for using this strategy with a set of 2PL local databases is that it avoids global deadlocks. Intuitively, a deadlock in the multidatabase that is not detectable locally involves some cycle that spans multiple local databases. If some global subtransaction T_1 is waiting (possibly indirectly) for a lock held by some other global subtransaction T_2, then eventually T_1 should precede T_2 in the global serialization order. A cycle would indicate that at least one local database is trying to produce a local history inconsistent with the global serialization order. The same mechanisms used for resolving the serialization order inconsistency also resolve the global deadlock.

5.2.3 Strict Timestamp-Ordered Databases

In strict TO, a transaction is issued a timestamp when it begins. When the transaction commits, the read and write sets of the transaction are checked to see that the operations in the transaction did not conflict with any uncommitted transaction with an earlier timestamp.

In this case, it is sufficient to ensure that the timestamps on the global subtransactions are ordered according to the global serialization order. Since timestamp are assigned when the transaction begins, it is sufficient to begin the global subtransactions in the global serialization order of their respective atomic blocks.

One problem with strict TO (and other TO schemes) occurs if the local database automatically restarts a transaction if it has aborted due to a conflict. In this case, the restarted transaction would have a new timestamp, which could change its serialization order in the local database. If this can happen, the Snoopy Agent would have to use the general scheme for enforcing the *GSO* that is described in Section 5.2.1.

5.2.4 Serialization Graph Testing Databases

In an SGT database, a serialization graph is maintained that makes the serialization order explicit. Nodes represent transactions, and edges represent conflicting operations in those transactions, specifically from the earlier operation in the history to the later operation. If an operation completes a cycle in the SG, the resulting history would not be serializable, so the transaction is aborted.

The problem with serialization graph testing is that the serialization point for a global subtransaction may occur outside of its lifetime [ED90b]. This is because arcs can be added from the node representing a transaction in the SG even after the transaction commits. With serialization graph testing, the general approach described in Section 5.2.1 is unfortunately the only way to ensure that the local serialization order is consistent with the *GSO*.

A second approach would be to try to shadow the serialization graph in the Snoopy Agent. This has the drawback that we do no assume a Snoopy Agent capable of examining the local database requests and responses to determine the read and write sets of all the transactions. Thus, we have to assume that every transaction depends on every committed transaction and all uncommitted transactions may depend on each

other, even though not all of these dependencies can hold in the local database. Unfortunately, we can show by an argument argument similar to the one we used in the general case that this method does not improve global transaction concurrency over the method described in the previous paragraph.

5.2.5 Basic 2PL, TSO, and SGT Databases

The basic methods of concurrency control differ from the strict methods in that they allow a transaction to read from another transaction once the transaction is finished with the data to be read. For example, a basic 2PL database transaction has a growing phase, when it takes locks, and a shrinking phase where it releases them. We know that the serialization point of a basic 2PL database is at the end of the growing phase. However, by the same argument we used with strict 2PL, we can choose the effective serialization point in a basic 2PL database to be the time of the first unlock. Basic TSO and basic SGT serialization points are chosen identically to the way they are chosen in their strict variants.

We see also that, with a basic scheme, there is a gap of time before the commit where the transaction has released some resources that other transactions can access. This means that basic databases are not recoverable [BHG87] and allow cascading aborts, so the Snoopy Agent needs to take some effort to ensure that committed global subtransactions stay committed. Therefore, in addition to enforcing the serialization order as required with the strict schemes, the Snoopy Agent also should delay passing on a commit for any subsequent global subtransaction in the global serialization order until all transactions on the local database that preceded the global transaction in the local serialization order have committed or aborted. This avoids having the abort of some later global subtransaction cascade to an earlier global subtransaction.

5.2.6 Conservative 2PL and SGT Databases

In the conservative schemes, each transaction is required to predeclare its read set and write set. The local transaction manager uses this information to ensure that all the resources the transaction needs are available before it executes any of the transaction's steps. Thus, once a transaction begins, it will not be aborted due to unavailable resources.

With conservative 2PL, the Snoopy Agent can submit a global subtransaction once all earlier global subtransactions in the global serialization order have executed their first instruction. At this point, we know that all earlier global subtransactions have completed their growing phase. If the new global subtransaction conflicts, its execution is blocked by the local database until it can obtain all of its locks. Otherwise, it is independent of any running global subtransaction, and therefore can always be serialized after all of them in the local database.

With conservative SGT databases, the Snoopy Agent can maintain an accurate shadow of the local serialization graph, because it knows each of its transaction's read and write sets at the time the transaction begins. Thus, the next global subtransaction in the serialization order can be submitted once it completes no cycles in the Snoopy Agent's serialization graph.

5.2.7 Certifiers

Certifiers do not check whether or not a transaction conflicts until the transaction commits. Then, using a procedure depending on the type of certifier, it decides whether or not the transaction conflicted with other active transactions. If so, it aborts the transaction.

With a strict 2PL certifier, the serialization point is the commit point (as it was with strict 2PL). Basic 2PL certifiers are difficult in that there is no way that the Snoopy Agent can tell if a transaction has completed its growing phase until it commits. Therefore, the Snoopy Agent must treat the transactions in a basic 2PL certifier as if it were a strict 2PL certifier.

SGT certifiers have the same problems as normal SGT does, in that the running history of operations is reflected immediately in the SG, and the serialization order is determined by when the arcs were added. They just don't check for cycles until a transaction commits. Thus, SGT certifiers must be treated identically to normal SGT.

As with the basic systems, non-strict certifiers do not necessarily generate recoverable histories or histories that avoid cascading aborts. Therefore, the Snoopy Agent must ensure with a certifier that the commits are processed in the global serialization order.

5.3 The Traditional Approach to Atomic Commit

In the InterAction Model, the procedures for committing an atomic block are identical to the procedures used for committing global transactions in a multidatabase that enforces all transactions to be executed serializably. Achieving an atomic commit of a global transaction in a multidatabase is known to be difficult [ME91][2]. This is primarily because the local databases cannot be modified to support a "prepared" state, and thus the normal distributed commit processes such as two-phase commit cannot be supported. Work that supports an atomic commit either severely restricts the structure of the global transactions [SBD+81a, EH88], or compromises local database autonomy [GP86, Pu88, BST90]. HYDRO [PRR91] circumvents the need to do either by using an architecture similar to our Snoopy Agent architecture, but requires every global and independent transaction to predeclare its read and write sets.

In this section, we describe how to implement a traditional two-phase commit process for atomic blocks in a multidatabase. We include a description on how to use the Snoopy Agents to emulate the prepared state in the two-phase commit when the local database does not support a visible prepared state.

5.3.1 Two-Phase Commit

With the traditional approach, we use a scheme for atomic commit that is very similar to the one found in [PRR91]. In this scheme, the Snoopy Agents and the local databases work together to emulate the two-phase commit. In particular, we use the Snoopy Agents to enforce a prepared state in the local databases that do not directly support one themselves.

At the InterAction level, when an atomic block enters the voting phase of the two-phase commit, it sends vote requests to each of its global subtransactions. It then waits for replies from each of those subtransactions before it continues with the commit process.

In the local database, we use the commit of a subtransaction on the local database to emulate the stable storage of the subtransaction's effects as it enters the "prepared" state in the two-phase commit process. Thus, a subtransaction in the prepared state is committed on the local database. Because of this, we require compensation if the atomic block eventually aborts. Therefore, we must not allow the atomic blocks to contain any steps that have uncompensatable side-effects (unless they only execute steps on a single local database, in which case they abort directly with no effects on the other local databases).

Note here that it would be incorrect just to maintain the prepared state by saving the changed pages in the Snoopy Agent's stable store and not committing the subtransaction on the local database. This is because the Snoopy Agent can only guarantee that the commit will proceed if the subtransaction's final state (committed or aborted) in the local database is known. However, if the local database uses a scheme such as timestamp ordering for concurrency control, then it is quite possible for the subtransaction to be aborted when it actually reaches the commit point in the local database. It may not be possible to re-execute the subtransaction (given what was saved in the Snoopy Agent), replacing its effects in the database, without wiping out the effects of other transactions that conflicted with the original subtransaction.

Once the subtransaction has successfully entered the prepared state, it can send its "yes" vote back to the InterAction Manager. At this point, the Snoopy Agent for the local database delays all other transaction commits until the InterAction completes its commit. This prevents any undesirable interference with the local database in case the atomic block is eventually aborted.

Once all of the subtransactions indicate that they are in the prepared state (i.e., all have successfully committed), the atomic block enters the decision phase by notifying each Snoopy Agent that prepared one of its subtransactions of the commit decision. At that point, the Snoopy Agent can pass subsequent delayed commits to its local database.

The message sequence described here is shown in Figure 7(a).

5.3.2 "No" Decision and Global Subtransaction Abort

Problems can occur with the two-phase commit if some subtransaction decides to abort. When some subtransaction aborts (as shown in Figure 7(b)), the InterAction Manager must abort the atomic block. Each subtransaction that has not yet entered the prepared state is directly aborted. Each subtransaction in the

[2]We have different conclusions in this work because we make different assumptions about the multidatabase architecture and its capabilities.

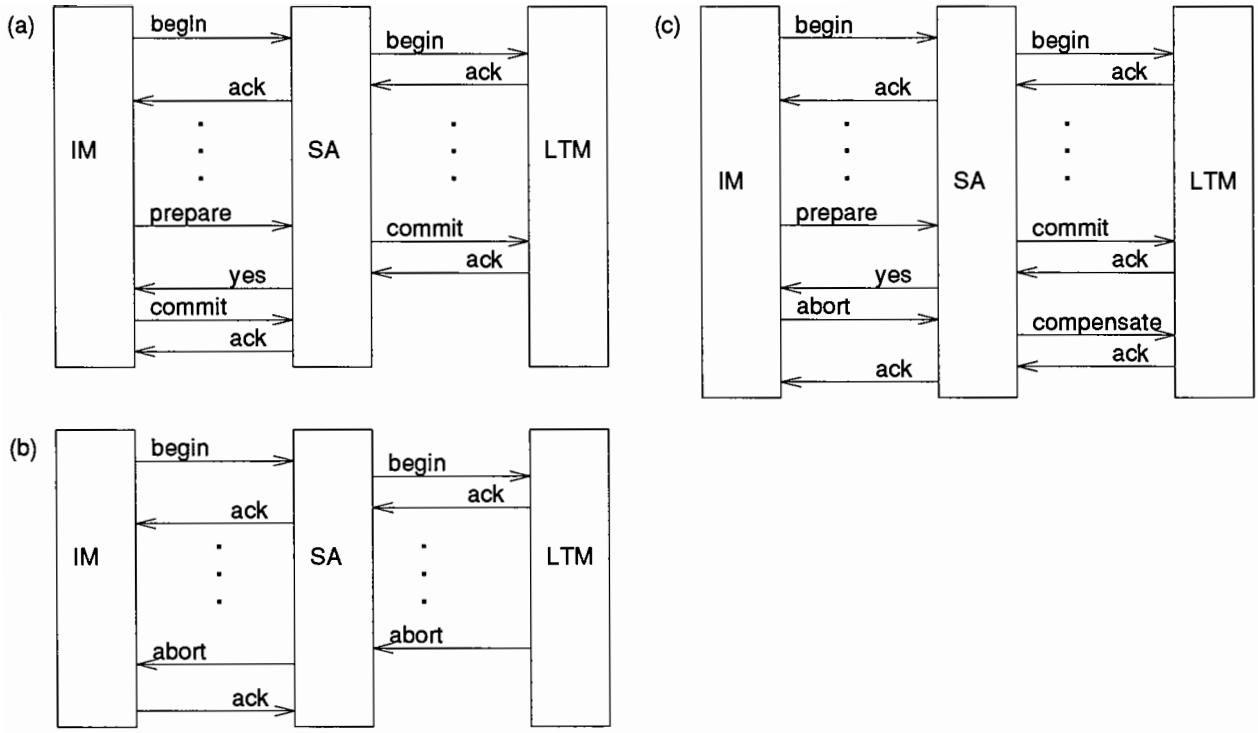


Figure 7: Message flows in the traditional case: (a) Standard commit, (b) Locally-initiated abort, (c) "No" decision.

prepared state is also notified that it must remove its effects from the local database. This is done by running a compensating transaction that writes the before-images of all pages that were updated by the committed transaction. Since there were no intervening writes (because all operation requests sent to the local database have been delayed by the Snoopy Agent), the compensating subtransaction correctly undoes the effects of the original subtransaction. This is because the history at the local database is (trivially) *sound*, as defined in [KLS90][3]. Once the compensating subtransaction succeeds, the Snoopy Agent can pass the operation requests that were delayed by the atomic block commit to the local database. This message sequence is shown in Figure 7(c).

It is sometimes difficult to determine what the before images of the subtransactions were. For example, if some database update is done declaratively through a query (e.g., "add 10% to all salaries of employees in the shoe department"), the before images would have to be retrieved by modifying the request beforehand to read the original salaries (e.g., "find all employees in the shoe department and return their keys and their salaries"). Once the result of the query is safely stored in the Snoopy Agent's stable store, then the original update request can be passed to the local database. This message exchange is shown in Figure 8. Given that the before images are safely stored, it is easy to generate compensating transactions. The Snoopy Agent simply needs to process the updates in inverse order, using the stored information to rewrite the original data back into the local database.

If we assume that the IM determines the place of an atomic block in the GSO at the time the atomic block begins, then aborting an atomic block has no effect on the serialization order of the other (concurrent) atomic blocks. There are two factors that ensure this. First, since $<_F^S \subseteq <_I^S$, then no other atomic block containing a step that consumes information produced by some step in this atomic block has begun yet. Presumably, the IM will adjust its execution of the InterAction once the abort of the atomic block is complete. Therefore, we only need to worry about subtransactions and independent transactions that may read the effects of the aborted atomic block in some local database.

[3]Korth et.al. define *sound* as follows: Let X be a history containing T, *dep(T)* (a set of transactions dependent on T that follow T), and its compensating transaction CT, with an initial state S. Let Y be a history with initial state S containing only *dep(T)*. The history of X is *sound* if $X(S) = Y(S)$. In our case, this holds because *dep(T)* is always empty.

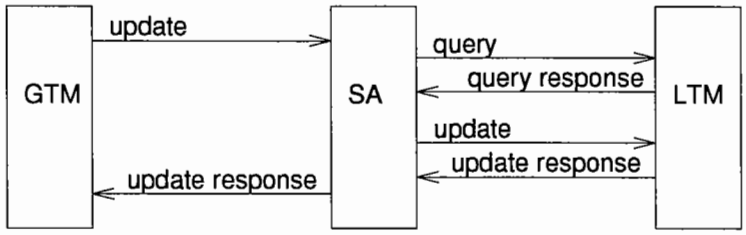


Figure 8: Message flows to determine before images when updating.

The second factor is that all concurrent conflicting atomic blocks are forced to serialize in the global serialization order. An atomic block or independent transaction that preceded the aborted one in the local serialization order never sees the effects of the aborted block. Since the histories of the local databases are sound, compensation exactly undoes the effects of the aborted block in each local database. Therefore, concurrent atomic blocks and independent transactions that follow the aborted block in the local serialization order do not see its effects. Therefore, the atomic block's abort preserves the atomicity constraint that none of the effects of the aborted block are visible.

5.3.3 Local Database Failure

There are two cases to consider where the local database fails. The first is when the local database fails but not the Snoopy Agent. The second case is when the Snoopy Agent fails as well. The second case is likely to occur if the system on which the local database and the Snoopy Agent reside goes down.

In the first case, any global subtransactions not in the prepared state abort when the local database comes online again. This causes a subtransaction abort, and is handled as described in the previous section. If there is a subtransaction in the prepared state it is not touched, because if the atomic block eventually commits, the processing of the commit is independent of the local database. However, the Snoopy Agent must delay all commits until after the subtransaction has completed the two-phase commit. If the atomic block eventually aborts, then the Snoopy Agent must submit the compensating subtransaction as soon as the local database completes its recovery process, and before further independent transactions and subtransactions are begun.

Also in the first case, the Snoopy Agent must explicitly transmit "aborted" messages to all clients that have uncommitted independent transactions. This ensures that the clients that submitted the independent transactions correctly recover from the local database failure, as they do not see the failure of the local database directly.

If the Snoopy Agent fails along with the local database, the InterAction Manager and the independent clients notice the failure and assume that all uncommitted subtransactions on the local database abort. They will then recover from the situation as appropriate to their specific applications. When the Snoopy Agent comes back online, it should ensure that all aborts were processed correctly before it continues normal operation.

Once both the Snoopy Agent and the local database are online again, the Snoopy Agent should retransmit the "yes" vote for any subtransaction in the prepared state, to ensure that the InterAction Manager was correctly informed of the vote. While waiting for the InterAction Manager to reply, it continues to delay other commits (because some global subtransaction is in the prepared state). In the meantime, the InterAction Manager returns its decision to commit or abort normally. In this case, the InterAction Manager may have become impatient and aborted the atomic block already.

5.3.4 Snoopy Agent Failure

If the Snoopy Agent fails but not the local database, then all communication with the local database halts. This looks to the InterAction Manager and the independent applications as if the database were down, and they should ensure that their uncommitted transactions on the local database aborted. To the local database, this looks like all of its clients failed, and it aborts all of their transactions. When the Snoopy Agent comes back online, it must ensure that everything was aborted correctly before it continues normal

operation. If this is in doubt, the Snoopy Agent can explicitly issue abort requests to the local database, and abort notifications to the clients.

5.3.5 InterAction Manager Failure

If the InterAction Manager fails, the Snoopy Agent must abort all subtransactions that are not a part of committed atomic blocks, because clients initiating the InterActions see the failure and assume that the atomic blocks abort. Aborting uncommitted subtransactions and independent transactions is straightforward, but aborting subtransactions that are in the prepared state means that a compensating subtransaction must be initiated. This is done in the manner described for the "no" decision case.

5.4 The Best Effort Approach to Atomic Commit

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

In this commit scheme, we take a "best effort" approach to trying to commit an atomic block. That is, once the InterAction Manager has specified that some atomic block should commit, it works together with the Snoopy Agents to make an effort to ensure that all of its subtransactions' effects persist. Thus, the response to a "no" vote is an atomic block abort only if there is no reasonable way for the subtransactions' effects to be committed in the local database. For example, an atomic block might abort if one of its subtransactions aborted, and immediately afterwards, the local database failed.

This commit process is better than the previous one in that it tries to prevent system aborts of subtransactions from affecting the entire InterAction. However, it is a more complex process. It requires longer to run, even when the commit proceeds smoothly.

5.4.1 Atomic Commit

In this situation, the commit of an atomic block may not occur all at the same time. This is true, for instance, in the situations where one of the subtransactions is aborted at commit time, and the InterAction Manager tries to redo its work. This causes the re-execution of an entire transaction on some local database in the middle of the atomic block commit process.

To understand the best effort commit, we first need to examine information flow between the subtransactions in the atomic block being committed. Let the *subtransaction information flow (SIF)* be a relationship

$$(T, \rightarrow_F)$$

where

1. T is the set of subtransactions in the atomic block, and
2. $T_a \rightarrow_F T_b$ if subtransaction T_a and T_b in T contain steps S_a and S_b, respectively, and $S_a <_F^S S_b$.

Note that the relationship $\rightarrow_F$ may contain cycles.

In the best effort scheme, the basic idea is to commit the subtransactions in the atomic block according to the subtransaction information flow. If some subtransaction aborts and has to be re-executed, any subtransaction following it in the information flow order must also be aborted (because it depends on incorrect results). If some such subtransaction has committed, then it must be compensated for. It is much simpler just to hold that subtransaction commit until we are sure that it will only be aborted if the entire atomic block aborts.

The best effort atomic commit scheme is similar to a two-phase commit, except that the voting phase proceeds in an order defined by the subtransaction information flow. If the subtransaction information flow forms cycles, then the subtransactions in the cycle vote together in the same manner as the one used in the traditional scheme (i.e., one "no" vote counts to abort the whole set of subtransactions together).

To be more specific, during the voting phase the InterAction Manager first asks for votes from those subtransactions (or subtransaction cycles) that have nothing preceding them in the *SIF* for the atomic block. Subsequently, it requests votes from the other subtransactions or subtransaction cycles as they reach the state where all subtransactions directly preceding them in the *SIF* have voted "yes". Once all subtransactions have given a "yes" vote, the atomic block precedes to the decision phase. The decision phase is identical to that of the traditional approach.

As with the traditional approach, commits of atomic blocks that access completely disjoint sets of local databases can proceed simultaneously. However, if two atomic blocks overlap in the sets of local databases they access, then their commits must be done sequentially in their global serialization order. The reasons for this are explained in detail in Section 5.4.4.

5.4.2 "No" Decision and Global Subtransaction Abort

With the best effort scheme, a response to a "no" decision or to a subtransaction abort is to try to replace the effects of the aborted subtransaction(s) in the local database. Thus, in these cases, the Snoopy Agent notifies the InterAction Manager of the subtransaction(s) that were aborted, and requests that they be re-executed. The InterAction Manager must be involved in this process because the abort can affect other subtransactions on other local databases.

If one or more subtransactions aborts, then any subtransactions (on other local databases) that follow them in the subtransaction information flow could possibly be affected by the aborted subtransaction. Therefore, all of those subtransactions must be aborted by the InterAction Manager, and re-executed using the results produced by the new subtransaction.

If the InterAction Manager is asked to re-execute the same global subtransaction too many times, it may choose to abort the entire atomic block rather than continuing to attempt the forward recovery. It then proceeds with the abort as if this were a "no" decision in the traditional approach.

5.4.3 Local Database, Snoopy Agent, or InterAction Manager Failure

From the point of view of process failure, the best effort scheme looks like the traditional scheme, except that the voting phase in the two-phase commit is prolonged. Therefore, the same recovery strategies that are used in the traditional approach can be used here as well.

5.4.4 Abort, Failure, and Global Serialization Order

In the best effort approach, once some subtransaction fails and efforts begin to try to bring its atomic block to a commit point, the recovery procedures may impact the efforts to enforce a global serialization order. In particular, there are several situations where the recovery process begins new subtransactions. These include when the local database decides to abort a subtransaction for system reasons (e.g., deadlock resolution), and when the InterAction Manager needs to abort a subtransaction to preserve correct information flow in some InterAction (see Section 5.4.2). In these cases, it may be difficult to ensure that the new subtransactions take an appropriate place in the serialization order. Thus, subsequent active atomic blocks may be affected by the efforts to commit the current atomic block.

We explore the effects of attempting to continue an atomic block on the subsequent atomic blocks, from the point of view of maintaining the serialization order. Note here that, in all cases, an atomic block must complete its commit *before* any conflicting[4] atomic block that follows it in the global serialization order initiates its commit. This is because new subtransactions may be initiated as a result of an attempt to re-execute an aborted global subtransaction. Since this may affect subsequent conflicting atomic blocks, those cannot commit until the earlier ones are complete. Therefore, they cannot yet return a "yes" vote in the voting phase. This situation did not exist in the traditional scheme, because no new subtransactions are initiated for a atomic block once it initiates its commit or abort.

When some subtransaction aborts, all subtransactions (of other atomic blocks) that follow it in the global serialization order, and that have already reached their serialization points, must abort. This is because the

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

re-executed subtransaction must maintain the atomic block's position in the global serialization order. Also, any subsequent transactions on the local database may conflict locally with the re-executed subtransaction. If the re-executed subtransaction fails or is blocked because of resource conflicts, other transactions on the local database may need to be aborted as well.

6 InterAction Execution and Synchronization

The InterAction Manager is responsible not only for ensuring that the steps of each InterAction succeed in an appropriate order, but also for ensuring that the InterActions use transactions and Snoopy Agent processes on the local databases in such a way that the ensuing execution 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 Snoopy 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.

6.1 Scheduling Actions and Steps

At any specific time, the different in-process InterActions in the database have one or more concurrent threads of execution 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 is currently waiting for some event to occur (if the next statement is a *wait* statement) or for some set of concurrent execution threads to terminate (if the next statement is a *join* statement) before it can continue execution. When the event occurs and is detected (see Section 6.3 for details), then the InterAction thread is moved to the active set.

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 a separate execution thread. The *fork* statements in an InterAction indicate when two of its different subtasks can be scheduled concurrently. This is done by forking threads for the concurrent subtasks. Thus, at any given time, each concurrent thread in the InterAction manager's active set is scheduling the steps to do its own private depth-first search to best accomplish its own private subtask. Any conflicts among these subtasks are mediated in the local databases using their concurrency control schemes, and by the Snoopy Agent processes on the local systems by controlling the submission time of the steps and the global subtransactions so that the local serialization order conforms to the global serialization order.

The following is the algorithm for what an InterAction thread should do when it is at a specific point in its task where the next step is to choose and execute some action in the active set:

1. Try the highest-priority alternative action that has not yet failed. If more than one action is still untried at the current priority, then pick a random one to try next.

2. If all alternative actions have been tried and failed, then fail the last successful step in this execution. This causes recovery to occur, which executes a compensating step. The net effect is to cause the InterAction to back up a step in its depth-first search.

Attempting an action is done using the following algorithm:

1. Try the highest-priority step in the action that has not yet failed. This involves making a procedure call to some procedure in the step library. 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 abort if it was involved in a deadlock. In this case, the step may be allowed to fail, or it may be retried.

   (b) The step is not aborted, but the results are that the step failed to accomplish its subtask. In this case, the step fails.

   (c) The step is not aborted, and it accomplishes its subtask. In this case, the step succeeds and the action succeeds.

2. If the step/action succeeds, then the results and the information required for compensation must be recorded in some stable location.

6.2 Beginning and Ending Global Subtransactions

Recall from Section 4 that one of the jobs of the Snoopy Agent is to enforce a global serialization order among the atomic blocks of the different global transactions, so that the ordering is consistent across the different local databases. For this to work, the InterAction Manager must maintain serialization order information for each executing atomic block. As new atomic blocks begin, they are assigned their spots at the end of the serialization order being generated.

Assume that the InterAction Manager is attempting step S of atomic block B on local database D. When step S is begun, the following steps are taken:

1. If no place in the serialization order is assigned to B, assign one.
2. If B has no global subtransaction running on local database D, then begin a new global subtransaction. Assign the subtransaction B's serialization order. Inform the Snoopy Agent for local database D so that it can begin the new global subtransaction and update its records on the current global serialization order.
3. Submit S by submitting its procedure to the Snoopy Agent on the local database in the context of the global subtransaction.

When the last step of atomic block B is executed, the atomic block must commit. Note that, since $<_F^S \subseteq <_I^S$, the commit of the atomic block can proceed immediately, because all producers of information consumed by B either were in earlier atomic blocks that have already committed, or are steps executed in B and will commit atomically with the consumer steps.

There are two cases in deciding how to do the atomic commit:

1. B only executed one global subtransaction on one local database. In this case, the atomic block commit is accomplished by directly committing that global subtransaction.
2. B executed more than one global subtransaction. In this case, the subtransactions associated with B must be committed using one of the atomic commit algorithms defined in the previous section (Traditional or Best Effort).

6.3 Detecting and Processing Events

Events in the database are used to change the timing and/or the success of the current execution of an InterAction. Specifically, the *when* statement is used to delay the execution of an InterAction thread until the event occurs in the database. The *on* statement is used to detect undesirable events and abort/recover from those events.

Events in the database are specified as operations by specific sets of local transactions and/or global subtransactions. For instance, an event might be the cancellation of a plane reservation by anyone. The agent processes monitor the database for instances of those operations, and notifies the InterAction Manager when those events occur. The monitoring for an event is set up at the time that the *when* or *on* statement is executed. Monitoring can be done by examining the stream of operations input to the database for instances of the specific operations. Alternatively, if the events are restricted to waiting for changes, monitoring can be done externally by polling the database periodically to see if the change has occurred. If the event is set up for monitoring as a result of a *when* statement, then the monitoring only continues until the event occurs. After that, the InterAction thread continues execution, and the event is irrelevant. If the event is set up as a result of an *on* statement, it continues to be in effect until either

a. the statement specified by the label in its until statement is reached,

b. the InterAction terminates, or

c. the event is reached, the condition is TRUE, and the action is executed.

6.4 Concurrent Threads

When an InterAction thread reaches a *fork* statement, then a new execution thread is started for the forked process. This execution thread begins with a copy of the context of the forking thread, except that the new thread begins a new atomic block which is assigned its own place in the global serialization order. Once the *fork* completes, the new thread executes independently to the forking thread. This means that the two threads may ultimately conflict with each other, as specified in the InterAction definition.

When an InterAction thread reaches a *join* statement, it is placed in the waiting set until all threads that it is to join with terminate. If any of the joining threads explicitly fail, all of the joining threads are aborted. Otherwise, once all of the joining threads terminate, the thread with the *join* statement is placed back in the active set and continues its execution.

7 Related Research

7.1 Multidatabase Transaction Models

7.1.1 Work on Serializability without Wrappers

Most of the proposed models of transaction management in the multidatabase domain attempt to emulate traditional database transactions. That is, they require that the global transactions be atomic, and use serializability as the correctness criterion. [GP86] was the first work that really addressed the requirements that the local databases in a multidatabase must fulfill in order to support general, serializable global transactions. Unfortunately, many of these requirements cannot be satisfied if the local databases are allowed to maintain their autonomy. Examples of such requirements include the need to force a serialization order on the global subtransactions executed on a single local database, and the requirement for the local databases to support a prepared state so two-phase commit protocols can be used. [DELO88] looks at the problem from the point of view of autonomy, and concludes that if the autonomy of local databases is maintained by the multidatabase, then it is impossible to guarantee serializability for general global transactions.

Work that attempts to provide serializable global transactions in a limited way includes Superdatabases [Pu88], work by Breitbart et.al. [BST90], and some work by Elmagarmid et.al. [EH88, ED90b]. This work differs from ours in that it does not assume an architecture using modules like Snoopy Agents that are capable of delaying and reordering messages associated with the independent transactions on the local databases.

In his Superdatabases work [Pu88], Pu takes an optimistic approach to providing serializable global transactions. He requires each local database to provide the serialization order of its transactions to the global transaction manager. The global transaction manager uses these orders *(O-vectors)* to compose the local transactions into global transactions, and at the same time determine if any of the global transactions do not serialize. The other global transactions are correct, and may be committed. In this work, Pu indicates that commitment of atomic global transactions where the local databases have full autonomy in their local commit procedures is all but impossible.

In [EH88], Elmagarmid and Helal explore global transactions that are restricted in their function so that serializability can be enforced. For each global transaction, they determine the local databases that the transaction reads from *(read_sites)*, and the local databases that the global transaction writes to *(write_sites)*. They only allow global transactions whose read_sites and write_sites overlap by at most one site. They also use *STUB* processes to emulate a two-phase commit. These differ from Snoopy Agents in that they do not intercept the local database communications with the independent transactions.

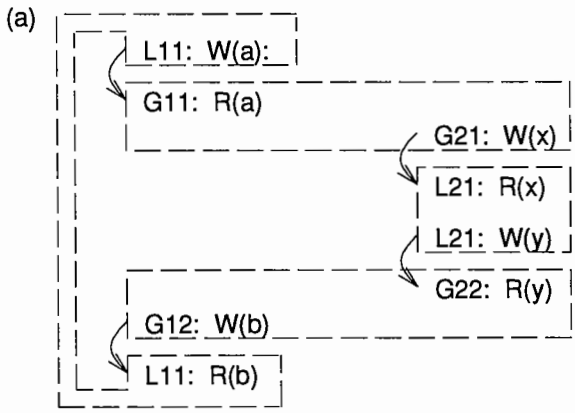
Breitbart et.al. [BST90] also explore issues of implementing serializable global transactions on multidatabases. They assume that the local databases maintain strict two-phase locking. They also assume that the local databases explicitly partition their data into two parts, one containing information that can only be updated globally, and one containing information that can only be updated locally. They provide a robust commit protocol for global transactions in this situation. We also provide a robust commit protocol, but do not require the local databases to be partitioned.

[ED90b] attempts to satisfy the requirement in [GP86] that the global database must be able to enforce a serialization order on the global subtransactions on a single database. This allows the multidatabase to enforce a common serialization order for global transactions across the different local databases. They explore different concurrency control mechanisms that could be used in the local databases, and describe various ways to use *STUB* processes associated with the local databases to delay submission of the global subtransactions in such a way that their serialization order is enforced. The concurrency control mechanisms they can do this for include two-phase locking, value date, and timestamp ordering. Unfortunately, with the exception of timestamp ordering, the solution usually involves submitting the global subtransactions almost serially. This work serves as a basis for our work on enforcing the global serialization order.

7.1.2 Work Using Wrappers for Atomic Commit

There are a few examples of how to use processes similar to Snoopy Agents to enforce concurrency control requirements in a multidatabase. These include HYDRO [PRR91] and the *A la carte* framework [DKB90].

HYDRO routes all transactions on a local database through a HYDRO server. It assumes that all independent transactions and global subtransactions predeclare the resources they need, though they also



(b)

	a RW	b RW		x RW	y RW
L11:	01	10			
G1:	10	00	G1:	01	00
			L21:	10	01
G2:	00	01	G2:	00	10

Figure 9: (a) Cyclic information flow problem. (b) ROLL structure for the problem.

allude to a scheme that does not make these assumptions. Given that they can make bit vectors to represent requests, they use a request-order linked list scheme to ensure serializability in the local databases. Their scheme for maintaining the prepared state in situations where the local database does not provide one directly is virtually identical to ours.

The approach HYDRO takes is generally reasonable, its major drawback being that it enforces predeclaration of resources for all global subtransactions and independent transactions. Also, if you separate out read and write permission in the request bit vectors, then serialization problems such as the one shown in Figure 9 (which is a simplification of an example by Elmagarmid et.al. in [ED90b]) will occur. This can be easily fixed, however, by setting a read bit to 1 as well as the write bit if the resource is to be updated.

A la carte is a framework, and as such, does not provide details on the internals of the functionality required in the code that "wraps" the local databases. However, they do provide some insights into other areas where this type of architecture is required, particularly in situations where the local databases do not supply all of the ACID requirements.

7.1.3 Work Relaxing Serializability of Multidatabase Transactions

Du and Elmagarmid [DE89] have developed a transaction correctness criterion for multidatabases called *quasi-serializability*. Quasi-serializability relaxes serializability in that it defines a correct history as one where each of the local databases has a local history that is serializable, and the history of the global transaction manger (which contains only the global transactions) is such that no two global transactions that run on the same local database executed concurrently. Extensive work has been done on implementing quasi-serializability in InterBase. Some of the stickier issues that they have addressed are presented in [ED90b, ED90a]. In his Ph.D. thesis [Bar90], Barker defines a concept of multidatabase serializability that is essentially identical to quasi-serializability.

Also in the context of their work on InterBase, Elmagarmid et.al. [ELLR90] offer a model to define the flow of a global transaction called a *predicate petri net*. This paper addresses transaction management issues such as function replication and non-vital functions. However, this work still assumes that global transactions execute quasi-serializably, and does not address recovery well.

7.2 Non-Serializable Transaction Models

This section describes some transaction models that, like InterActions, do not attempt to emulate serializability at all. Instead, they provide their own guarantees of correctness. While some of this work was done outside the context of multidatabases, the models described here provide useful insights into the problems that this thesis work addresses.

7.2.1 Sagas and Nested Sagas

Garcia-Molina and Salem [GS87] propose *sagas* and *nested sagas* as a way of ameliorating the problems associated with long-duration transactions. Sagas are sequences of traditional, short-duration transactions. The short-duration transactions operate in a traditional transaction environment, but the sagas themselves are allowed to interleave arbitrarily. However, sagas are atomic in the sense that they either execute all of their short-duration transactions or none of them. If one of the short-duration transactions in a saga aborts, forward recovery is attempted. If a saga aborts, compensation is used to "undo" its effects in the database. Recently, the saga model has been extended so that sagas can be nested [GGK+90].

7.2.2 ConTracts

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

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

7.3 Active Databases

Because InterActions use database events to pace themselves (as with the *when* and *on* statements), this work bears some relationship to the work on active databases. Active databases not only run transactions initiated outside the database, but also may react to events inside the database by triggering other transactions. Examples of active databases include HiPAC, DOM [BOH+91], and Cactis [HK86].

Dayal et.al. [DHL90] define an active transaction model for long-duration transactions called the *ECA Model*. Their model uses open nested transactions. It is a rule-based model as opposed to a procedural model. Each rule is defined as an event-condition-action triplet. When an event occurs, it triggers the evaluation of a condition. If the condition holds, then the action is executed. The different parts of the rule may be run within the same transaction (either immediately or at the end of the transaction), or they can be decoupled and run in separate transactions. This model also defines a way to express an order for the transactions to terminate. For example, one transaction may be constrained to terminate only after a specific earlier transaction has terminated.

7.4 Process Modeling

Process modeling attempts to model the flow of tasks in a system. Process modeling systems define specific processes, and how those processes flow from state to state as different tasks are completed. An example of process modeling is *Statecharts* [Har88]. Certain events and notifications can occur when the process is in a specific state. While not directly related to transactions in database processing, process modeling systems are similar to InterActions in that InterActions use patterns to define the flow of a task in a multidatabase environment.

The major difference between process modeling and InterActions is that InterActions assume that the different tasks use and share a persistent set of data. This data needs to be managed appropriately so that it remains consistent, and so that the different tasks do not interact inappropriately with each other through the shared data. Thus, InterActions allow the definer to use conflicts to specify the task's requirements for isolation from the effects of other, concurrent tasks.

7.5 Compensation-Based Recovery

Many transaction models define compensation as the basis for their recovery schemes. However, few papers have explored in detail the actual restrictions and implications of using compensation-based recovery. Also,

no really detailed algorithms have been described. Garcia-Molina and Salem [GS87] define compensation for sagas, and state that compensation means executing the compensating transactions in a saga in the inverse order of the execution of the original transactions. However, Waechter and Reuter [WR91] note that you can have a higher degree of parallelism during compensation than you did during execution, because the preconditions needed to execute the compensating transaction are already known. When you complete a step, you must have logged the information required to compensate for the step.

Korth et.al. [KLS90] attempt to formally characterize compensation-based recovery in the context of their own correctness criterion for design transactions (entity-wise serializability). They explicitly examine issues of dependencies among committed transactions and how they effect strategies for using compensation in recovery. They define a notion of a sound history, which is one where the execution of each compensating transaction CT compensating for some original transaction T does not disturb any of the outcomes of any transactions whose execution depends on T. They explore ways to approximate soundness when CT is not allowed to violate constraints that were upheld by the transactions dependent on T.

7.6 Distributed Programming Languages

The Argus programming language [LS83] developed at MIT attempts to address issues of managing persistent data in a distributed environment. Argus attempts to combine the programming notion of defining specific processes with the database notions of persistent, shared data and atomic, serializable access to that data. They also support non-atomic data types to increase concurrency.

The environment that Argus was designed for is really different from the multidatabase environment. Argus is defined to run on a heterogeneous distributed system, but the programmer defines both the format of the persistent data and the units in which it is distributed. Thus, the local data repositories really have no autonomy whatsoever. Issues of committing and aborting are handled completely within the Argus environment.

Another difference between InterActions and Argus is that Argus defines atomicity as a property of atomic objects in the database, whereas InterActions associate atomicity with certain portions of the procedure that accomplishes the InterAction's objective. These views of where atomicity should be enforced are really orthogonal to each other.

8 Conclusions

Planning applications can require access to diverse information stored in different databases. They also need not only to access that information on a one-time-only basis, but also to be aware of situations where the database evolves in a way that is relevant to the application's particular tasks.

In this paper, we present a multidatabase model that is tailored to support planning applications. Interactions that the planning application has with the databases concerning a specific task are executed as a single, identified procedure called an *InterAction*. The InterActions themselves are broken up into units that should be executed atomically, called *atomic blocks*. Information not only is sent and retrieved from the databases in the multidatabase, but also flows through the InterAction. Information read by earlier atomic blocks may be used to update other information in later atomic blocks.

InterActions take into account the need for understanding the evolution of relevant information in the multidatabase. Thus, and InterAction can set up an event handler and watch for particular events in the database. The occurrence of one of these events may indicate that somehow the InterAction's progress to date has been compromised, and may cause the InterAction to initiate specific recovery procedures.

We also propose a multidatabase architecture that allows the multidatabase more control on the order that requests and commands are fed to the database. This model allows us to delay and/or reorder messages going into the database as appropriate to the needs of the multidatabase. This feature allows us to support a fairly general multidatabase transaction model in a way that was not possible in earlier architectures.

Based on the InterAction model and our multidatabase model, we present a formal definition of multidatabases, InterActions, and correct execution. This definition serves as a basis for a synchronization algorithm, and later will guide the work on recovery.

References

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

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

[BOH+91] Alejandro Buchmann, M. Tamer Ozsu, 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.

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

[CR91] P K. Chrysanthis and K. Ramamritham. A unifying framework for transactions in cooperative and competitive environments. *Office Knowledge Engineering*, 4(1):3–21, February 1991.

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

[DELO88] Weimin Du, Ahmed Elmagarmid, Yongho Leu, and Shawn Ostermann. Effects of autonomy on maintaining global serializability in heterogeneous database systems. Technical Report CSD-TR-835, Purdue University ComputerSciences Department, 1988.

[DHL90] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Organizing long-running activities with triggers and transactions. In *SIGMOD Proceedings*, 1990.

[DKB90] Pamela Drew, Roger King, and Jonathan Bein. A la carte: An extensible framework for the tailorable construction of heterogeneous object stores. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice*, pages 239–252. Morgan-Kaufmann, 1990.

[ED90a] Ahmed K. Elmagarmid and Weimin Du. Maintaining hddbs consistency: The quasi serializability approach. Technical Report CSD-TR-1017, Purdue University Department of Computer Sciences, September 1990.

[ED90b] Ahmed K. Elmagarmid and Weimin Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of the 6th International Conference on Data Engineering*, pages 37–46, 1990.

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

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

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

[GP86] Virgil Gligor and Radu Popescu-Zeletin. Transaction management in heterogeneous database management systems. *Information Systems*, 11(4):287–297, 1986.

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

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

[GSW87] Irene Greif, Robert Seliger, and William Weihl. A case study of ces: A distributed collaborative editing system implemented in argus, 1987.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[HK86] Scott E. Hudson and Roger King. Cactis: A database system for specifying functionally-defined data. In *IEEE OODBS Workshop*, 1986.

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

[LS83] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[ME91] James G. Mullen and Ahmed K. Elmagarmid. On the impossibility of atomic commitment in multidatabase systems. Technical Report CSD-TR-91-029, Purdue University Department of Computer Sciences, April 1991.

[NRZ91] Marian H. Nodine, Sridhar Ramaswamy, and Stanley B. Zdonik. A cooperative transaction model for design databases. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan-Kauffman, 1991. to appear.

[NZ90] Marian H. Nodine and Stanley B. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In *VLDB Proceedings*, pages 83–94, 1990.

[OB90] M. Tamer Ozsu and Ken Barker. Architectural classification and transaction execution models of multidatabase systems. In *Proceedings of the Int'l Conference on Computing and Information*, pages 275–279, 1990.

[PRR91] William Perrizo, Joseph Rajkumar, and Prabhu Ram. Hydro: A heterogeneous distributed database system. In *SIGMOD Proceedings*, pages 32–39, 1991.

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

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

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

[WR91] Helmut Waechter and Andreas Reuter. The contract model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan-Kauffman, 1991. to appear.