Specifying Flexible Tasks in a Multidatabase [†]

(Research)

Marian H. Nodine
Computer Science Dept.
Brown University
Providence, RI 02912
mhn@cs.brown.edu

Noela Nakos
Oracle Corporation
500 Oracle Pkwy, Box MD40P12
Redwood Shores, CA 94065
nnakos@us.oracle.com

Stanley B. Zdonik
Computer Science Dept.
Brown University
Providence, RI 02912
sbz@cs.brown.edu

Abstract

Interactions are open nested flexible transactions used to define long-lived tasks that access multiple heterogeneous databases. Interactions operate on a multidatabase where the local databases are accessed via procedures called steps. *The steps in a local database encapsulate its information and allow it to be accessed uniformly.*

In this paper, we focus on issues of flexible task specification in a multidatabase. We define a language, TaSL, that allows a multidatabase user to define an Interaction to the multidatabase in terms of the steps provided by the local databases. TaSL allows an Interaction definer to specify sequences of steps that should be executed atomically (as classic transactions) in the multidatabase. The user also can specify constraints that must be maintained on the data for the Interaction itself to remain consistent. If these constraints are violated, committed transactions may need to be compensated for, so an alternative execution can be attempted.

1 Introduction

With the advent of high-speed communications, applications are being developed that access information in multiple, independent databases. These systems not only include support for gathering information from different databases operated by different organizations, but also integrating information within an organization that is stored in distinct, "legacy" systems.

The information in traditional databases is accessed from within transactions. The purpose of a transaction is to ensure the consistency of the information in the database by executing operations within one transaction atomically. However, this transaction model is problematic if we try to use it for accessing information in multiple databases in a consistent manner. For example, a user accessing multiple databases probably specifies longer transactions than those seen in a centralized system, if only because the transactions have to access a database remotely across a network. Also, the user may not have the same strict consistency requirements that atomic transactions enforce; it may be acceptable to do a portion of the task now, and a portion later. For instance, consider a task a travel agent might do – book a trip for a traveler. The traveler may not always choose the seat on his flight and order a special meal at the same time he makes his reservation. Yet a third problem occurs when the user is not fussy about how a specific task is accomplished, but will accept several alternatives. If a traveler wants to travel from Providence to Toronto, she may not be fussy about which airline she takes.

In this paper, we define a mechanism for specifying tasks that span multiple heterogeneous databases *(multidatabase tasks)*. We assume that a task is solely concerned with accessing a set of remote databases using their local, atomic, recoverable transaction interfaces. Each task accomplishes a specific goal, such as planning a specific trip. We provide an overview of our transaction model, *Interactions*, which defines the concurrency control requirements for the task.

Since the component databases of the multidatabase *(local databases)* are assumed to be autonomous, we need to define an interface that allows for interoperability. We provide this in two ways. First, we define a model for the uniform access of its local databases, called the *step model*. The step model basically assumes that the local database is encapsulated by a procedural interface. Additionally, it provides the mechanisms for automatic generation of *compensating transactions* if the Interaction aborts and its committed work must be semantically undone. Second, we define two processes for each local database that run on its machine on behalf of the multidatabase. The *Agent* coordinates the serialization and commitment of global transactions and executes their steps on the local database. The *Activator* detects conflicts that occur when a committed global transaction's effects are overwritten in a way that makes the Interaction internally inconsistent. For example, this could occur if a traveler's flight is canceled.

The focus of this paper is on issues of task specification and execution on a multidatabase. We

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

describe our task specification language, TaSL, for defining tasks on a multidatabase. TaSL assumes that each of the local databases has has a step interface. TaSL differs from many of the languages that support distributed, heterogeneous applications in that it focuses on the database issues of transactions and recovery. Thus, it groups steps on different local databases into a single (semantically) atomic transaction. These transactions are grouped into a partially-ordered set specifying how the task is accomplished. Different alternative sets can be defined to accomplish the same task in different ways.

TaSL supports the ability to backtrack when some transaction aborts and causes some subtask to fail. In this case, backtracking is required before the task can attempt an alternative execution. It can also define constraints on the data that must hold for the duration of the Interaction in order for its execution to remain consistent. Violation of these constraints will also cause backtracking to occur.

2 Overview of Interactions

Interactions are a flexible open nested transaction model for specifying long-lived tasks that span multiple databases[1]. These tasks are broken up into smaller atomic units, called *subtasks*. Open nesting [9] means that the different Interactions can interleave their subtasks arbitrarily; there is no concurrency control assumed beyond the fact that the subtasks are executed as atomic transactions.

Flexibility [6] allows equivalent alternative subtasks to be specified where possible, so that more than one set of subtasks can accomplish the overall task. With Interactions, flexibility is specified as alternative sets of global transactions. This differs from other models in that flexibility is implemented not within atomic global transactions in, but rather in terms of sets of inflexible global transactions. When one execution fails (i.e., some transaction aborts), compensation is used to back out the effects of that execution from the multidatabase until a consistent state is reached, so a different alternative can be attempted.

Unlike other existing models, Interactions also support a mechanism for regaining Interaction consistency if specific, necessary effects of a committed subtask are overwritten before the Interaction itself is committed. This type of overwriting is possible because of the open nested structure and the greater concurrency it allows. It causes a type of event to occur, which ultimately results in the failure of the (previously successful) alternative. When such an event occurs, the failed alternative is also compensated for and a new alternative attempted.

[1]A detailed description can be found in [15].

2.1 The Interaction Model

An Interaction is defined as a partial order of global transactions, each of which accomplishes some subtask of the Interaction's task. The atomic global transactions are ordered according to two criteria. Because of flexibility, two potential executions of the same Interaction may order the transactions according to which alternative should be attempted first. Also, within a given potential execution, a transaction may depend on the completion of a different transaction in order to perform its subtask correctly.

The state of an Interaction is kept in its *variables*. We assume that the variables are maintained in a *state database*, which is used exclusively for multidatabase management information. The different global transactions of the Interaction access and/or modify the variables. This is the means by which information is passed from one global transaction to another. The partial order of the global transactions reflects the execution dependencies among the global transactions (i.e. places where one global transaction must commit before a second can begin) as well as the order in which the global transactions of the Interaction read information in the Interaction variables from each other.

Task flexibility is represented in Interactions as alternative subsets of short atomic transactions, each of which accomplishes the same subtask. Exactly one alternative must have all its transactions commit for the task to succeed. These alternatives are prioritized, so the highest-priority subset that succeeds is the one whose effects are in place when the Interaction commits. If any other subset is attempted, its committed effects must be compensated for in the multidatabase before the Interaction itself commits.

2.2 Task Conflict and Task Isolation

Because of the arbitrary interleaving of global transactions in the different Interactions, as well as independent transactions on the local database, careful attention must be paid to isolation and to interference with the Interaction's task execution. Therefore, the Interaction model assumes two forms of isolation. The first, called a *strong conflict*, corresponds to traditional transaction atomicity – no conflicting interleaving operations are allowed. The second form is called a *weak conflict*. It defines a set of conditions set by the Interaction which should be preserved for a specific span of the Interaction's execution. This form of conflict is necessary because the other transactions running on the local databases change the information in those databases, potentially overriding information set by the Interaction and necessary for its correct

	Strong Conflict	Weak Conflict
Conflict Precedence	Transaction's Operation	Conflicting Operation
Scope / Duration	Single Atomic Transaction	Multiple Atomic Transactions (Single Interaction)

Figure 1: Summary of Strong and Weak Conflicts.

completion.

When a global transaction T_i updates a data item, and that value must remain constant after T_i commits for the Interaction to remain consistent, a weak conflict is associated with T_i to specify where to backtrack to if the data item is overwritten. For example, a weak conflict might be specified stating that once the traveler's hotel is booked, his reservation stays around until he actually completes his stay at the hotel. If, for instance, the hotel is blown down by a hurricane before the planned stay, the conflict is violated and the Interaction execution must be recovered from the point at which the hotel was booked. A simplistic way of accomplishing this would be to *compensate for* or semantically undo, all of the global transactions that depend on the invalid hotel reservation, then re-execute the Interaction from that point, choosing a different alternative.

The two forms of isolation and their differences are summarized in Figure 1.

2.3 The Execution Model

When an Interaction is executed, the best alternative set of global transactions is attempted, in the specified partial order.[2] Failure of a specific subtask may occur if either a subtransaction aborts, or the user initiates a subtransaction abort because of some logical failure. In this case, the remaining subtransactions of the global transaction are aborted. The most recent place in the execution where an alternative may be selected is located, and all global transactions that follow that point are compensated for in inverse order. Then execution continues from the current state as specified by the selected alternative.

An Interaction execution is assumed to be correct if its global transactions are serialized in the multidatabase in an order consistent with the defined execution order and also consistent with the order conflicts on the Interaction state were resolved. No concurrency control is enforced except for that necessary to ensure global transaction atomicity. Therefore, the global transactions of different Interaction executions can interleave arbitrarily in the multidatabase history (i.e., Interactions are open nested).

[2]Alternatives could be attempted concurrently, provided that it is possible to remove the effects of a lower-priority alternative if a higher-priority alternative commits.

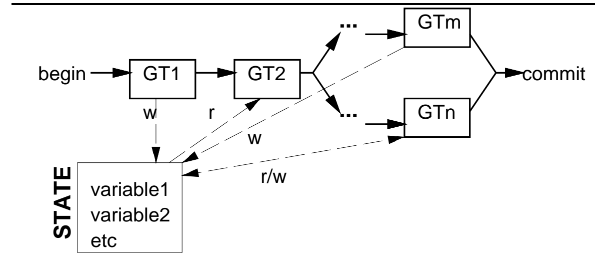

Figure 2: Interaction execution.

If a weak conflict is violated, the current alternative fails. In this case, other dependent parts of the Interaction may have committed since the invalidated transaction committed. Such invalid transactions include any transaction that has a predecessor that is invalid, or which read state from an invalid transaction. Once all of the invalid transactions have been located, the Interaction can regain consistency by compensating for all of its invalid transactions in the inverse order of their execution. Once a consistent state is regained, the most recent place where an alternative can be attempted is located, and further compensation must occur in the same manner as if the invalidated global transaction had been aborted. Once this is complete, a new alternative can be attempted.

Figure 2 shows an example of an Interaction execution. In this figure, the boxes labeled $GT1, \cdots, GTn$ represent global transactions on the multidatabase, each of which is assumed to execute atomically. The arrows between the boxes represent the execution dependencies among the global transactions; thus, $GT1$ must commit before $GT2$ begins, but the executions of GTm and GTn can overlap. Each of the different global transactions also accesses the Interaction state in some way; these accesses are represented by labeled dashed arrows indicating whether the information is read from or written to the Interaction state, or both. In this Interaction, if GTm writes some information read later by GTn, then GTm would have to precede GTn in the partial order.

2.4 Global Transaction Execution

We assume that each global transaction of some Interaction is executed as a set of *global subtransactions*, each of which executes on a single local database. The global transactions themselves span multiple local databases. However, to ensure that the global transactions are serializable, no two

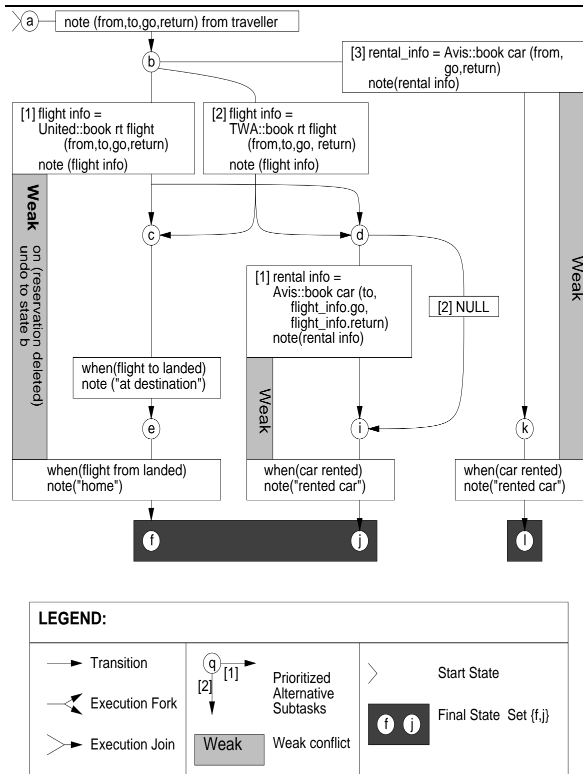

Figure 3: Task flow for a trip reservation.

global subtransactions of the same transaction execute on the same local database. To maintain global transaction atomicity, each global subtransaction commits atomically with its global transaction commit.

Since each global transaction in the Interaction is specified as an atomic set of procedure calls *(steps)* chosen from those made available by the local databases, each global subtransaction executes the subset of those steps that accesses its data. All steps in the subtransaction are executed as a single local database transaction. Thus, conflicts on information in the local databases are resolved by the local database concurrency control.

The subtransactions all commit atomically with the global transaction.

2.5 Example

Figure 3 gives a flow diagram for an example trip planning task for a travel agent. In this example, the traveler specifies information about the trip to a travel agent, who tries to make reservations for round-trip plane tickets and a rental car. If no appropriate flights can be found, the travel agent is just to book a rental car so the traveler can drive.

In this figure, states are represented by circles, with the start state indicated by a $>$. Transitions are indicated by arrows ($\rightarrow$), and are labeled with the global transactions that cause the transition. The legend gives a full description of the different symbols and what they mean.

This example shows the weak conflicts for the Interaction planning the trip. For example, once the United flight is correctly booked, that reservation must remain present in the United Airlines database until the return flight lands. If this is not the case – for instance, if the flight gets canceled – then the task must recover in order to remain consistent. Note that this means that the Interaction remains in existence until at least when the flight lands.

Weak conflicts in Figure 3 are represented as follows: The duration of each conflict is indicated by a lightly shaded box. The label associated with the box indicates an event that should occur during the span delimited by the box. The example weak conflict on the United flight ensures that the reservation stays around until the traveler actually takes the flight.

An example execution of this Interaction might be as follows: Note the information from the traveler. Make a reservation U on a United flight. Make a reservation A for an Avis car at the destination airport. Later, discover that the United flight was canceled. Cancel the Avis reservation A. Cancel the United flight reservation U. Try to book a TWA flight, but discover that it is full. Finally, make a second reservation B for an Avis car from home. Take the trip. Commit the Interaction.

3 Multidatabase Support

In this section, we describe the support the local database must provide for the local database to be able to participate in an Interaction execution. We also describe the Interaction support required at the global level of the multidatabase.

3.1 The Step Model

In the Mongrel multidatabase, each local database defines its own *Step Library*, which is the interface that the multidatabase can use to access the information in the local database. Each step is a procedure that can be executed on its local database by the multidatabase. Different steps can be composed together into a single local database transaction. The “begin”, “commit”, and “abort” operations on the local database are invoked explicitly and are distinct from the step invocations that query and/or update the local database.

Figure 4 shows a local database and its Step Library. Note that each step in the Step Library is explicitly paired with its compensating step, for easy correlation during backtracking. Since the step knows exactly what is run on the local

Step	Compensating Step
ReserveFlight(...)	CancelFlight(...)
CheckFlight(...)	DoNothing()
etc.	etc.

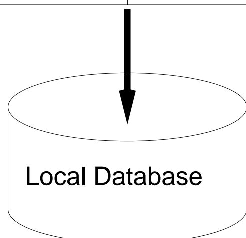

Figure 4: The step library interface between the local database and the multidatabase.

database, it also can derive and log what its compensating step call and arguments are. The compensating transaction may also log other information to be used by the compensating step: for instance, if the original step raised everyone's salary by 10%, the information logged may include exactly whose salary was raised, and by how much. During the step execution, this information can easily be derived with the help of some supplemental queries to the local database (within the existing local database transaction). Given the logged information concerning the compensating steps, a global transaction is semantically undone by executing its compensating steps in the inverse order of the original step execution.[3]

The step approach differs from the two existing approaches for integrating local databases, which are the restricted and the unrestricted approaches. Figure 5 summarizes these differences. The unrestricted model (e.g. [2]) allows global transactions in the multidatabase to run arbitrary sets of queries and updates in the local databases. Therefore, global transactions must include code in the various local databases' DMLs. The step model differs from the unrestricted model in that it only allows access to the local databases through the defined steps. This restriction provides a uniform way of accessing the information in the multidatabase, independent of any local database's DML. It also frees the multidatabase user from having to specify compensating subtransactions and ensuring that they are correct.

The restricted approach (e.g., [11; 1]) states that each time a multidatabase application accesses a local database, it runs a complete, predefined transaction on that local database. The step approach differs from the restricted approach in that each step is not itself a separate local database transaction. Rather, different steps on the same local database can be composed into a single atomic global subtransaction. Global subtransactions on different local databases can be grouped into a single atomic global transaction. Thus, the step model is more flexible than the restricted model.

	Restricted	Unrestricted	Step
Local Databases Encapsulated?	yes	no	yes
Global Transactions Span Multiple Local Databases?	no	yes	yes
Can Automate Compensation?	yes	no	yes

Figure 5: Comparison of different local database access approaches.

[4; 5] extend the restricted approach somewhat by allowing the steps themselves to be nested, and by allowing their execution to be decoupled from the global transaction execution. This extension allows for more flexibility, but their approach is orthogonal to the step approach.

3.2 Global and Local Levels

The multidatabase provides several functions at its global and local levels to support the execution of flexible tasks. At the global level, the multidatabase stores each Interaction definition and its execution state. As the Interaction executes, the multidatabase uses this stored specification to determine what alternative to try next. The global level also stores information on which alternatives were tried and failed, which alternatives succeeded, and which alternatives were never tried. If backtracking is required, it determines which new alternatives to retry, once the invalid work is compensated for.

The global level also provides a directory of the different steps available on the different local databases, and the arguments to the step calls. The multidatabase user can access this directory while defining an Interaction. At the global level the multidatabase also schedules and executes the global transactions, ensuring that each global transaction is atomic and that the Interactions are executed correctly.

The multidatabase has a local level extension called an *Agent* for each local database. It is responsible for representing the global transactions to its local database. This includes executing each global transaction's steps in the local database as a single local transaction, logging information re-

[3] For details, see [16].

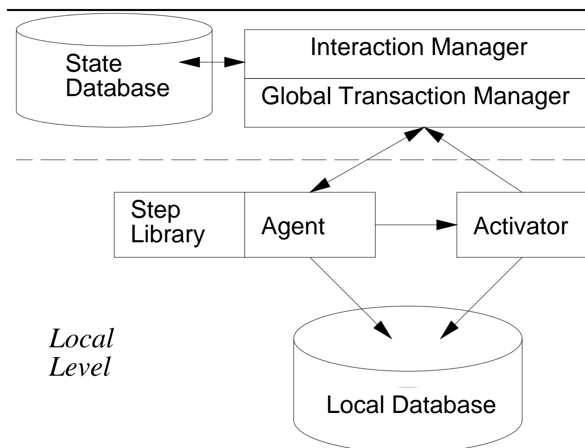

Figure 6: System modules in Mongrel.

quired for compensation, and taking the place of the local database in any global agreement protocols such as two-phase commit. The Step Library resides in the Agent, with its steps and compensating steps.

Because the Interaction model supports a notion of backtracking that requires noticing when committed effects are overwritten, the local level of the multidatabase also supports an *Activator* that monitors the state of the local database explicitly using queries. The Activator is notified of any conditions that must be maintained in the database and polls specific data items for changes that invalidate those conditions. This polling can be done periodically, although there are also more intelligent schemes that minimize the amount of polling queries generated by the Activator.

Both the Agent and the Activator properly reside on the local database's computer. It is these modules that allow the different, autonomous databases to interoperate.

An overview of the modules in the Mongrel system is shown in Figure 6.

4 Task Specification in TaSL

In this section, we give an overview of task specification in TaSL[4]. A complete definition of TaSL is given in Appendix A.

4.1 Interaction Definition

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.

Variable declarations are of the form $<type>$ $<name>$ or $<type>$ $<name_1>, \ldots, <name_n>$. A type is either the standard *int*, *char*, etc. used in C or some object view of information on some local database. Variables are used in the normal way. At any time, the results of a step can be assigned to an internal variable. This is done with a simple assignment statement:

$$< variable > = < step_result > .$$

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

The following sections define the structure of the body of an Interaction.

4.2 Alternatives

Flexibility is specified as choices between alternative global transactions for continuing from a given state if that state is reached during the execution. These alternatives are prioritized, so the highest-priority one is tried first and so on, until the one unique alternative path succeeds in accomplishing the remainder of the task. For example, in Figure 3, booking a United flight, booking a TWA flight, and renting a car are all different alternatives to be considered when the Interaction is in state b.

Subtasks in an Interaction may be specified as *vital* or *non-vital*. A vital subtask must be accomplished for the task to succeed. A non-vital subtask does not need to be accomplished. Non-vital subtasks are expressed using the $NULL$ alternative.[5] In Figure 3, the subtask that reserves the car at the destination *(from)* is non-vital, so there is a $NULL$ alternative (with priority [2]) from state d to state i. Thus, the car rental is tried first. However, if the car rental did not succeed, the $NULL$ subtask succeeds automatically.

In TaSL, the representation of alternative global transactions that can fulfill the task is represented using a modified form of the standard if-then-else construct. An example of the full form of the construct is as follows:

[4] Ta(sk) S(pecification) L(anguage)

[5] In other work [7; 6], subtasks were explicitly tagged as vital or non-vital. If a non-vital subtask could not be accomplished, it still succeeded. We instead provide an explicit $NULL$ alternative, because the non-vital construct is ambiguous in the presence of prioritized alternatives. For example, consider the case where there is a higher-priority non-vital alternative and a lower-priority vital alternative. If the first alternative fails, do you assume it succeeds or do you try the second alternative?

if not { $<global_transaction_1>$ }
then if not { $<global_transaction_{2a}>$ **or**
$\cdots$ **or** $<global_transaction_{2m}>$ }
then if not ...
then { $<global_transaction_n>$ }

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 the subtask executed by the alternative global transactions is not necessarily vital, then the last alternative should be **then NULL**.

4.3 Global Transactions

For correct operation in the multidatabase, each strong conflict must be enforced by executing steps within a single atomic global transaction on the multidatabase. This transaction prevents other global transactions on the multidatabase or independent transactions on the local database from inappropriately accessing the resources used by the steps it contains. For example, since a strong conflict is defined between the time the travel agent finds an available reservation for the traveler and the time the travel agent actually makes the reservation and updates the traveler's itinerary, there must be a single atomic global transaction that executes all of the necessary steps.

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

In TaSL, global transactions are defined using the *atomic* construct:

$$\textbf{atomic}\{ < body > \}.$$

Statements not contained explicitly in the body of some atomic block are executed individually in their own global transactions.

4.4 Steps

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

Each step is implemented as a procedure call on the *Step Library* associated with the local database:

$$< ldb_name > \textbf{::} < step_call > (< arglist >)$$

The operations of a step are 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 steps executed as a part of a single global subtransaction are also atomic and recoverable. The Step Library procedures generate the DML that is sent to the local database, and also formats the information returned from the local database into the form expected by the caller.

4.5 Events and Waiting

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

There are two ways we use events in an Interaction definition. The first way is to delay the execution of the Interaction until some event has occurred. For example, the travel agent may wish to delay the commitment of an Interaction until the traveler has completed his trip. An example of this is shown in Figure 3, where the Interaction waits for the flight to land before entering final state f.

The second way events are useful is in noting when weak conflicts are violated. Weak conflicts indicate the events that should not occur in the multidatabase during a specific execution span of the Interaction if the Interaction is to be able to continue executing from its current state. For example, a weak conflict might occur if some global transaction in an Interaction made a hotel reservation on a specific date, and that reservation was canceled (and its record deleted from the hotel database) because the hotel was closed.

Events in TaSL are represented as updates to a specific data item to a new value that violates some condition.

$$< ldb_name > \textbf{::} < step_call >< cond >< value >$$

The step call returns some single value from the local database, the condition *cond* is taken from the set $\{<, \leq, =, \neq, >, \geq, deleted\}$, and the value (when required) is some constant. The arguments to the step call may consist of constants or values read from the local database in which the event occurred.

When the event occurs, it is detected eventually by the Activator. The multidatabase can then process the event as needed.

The *wait* command is used when an Interaction wants to pause in its execution until some event

occurs. When a wait command is reached, the Interaction is blocked until the specified event occurs. Then the execution resumes with the statement after the wait. A wait command has the form:

wait(< *event* >).

Currently, we do not support the termination of the waiting period in the case where the event never happens, but rather expect that the user will notice if his Interaction is waiting forever. In the future, we may implement clock events as well, and allow the user to wait for an event, specifying a timeout period. If the timeout period expires, the wait terminates and the user receives a notification of the timeout event.

4.6 Weak Conflicts and Aborting

A weak conflict specification defines some event, and also which transaction to backtrack to when the event occurs. The event acts like a guard, watching the database for changes that indicate that might indicate that part or all of the Interaction's work has become invalid. Since weak conflicts do not indicate something that should (or could) be prevented necessarily, the conflicting operation is allowed to overwrite the Interaction's effects. Consequently the Interaction must recover its consistency by undoing some of its work and attempting a different alternative. The global transaction associated with the event specifies where to backtrack to during this recovery process. Thus, we treat a weak conflict event analogously to an explicit command to back out of the subset of the committed global transactions that has become invalid.

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

Because of this, TaSL treats weak conflicts more like exceptions than like locks. It uses a structure derived from Dayal *et al.*'s event-condition-action (ECA) rules [4] to define the exception. Since the weak conflict may be released at some point of time that is not the end of the Interaction, there must be an option to specify the ending time of the weak conflict. This is done using an *until* statement, which specifies the label indicating the statement that ends the weak conflict.

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

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

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

The event in the **on** statement is a standard event as defined earlier. The handler specifies a step to abort using the label associated with the global transaction that executed that step. At runtime the current global transaction identifier is associated with the label, and that identifier is passed to the Activator along with a specification of the event and condition.

When the Activator detects that some condition has been violated, the specified global transaction is invalidated. This does not necessarily mean that the global transaction must be aborted in the traditional sense, because likely it is no longer running. Rather, compensation must occur for that global transaction and all of its successors in the Interaction's partial order, as defined in Section 2.3.

4.7 Concurrency

We need also to allow unrelated subtasks to execute concurrently. For instance, when planning a trip, the hotel reservation and car reservation steps for a specific stop can happen concurrently once the plane reservation is made. We use the standard **fork** and **join** primitives 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 a global transaction, the new thread is assumed *not* to be a part of the same global transaction as the current thread. Instead, the new execution thread executes outside the scope of that global transaction, with its (execution) predecessor being the forking global transaction. The continuing thread continues executing the forking global transaction. The

one constraint on the execution of the Interaction is that the forked thread may not commit a global transaction until the forking thread's transaction has committed (for recoverability).

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

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

where $name_1$, ..., $name_n$ are names of concurrent threads. The thread that executes the join statement is called the *primary thread*, and the named threads are called the *joining threads*. When the join statement is reached, the primary thread first waits until all the joining threads are complete, then it executes the join statement. With Interactions, forks and joins 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.

4.8 Subroutines

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

$$[<type>]\ <proc>\ (<args>)\ \{<body>\}$$

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

A subroutine call is of the form:

$$[<variable>=]\ <proc>\ (<args>)$$

We assume that arguments are passed to the subroutine by value.

4.9 Summary

In this section, we have discussed the features we use to specify long-lived multidatabase tasks in a procedural manner. The TaSL language used to describe these tasks operates on a multidatabase that follows the step model for local database access.

Figure 7 shows much of the TaSL code for the trip reservation example in Figure 3.

5 Related Work

TaSL as a database language is the most procedural of the languages that support flexible transaction specifications for databases. lso, unlike any of the languages described below, TaSL is the only language the explicitly supports being able to note when some committed global transaction (subtask) has been invalidated (i.e., its effects were overwritten in a way that compromised the overall task's consistency before the task itself completed), and being able to recover by backtracking to that point and trying alternative subtransactions. Other sys-

```
begin_Interaction {
  place from, to;
  date go, return;
  flight * go, * return;
  car_rental * rental;

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

  if not {                          /*United first*/
    atomic {
      book_UN_flights(from,to,go,return);
      fork (T2) {
        atomic { book_Avis_car(go,return,to); }
        car_rented: wait (Avis::status(rental.resno)
              ==RENTED);
      }
    }
    wait (UN::status(return.flight)==LANDED);
  }
  then if not {                     /*Try TWA*/
    atomic {
      book_TWA_flights(from,to,go,return);
      fork (T2) {
        atomic { book_Avis_car(go,return,to); }
        car_rented: wait (Avis::status(rental.resno)
              ==RENTED);
      }
    }
    wait (TWA::status(return.flight)==LANDED);
  }
  then if not {                     /*Rent a car?*/
    atomic { book_Avis_car(go,return,from); }
    wait (Avis::status(rental.resno)==RENTED);
  }
  then abort_Interaction;           /*No go.*/

  commit_Interaction;               /*Waits done*/
} /*End Interaction*/

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

Figure 7: Interaction definition for trip reservation example (incomplete).

tems such as Narada [10] and Flex Transactions [6] support backtracking after subtask failure (abort or logical failure), but not after a failure due to an external overwrite.

ACTA [3] is a formalism than can be used to specify the types of dependency found among the different subtasks of a task. IPL [2] specifies the subtransactions and dependencies independently, separating out issues of local transaction definition from the specification of the overall task flow and preferences. IPL also follows the Flex Transaction model [6], which implements flexibility within the context of nearly-atomic global transactions. In contrast, TaSL specifies flexibility as a choice between different global transactions.[6] ConTracts [18] define a scripting language that supports similar features to TaSL. It has a notion of exit invariants similar to that of weak conflicts, but enforced as predicates that must hold true when a new subtask is initiated. With Interactions, weak conflicts are detected independently of subtask invocations. Both IPL and the ConTracts scripting language use a dependency structure similar to that defined by the ACTA framework.

VPL [12] is a logic-based language that supports flexible multidatabase transactions. In contrast, TaSL supports similar functionality using explicit control flow and prioritization. Interestingly, the local database access model assumed by VPL resembles the step model presented in this paper.

ATM [5] is a work-flow system that allows for nested and decoupled activities on multidatabases. Its structure is based primarily on ECA-rules. DOL [17] is a multidatabase workflow language that supports transactions explicitly. It also supports tasks that have portions that do not execute on local databases, and thus have portions that do not follow transaction semantics. The multisystem task specification environment Narada [10] provides an environment for executing tasks specified in DOL.

Other transaction models that support flexible transaction definitions include Sagas and Nested Sagas [7; 8], Flex transactions [6; 13], ConTracts [18], and ATM [4; 5]. The open nested model of Weikum and Schek [19; 20] supports open nesting but not flexibility.

6 Conclusions

In this paper, we discussed a transaction model, Interactions, that can be used to support flexible, long-lived transactions on a multidatabase. In particular, the Interaction model addresses the following issues:

- Breaking up a potentially long-lived task into a set of related shorter tasks each of which can execute atomically on the multidatabase.
- Specifying constraints that must hold on the multidatabase for the duration of the long-lived task in order for the shorter tasks to be valid and consistent among each other.
- Specifying tasks flexibly; that is, giving alternative ways to accomplish the task.
- Specifying priorities among the alternatives so the multidatabase can find the best feasible alternative. This implies that the multidatabase must support some ability to search through the different alternatives.
- Specifying backtracking properties, so that if some constraint is violated and the task becomes inconsistent, the multidatabase can compute what backtracking needs to be done. This also allows the multidatabase to determine, once the backtracking is complete, how to continue the search for the best possible alternative.

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

Interactions expect the local databases to present a procedural interface for use by the multidatabase. In particular, each local database presents a set of procedures, or *steps*, that the multidatabase applications can use to access its data. Each step has an associated compensating step, and knows how to log the information required for compensation. Steps in different local databases may be grouped into a single global transaction. This interface is less restrictive than the *restricted access* schemes, where each step is executed as a complete, atomic transaction. It is more restrictive than the *unrestricted access* multidatabase schemes, in that the multidatabase does not have unlimited access to the information in the multidatabase. However, steps do make an easy and intuitive way to specify the task at the global level, as they encapsulate the details of the local databases.

Based on the step model for multidatabase access to the local databases, we defined a task specification language, TaSL. TaSL is a procedural language that allows the task definer to specify prioritized alternative sets/sequences of global transactions, any one of which can be used to accomplish the task.

[6] Flex Transactions restrict an overall task to have at most one subtransaction per local database, for serializability reasons. Interactions allow multiple subroutines on a single local database, but only one within a given global transaction.

The priorities guide which of the alternatives is selected in the case where more than one actually could successfully complete the task.

In addition to allowing the task definer to specify tasks in terms of steps grouped together into global transactions, TaSL also allows the declaration and use of variables internal to the Interaction. Since these variables may be accessed by concurrent global transactions of the Interaction, their use is synchronized in the same way a local database synchronizes access to its data.

TaSL also allows the user to define constructs that allow committed global transactions in the Interaction to be backed out of if their work is overwritten or otherwise made invalid before the Interaction itself commits. This type of conflict, termed a *weak conflict*, cannot be specified in other languages.

Along with others, we have implemented a prototype multidatabase, Mongrel, that contains several local databases that maintain travel information. These local databases have a step interface. We have partially implemented the TaSL language to run on Mongrel.

References

[1] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency control and recovery for global procedures in federated database systems. *IEEE Data Engineering Bulletin*, 10(3), 1987.

[2] Jiansan Chen, Omran Bukhres, and Ahmed K. Elmagarmid. IPL: A multidatabase transaction specification language. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, 1993.

[3] P. K. Chrysanthis and Krithi Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *SIGMOD Proceedings*, pages 194–203, 1990.

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

[5] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. A transactional model for long-running activities. In *Proceedings of the 17th VLDB*, pages 113–122, 1991.

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

[7] 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.

[8] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling long-running activities as nested sagas. *IEEE Data Engineering Bulletin*, 14(1):14–18, March 1991.

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

[10] Y. Halabi, M. Ansari, R. Batra, W. Jin, G. Karabatis, P. Krychniak, M. Rusinkiewicz, and L. Suardi. Narada: An environment for specification and execution of mulit-system applications. In *Proceedings of the 2nd International Conference on Systems Integration*, 1986.

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

[12] eva Kühn, Franz Puntigam, and Ahmed K. Elmagarmid. Multidatabase transaction and query processing in logic. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan-Kauffman, 1991.

[13] Y. Leu. *Flexible Transaction Management in the InterBase Project*. PhD thesis, Purdue University, August 1991.

[14] Marian H. Nodine. *Interactions: Multidatabase Support for Planning Applications*. PhD thesis, Brown University, 1993. (Also Brown University Computer Scinece Department Technical Report CS-93-17).

[15] Marian H. Nodine. Supporting long-running tasks on an evolving multidatabase using Interactions and events. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 125–132, 1993.

[16] Marian H. Nodine and Stanley B. Zdonik. Automating compensation in a multidatabase. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, 1994.

[17] M. Rusinkiewicz, S. Ostermann, A. Elmagarmid, and K. Loa. The distributed operation language for specifying multi-system applications. In *Proceedings of the First International Conference on System Integration*, pages 337–345, April 1990.

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

[19] Gerhard Weikum and Hans-J Schek. Multi-level transactions and open nested transactions. *IEEE Data Engineering Bulletin*, 14(1):60–64, March 1991.

[20] Gerhard Wiekum and Hans-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515–547. Morgan-Kaufmann, 1991.

A TaSL Grammar

<interaction> ::=
begin_interaction {
<decls> <body>
} **commit_interaction**
<subr_set>
<decls> ::= ϵ | <type> <namelist>**;** <decls>
<type> ::= **int** | **string** | ...
<namelist> ::= <name> | <name> **,** <namelist>
<name> ::= <ident>
<body> ::= ϵ | <tl_stmt> <body>
<tl_stmt> ::=
<stmt_list> | <alt_list> | <atomic>
<alt_list> ::=
if not { <eq_pris> } **then** <alt_list> |
if not { <eq_pris> } **then** { <eq_pris> } |
if not { <eq_pris> } **then NULL;**
<eq_pris> ::=
<atomic> | <atomic> **or** <eq_pris>
<atomic> ::= **atomic** { <stmt_list> }
<stmt_list> ::= <stmt> | <stmt> **;** <stmt_list>
<stmt> ::=
<step_call> | <¡assignment> |
<subr_call> | <fork> | <join> |
<wait> | <weak_conf> | <if>
<step_call> ::=
<db_name>**::**<proc> **(** <arglist> **)** |
<label>**:**
<db_name>**::**<proc> **(** <arglist> **)**
<label> ::= <ident>
<db_name> ::= <ident>
<proc> ::= <ident>
<arglist> ::= <arg> | <arg> **,** <arglist>
<arg> ::= <ident> | <value>
<assignment> ::=
<ident> **=** <step_call> |
<ident> **=** <subr_call>
<fork> ::= **fork** **(** <name> **)** { <body> }
<join> ::= **join** **(** <namelist> **)**
<wait> ::= **wait** **(** <event> **)**
<weak_conf> ::=
on **(** <event> **)** { <abort> } |
until <label> **on** **(** <event> **)** { <abort> }
<event> ::= <step_call> <cond> <value>
<cond> ::= **==** | **!=** | < | <= | > | >= | **del**
<abort> ::= **abort_to** **(** <label> **)**
<if> ::=
if(<if_cond> **)**
then { <stmt_list> }
else { <stmt_list> }
<if_cond> ::= <step_call> <cond> <value>
<subr_call> ::= <proc> **(** <arglist> **)**
<subr_set> ::= ϵ | <subr> <subr_set>
<subr> ::=
<type> <proc> **(** <arglist> **)** { <body> } |
void <proc> **(** <arglist> **)** { <body> }
<ident> ::= <string>
<value> ::= <string>