Executing Multidatabase Transactions

Mansoor Ansari, Marek Rusinkiewicz
Department of Computer Science
University of Houston
Houston, TX 77204

Linda Ness, Amit Sheth
Bellcore, MRE-2L359
445 South St.
Morristown, NJ 07962

Abstract

In a multidatabase environment, the traditional transaction model has been found to be too restrictive. Therefore, several extended transaction models have been proposed in which some of the requirements of transaction, such as isolation or atomicity, are optional. In this paper, we describe one of such extensions, the Flexible Transaction model and discuss the scheduling of transactions involving multiple autonomous database systems managed by heterogeneous DBMSs.

The scheduling algorithm for Flexible Transactions is implemented using L.0, a logically parallel language which provides a framework for concisely specifying the multidatabase transactions and for scheduling them. The key aspects of a Flexible Transaction specification, such as subtransaction execution dependencies and transaction success criteria, can be naturally represented in L.0. Furthermore, scheduling in L.0 achieves maximal parallelism allowed by the specifications of transactions, which results in the improvement of their response times.

To provide access to multiple heterogeneous hardware and software systems, we use the Distributed Operation Language (DOL). DOL approach is based on providing a common communication and data exchange protocol and uses Local Access Managers to protect the autonomy of member software systems. When L.0 determines that a subtransaction is ready to execute, it hands it through an interface to the DOL system for execution. The interface between L.0 and DOL provides the former with the execution status of subtransactions.

The work described in this paper has been performed in the course of a joint research project involving Bellcore and the University of Houston.

1 Introduction

The need to access data stored in multiple autonomous and possibly heterogeneous databases has led to the development of *federated multidatabase systems* [HM85, LA86, SL90]. One of the major issues in such systems is the management of multidatabase transactions. Since multidatabase transactions frequently involve long-running activities, some of the requirements of traditional transactions such as atomicity, isolation, and durability [Gra78] may become too restrictive for such transactions. Therefore, there have been several attempts to introduce a transaction model that would be more suitable for multidatabase environment.

In this paper, we discuss the execution of *Flexible Transactions* based on a multidatabase transaction model in which atomicity and isolation requirements of traditional transactions are relaxed. A Flexible Transaction [RELL90], is a collection of subtransactions related by a set of *execution dependencies*. Associated with each Flexible Transaction is a set of *acceptable states* corresponding to a successful completion of the global transaction.

The scheduling of Flexible Transactions is more complex than in the case of nested transactions, because the scheduler must keep track of execution dependencies and the acceptable states. Scheduling of Flexible Transactions using extended Petri Nets has been discussed in [ELLR90] and [LEB90]. Another scheduling mechanism for Flexible Transactions using **VPL** (Vienna Parallel Logic Language) is proposed in [KPE91].

The approach described in this paper uses an executable temporal logic language. The main advantage of using such a language to implement the scheduler is that its underlying semantics corresponds closely to the semantics of Flexible Transactions, thus allowing to achieve the maximum intra-transaction parallelism allowed by the specification of the Flexible Transaction. Unlike the proposals referenced above, the scheduler described here has been implemented and successfully applied to the scheduling of real telecommunication applications.

The *Executor* of Flexible Transactions accepts the specification of a transaction as an input, and schedules the subtransactions until the *success* or *failure* conditions of the transaction are satisfied. The processing of the scheduled subtransactions is supervised by the execution monitor. Two main software tools are used to implement the Executor. For scheduling subtransactions, **L.0**, a logical parallel language developed at Bellcore [CCG+91, Nes90a, Nes90b], is used. Since L.0 does not provide the facility to directly access databases, we use **DOL** (Distributed Operation Language) [ROEL90], developed at the University of Houston, for performing such tasks. The development of the interface between L.0 and DOL is an important task in the design of the Executor.

The implementation of the scheduler and the exe-

0073-1129-1/92 $3.00

cution monitor was the first phase of a joint research project involving Bellcore and the University of Houston. During the second phase of the project, we have used the Flexible Transaction paradigm to specify a telecommunication application involving multiple database systems. The implementation of the scheduler has provided us with a better understanding of the Flexible Transaction model, revealing a number of issues which were not discussed in the original specifications. This has lead to several additions to the model and resulted in more precise specifications. We believe that the experiences gathered in developing our applications will allow us to gain a deeper insight into both the strengths and weaknesses of the model for this class of applications.

The rest of the paper is organized as follows. In section 2, we review the transaction model, and discuss some extensions that allow us to capture more semantics of a transaction. In section 3, we discuss the implementation of the Executor and describe the software tools which are used. We also explain the advantages of using L.0 as the language of the scheduler. Section 4 presents the conclusions. Appendix A illustrates an L.0 specification of a Flexible Transaction. Appendix B gives details of the scheduler. Appendix C shows how a subtransaction can be expressed as a DOL program.

2 Flexible Transaction Model

We assume that each (global) transaction in the Flexible Transaction model can be specified by providing the following information [RELL90]: (a) a set of subtransactions, (b) (scheduling) preconditions associated with each subtransaction, and (c) a set of conditions defining the success of the global transaction. The model allows both *compensable* [Gra81] and *noncompensable* subtransactions to coexist within a single Flexible Transaction. We may take advantage of the compensability of a subtransaction to increase the availability of data and decrease the possibility of a deadlock [GMS87].

Most nested transaction models use *commitment protocols* to assure that all subtransactions constituting a global transaction are either committed or aborted. Typically, they assume the existence of a *prepared to commit* state. A subtransaction which has finished all its operations can wait in this state for a commit or abort signal from the global transaction manager. However, some DBMSs do not offer a visible *prepared to commit state* (e.g. IMS). Execution of a noncompensable subtransaction in such a DBMS can violate the consistency of the system. In such systems, we can execute compensable subtransactions, with the understanding that they will be compensated, when a global transaction aborts. In addition, since multidatabase transactions are frequently long-running activities, holding the lock on data by the subtransactions pending in the prepared to commit state, lowers the availability of the data. A compensable subtransaction can commit locally and release the lock on data, without waiting for a decision from the global transaction, assuming that it can be compensated if necessary.

The preconditions associated with subtransactions determine when they can be scheduled for execution. The precondition for a subtransaction is either an *execution dependency*, a *temporal dependency*, or a combination of both. A subtransaction with an execution dependency can be executed only in the case of a success and/or failure of one or more subtransactions. A dependency on the success of other subtransactions is referred to as *positive dependency*. A dependency on the failure of other subtransactions is referred to as *negative dependency*. A subtransaction with temporal dependency can be executed only before/after specific time. Subtransactions that do not have execution dependencies and have temporal dependency of time zero, the time at which global transaction starts to execute, are referred to as *primary subtransactions*.

Figure 1 illustrates a simple dependency among eight subtransactions of a Flexible Transaction. ST1 and ST6 are primary subtransactions and may start concurrently. ST2 is scheduled when ST1 succeeds. If ST2 fails, ST3 is scheduled, and if ST2 succeeds, ST5 is scheduled. ST5 can also be scheduled when ST4 succeeds.

In some situations, several subtransactions can achieve the same subgoal. This property of multidatabase transactions is called *function replication* [ELLR90]. Consider the following example.

A travel agent is planning a trip from Los Angeles to Houston. She has two subgoals to achieve: to find a seat on a flight to Houston and to reserve a car in a car rental agency. She has several options for each part. If she is not able to find a seat on a flight to Houston, it is useless to rent a car. So *renting a car* is positively dependent on *finding a seat*. Notice that the failure of a subtransaction such as getting a seat on a Delta flight does not effect the execution of renting a car in Avis. Semantically, it is the failure of the subgoal *finding a seat* that affects the subgoal *renting a car*. Let us assume that each subgoal can be achieved by successfully executing any of n subtransactions. In this case, execution dependencies exist between each of n *rent a car* options and each of the n *find a seat* options which means up to $n * n$ dependency specifications. This becomes more complicated, if there are more subgoals with multiple options.

To simplify specification of such dependencies among subtransactions we introduce the notion of a *cluster*. A cluster is a group of subtransactions that can achieve a given subgoal of a global transaction. Thus, execution dependencies may involve individual subtransactions or clusters. Associated with each cluster, is a condition for the success of the cluster. A subtransaction with a positive/negative dependency on a cluster can be executed only after success/failure of a cluster.

The conditions for the success of a Flexible Transaction can be specified by providing a *set of acceptable states*, defined in terms of the states of each of the subtransactions. An acceptable state of a Flexible Transaction is specified in disjunctive normal form.

Four execution states of a subtransaction are: *Not*

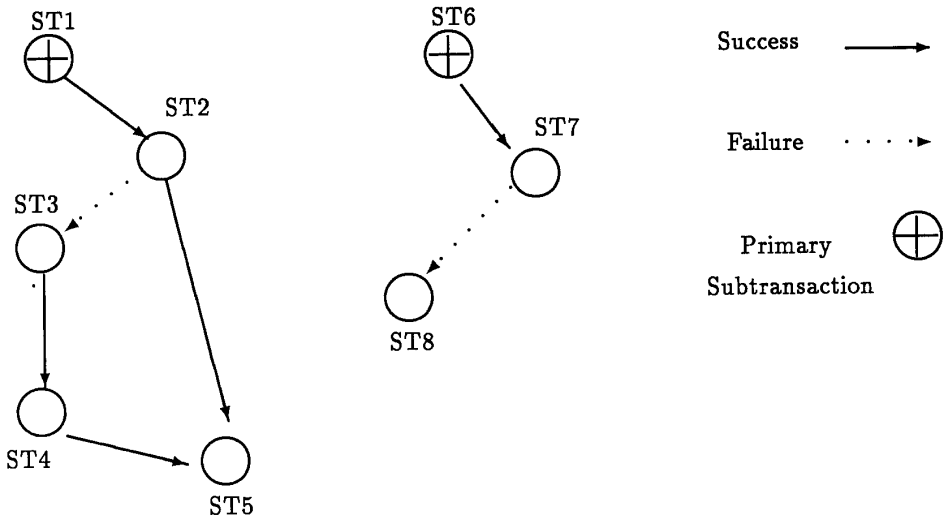

Figure 1. An Execution Dependency Graph

executed (N), *Executing (E)*, *Success (S)*, and *Failure (F)*. The notion of success or failure of a subtransaction has different semantics than commit or abort. We regard the execution state of a subtransaction which is prepared to commit or is locally committed as *S*. We regard the execution state of a subtransaction which is locally aborted or is compensated as *F*. These four execution states can be specified as a state of a subtransaction in the set of acceptable states. In addition, we use two additional states that can be assigned to subtransactions in the set of acceptable states. These two are described in the following paragraphs.

In some acceptable states, specifying the success or failure of a subset of subtransactions is sufficient for the determination of the success of a global transaction. In such cases, since the execution state of remaining subtransactions has no effect on determining the success of a transaction, we assign *Don't care (D)* to the state of such subtransactions. *D* implies any of four previously discussed execution states. It is used to simplify the specification of a set of acceptable states.

In some situations, it might be desirable to concurrently run several subtransactions with the same objective, but to allow only one of them to succeed. Consider concurrent execution of two subtransactions for reserving seats on two different airlines, each of which is selling a limited number of tickets at a special fare for a limited time. The success of one subtransaction is necessary as a part of an acceptable state. However, success of both subtransactions is undesirable. Such conflict can be resolved by requiring that if one of the two subtransactions succeeds, the other must fail and leave no effect on the database. To support concurrent execution of such subtransactions, we assign *S* to one of the subtransactions, and *Must fail (M)* to the rest. A subtransaction that is assigned *S* is given a higher priority and is used to resolve conflicts when an acceptable state is reached.[1].

Both *D* and *M* have no effect on determining whether an acceptable state has been reached. When an acceptable state is reached, no further action is taken for subtransactions whose state is designated as *D* in the accepted state. However, all the subtransactions designated as *M* in the accepted state are completed as follows. Before the global transaction commits, all subtransactions with state *M* which have been executed and not failed, must fail. If a subtransaction is in a prepared to commit state or if it is still executing, it is forced to abort. If a subtransaction has committed, its compensating transaction is scheduled for execution. In the simple two flight reservation example, the maximum concurrency can be achieved by specifying (*S*, *M*), (*N*, *S*), (*E*, *S*), (F, *S*) as the set of acceptable states.

A possible set of acceptable states corresponding to the dependency graph of Figure 1 is shown below as an example.

$$\{(S,S,N,N,S,M,M,M),\ (S,F,S,S,S,M,M,M),\ (M,D,D,D,M,S,S,N),\ (M,D,D,D,M,S,F,S)\}$$

The Flexible Transaction starts executing by scheduling the primary subtransactions. Upon the success or failure of a subtransaction, it checks whether an acceptable state is reached. If not, it

[1]This simple priority based conflict resolution can be further extended to include more complex conflict resolution mechanisms.

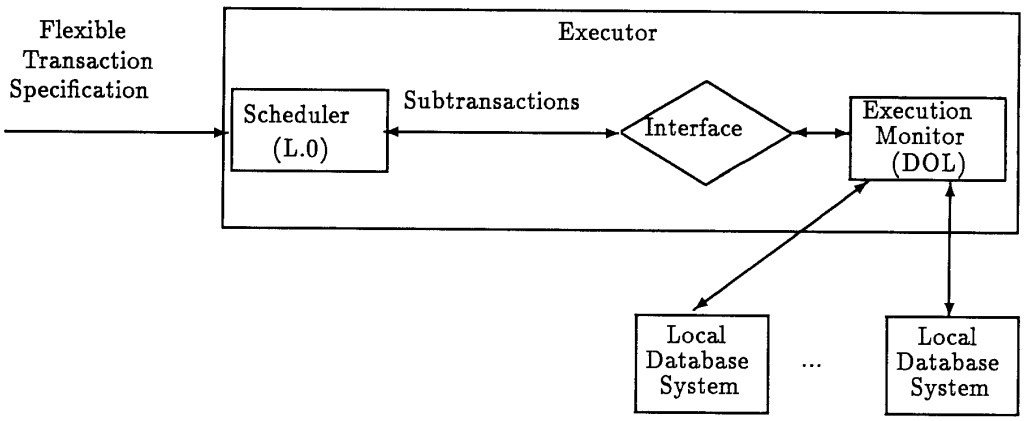

Figure 2. The structure of the Executor.

schedules further subtransactions which have execution dependencies on the finished subtransaction. The Flexible Transaction succeeds if any of the acceptable states is reached. This state is referred to as *accepted state*. It fails if there is no subtransaction to be scheduled, no subtransaction is executing, and no acceptable state is reached.

3 Execution of Flexible Transactions

There are two major tasks in the implementation of an Executor for Flexible Transactions. The first task is to schedule subtransactions and determine the success or failure of global transaction. The second task is to execute subtransactions in the local database systems.

The scheduling algorithm for Flexible Transactions is implemented using L.0 which allows concise specification of the scheduling constraints on the subtransactions [CNS91]. The Scheduler receives the specification of a Flexible Transaction consisting of a set of subtransactions, their dependency set, and the set of acceptable states[2]. Appendix A shows the specification of a Flexible Transaction, expressed in L.0. It corresponds to the Flexible Transaction shown in Figure 1.

To control the execution of the scheduled subtransactions, we use the Distributed Operation Language, DOL, which has been designed to access multiple and heterogeneous hardware and software systems. By interfacing L.0 and DOL, we allow the scheduler to cooperate with the execution monitor in executing Flexible Transactions. The structure of the Executor is illustrated in Figure 2.

[2]The specification of a Flexible Transaction can be expressed in a pseudo language. In this case, the specification must be translated to L.0 before being passed to the scheduler. The translation can be done either manually or by a program. In the latter case, a graphical interface to the user can be designed to accept the specification, and translate it to L.0. This would eliminate the user's need for knowing how to specify a Flexible Transaction in L.0, and the existence of L.0 will become transparent.

3.1 Execution of the subtransactions

Figure 3 shows the state transition diagram for subtransactions during scheduling. Some state transitions are determined by the Executor and some by the member database systems. Once a precondition of a subtransaction becomes true, it is scheduled for execution. Upon the failure of a subtransaction (determined by the member DBMS), its state changes to *Local Abort*. Upon the success of a subtransaction (also determined by the member DBMS), its state changes to *Local Commit*, if it has committed, or to *Prepared to Commit*, if it waits at the prepared to commit point. At any of these two points (i.e., Local Commit or Prepared to Commit), the subtransaction state does not change until the success or failure of global transaction is determined.

If the global transaction succeeds, the following state transitions occur. If a subtransaction is in a Prepared to Commit state and its success is part of the accepted state, then it commits and its state becomes Local Commit. If a subtransaction is in a Prepared to Commit state and its success is conflicts with the accepted state, it aborts and its state becomes Local Abort. If a subtransaction is in a Local Commit state and its success is a part of the accepted state, its state remains unchanged. If a subtransaction is in a Local Commit state and its success is contradictory to the accepted state, it is compensated (providing that it is compensable) and its state become *Compensated*.

If the global transaction fails, the following state transitions occur. If a subtransaction is in a Prepared to Commit state, it is aborted and its state becomes Local Abort. If a subtransaction is in a Local Commit

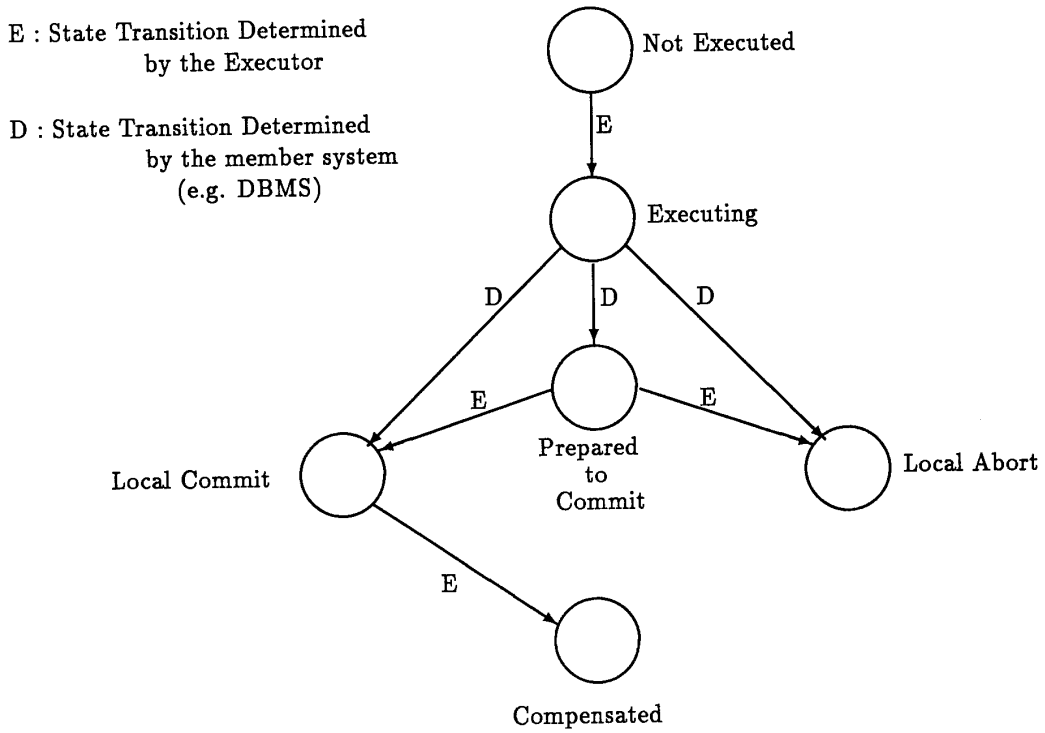

Figure 3. State Transition Diagram for subtransactions

state, it is compensated (providing that it is compensable) and its state becomes Compensated.

3.2 Using L.0 to implement the scheduler

L.0 is a rule-based language, which was designed to allow fast prototyping of software and hardware protocols [CCG+91]. Such protocols constrain the behavior of a number of different agents or components, so that when they act together, as prescribed by the protocol, a common goal is achieved, such as reliable transmission of data, fair resource allocation, recovery from an error state, correct execution of a hardware circuit, or success or failure of a Flexible Transaction.

Often these protocols are stated as sets of *guarded commands* (rules). Each set of guarded commands specifies the behavior of a particular agent or component. In hardware, all of the components are active simultaneously and forever. In the case of Flexible Transactions, each subtransaction may be viewed as an agent, and the whole Flexible Transaction may be viewed as a protocol for coordinating the behavior of each of these subtransactions. The agents (subtransactions) need to be active only until success or failure of the whole Flexible Transaction is detected. In Flexible Transactions, the protocol each agent is supposed to follow, is basically the same. Thus, it can be described via a parameterized set of guarded commands, which is instantiated once for each subtransaction, using the data particular to that subtransaction. In general in software protocols, only some of the agents need to be active simultaneously at each *phase*, and often there are a number of different groups of similar agents acting simultaneously.

To ease specification of such systems, the fundamental semantics of L.0 is synchronous execution of *guarded predicates*. At each L.0 step, all of the guards in all of the sets of guarded commands, which are currently active, are evaluated. Then, in the second phase of the same step, all of the *actions*, whose guards were true are taken. These actions appear to be simultaneous to the user. The guards in L.0 are usually referred to as *causes*, and the actions as *effects*.

For example in the scheduler of Flexible Transaction (whose code is in Appendix B), for each subtransaction, there is one guarded command of the form:

```
whenever
   <precondition for execution>
   &
   <the state of the subtransaction
    is "Not_Executing">
then
```

```
<assign state of subtransaction
 to be "Executing">
&
<invoke DOL to start execution
 of that subtransaction>;
```

The semantics of L.0 implies at each step, all of the subtransactions, which have not been executed yet and whose preconditions are true, will be scheduled for execution by DOL.

To permit different sets of guarded commands to be active at different times, an **until** construct is provided, which can be used to remove one set of guarded commands and activate other set(s) of guarded commands.

In the scheduler, an **until** construct is used to remove the set of guarded commands specifying the state transitions of the subtransactions when either an acceptable state is reached or the global transaction fails.

L.0 has only one data type, a tree with labeled edges. The size of the tree is logically unlimited. Declaration of variables is not required. Figure 4 illustrates an L.0 tree structure. The tree structure permits a hierarchical organization of data. The Flexible Transaction model may be viewed as a data type which can be realized within this L.0 type. Thus, each Flexible Transaction is represented as an instance of this data type. Two types of basic queries are permitted on this data type. One basic query is a test for the existence or non-existence of a path from the root. The other basic query is a test for equality between trees. The legal cause predicates are formed from the basic queries by combining them by the usual boolean operators, and possibly quantifying them universally or existentially. A tree may either be explicitly described, computed by a function written in **C** , or, if it is a subtree of the current or previous state tree, may be referred to by giving the path leading from its root. There are two basic types of updates permitted on this data type. One type of update, whose syntax is $< pathname >=< treedescription >$, adds the path name from the root of the current context if it does not already exist, and *assigns* the tree described to be its subtree (replacing the tree that might have already been there). The other type of update deletes a leaf edge. Intuitively path names from the root of a tree may be thought of as variable names. Conjunctions of these updates may be used in effects. In addition, an effect may *call* a number of procedures (called capsules in L.0), or a number of instances of one L.0 capsule, or both. The L.0-DOL interface is an interesting example of the exploitation of side-effects of **C** functions permitted in L.0.

Parameters may be passed by value or by reference in L.0. Parameter names are path names in the *local* tree. Reference parameters set up equalities between the *values* of path names in two different trees, referred to in a procedure and a procedure it calls.

L.0 also has quantification parameters, which can be viewed as in-line pass-by-value parameters. They are instantiated by a restricted form of universal quantification. Quantification is the key to the design of the scheduler, as illustrated in Appendix B. The syntax for quantification parameters in L.0 is $? < label >$. The restricting predicate for each quantification parameter, determines the set of possible label values that the parameter may assume. For example, in the expression below, the values of ?subtrans are restricted to be names of subtransactions of the Flexible Transaction:

```
forall (?subtrans) st
        {exists(Trans:?subtrans);}
```

Here `Trans` is the path name (of length one) leading to the tree which contains the specification of the Flexible Transaction. So `Trans` can be viewed as the variable whose (tree) value, is the specification of the Flexible Transaction. Technically, L.0 does not have a notion of variable.

The guarded commands for subtransactions are implemented in the scheduler using **whenever** cause-effect rules. Appendix B.1 shows these cause-effect rules for scheduling subtransactions. By using **forall** quantification, cause-effect rules are executed for all subtransactions. Guarded commands to determine the success or failure of the Flexible Transactions are implemented using **until** deactivator (Appendix B.2). Once the cause of any of the **until** deactivators becomes true, all other rules are deactivated and are removed from the rule space. Upon the completion of the effect of the **until** rule, it is also removed from the rule space and the execution stops.

Using L.0 to implement the scheduler has several advantages. The basic idea underlying L.0 is synchronous execution of quantified guarded predicates. The synchronous execution allows modeling of maximal parallelism. The parallelism may further be restricted according to the dependency constraints and the limitations of the execution environment.

Another important advantage is that some features of L.0 such as quantification and cause-effect rules are very expressive, and therefore permit an easy implementation of the scheduler. Furthermore, the specification of Flexible Transaction fits the nature of L.0's data structure and can be expressed easily using this data structure. Interfacing to DOL is straightforward, since L.0 provides the facility to call functions written in C.

3.3 Basic concepts of DOL

DOL can be used to specify a distributed execution of a global application in a heterogeneous computing environment [ROEL90]. Its major components are the Execution Engine, Service Directory, and LAMs (Local Access Managers).

The Engine is responsible for the execution of the DOL programs. Internally, it plays the role of a task controller and information flow controller. For each task to be performed at a site, it checks with Service Directory to determine how that site can be accessed. Then, it spawns an instance of a LAM on that site to perform the task. It supplies the LAM with all the necessary information, including commands and input data. Upon the termination of the task, it receives

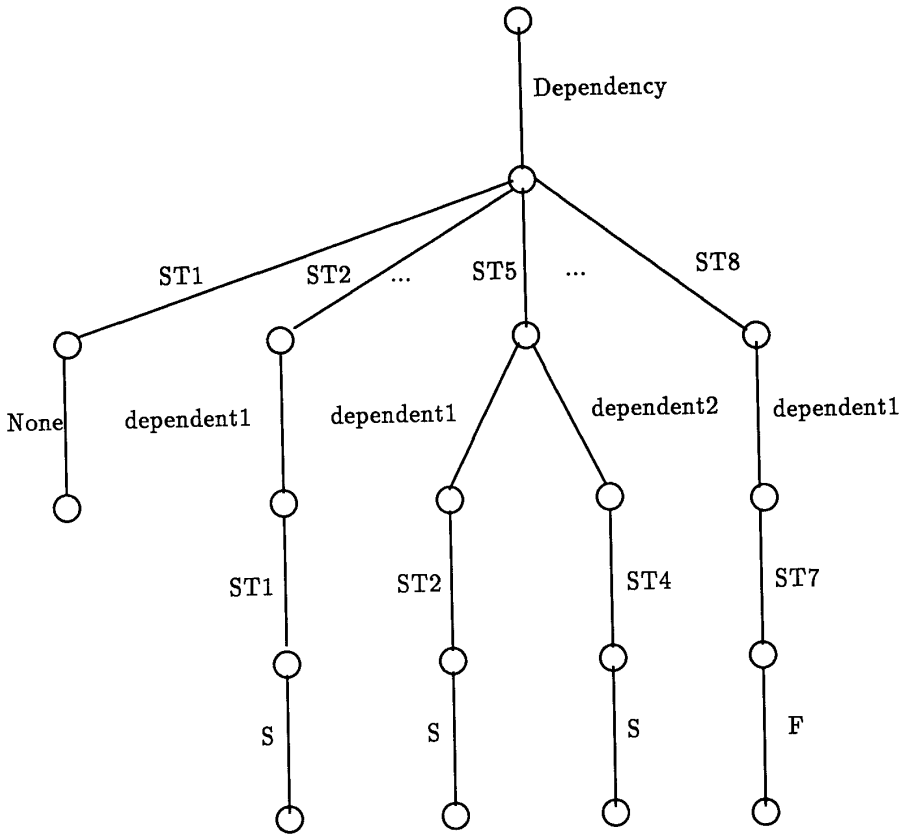

Figure 4. An L.0 tree structure corresponding to Execution Dependency Graph of Figure 1.

possible output and the status of the task from the LAM.

A LAM acts as a proxy user for the software system it manages, encompassing it in a sort of logical shell. Each LAM knows how to communicate with the Engine and with its local software system. It provides to the local software system commands and data which it receives from the Engine and returns back to the Engine the output produced by the local software system. It also provides the Engine with the status of the performed task.

This architecture allows an easy addition of software systems to DOL. To incorporate a new system, we need to design a LAM for it and add its access information, such as its network address, to the Service Directory.

3.4 Interfacing L.0 to DOL

The main concern in designing the L.0 interface to DOL was to allow the asynchronous execution of the DOL programs (subtransactions) so that L.0 program (scheduler) did not have to wait for each DOL program to finish before it would continue its scheduling job. The design of the interface is illustrated in Figure 5.

Four C functions: **DoTrans, GetState, CommitTrans**, and **AbortTrans** are added to the L.0 C-library to interface it with DOL. They are described as below.

DoTrans is in charge of establishing the communication channel and starting the Interface process. It first establishes a socket for communication between L.0 and Interface. Then, it creates a child process and initiates an instance of the Interface in the child process. It passes the communication information and the name of the file containing DOL program (a subtransaction) to be executed, as the arguments to the Interface. Finally, without waiting for the child process to finish, it returns back to the L.0. It returns the communication information of the established channel.

GetState reports the current state of a subtransaction, upon the request from L.0 program (the scheduler). It works as follows. It checks whether there is any returned result from the Interface or the LAM, in

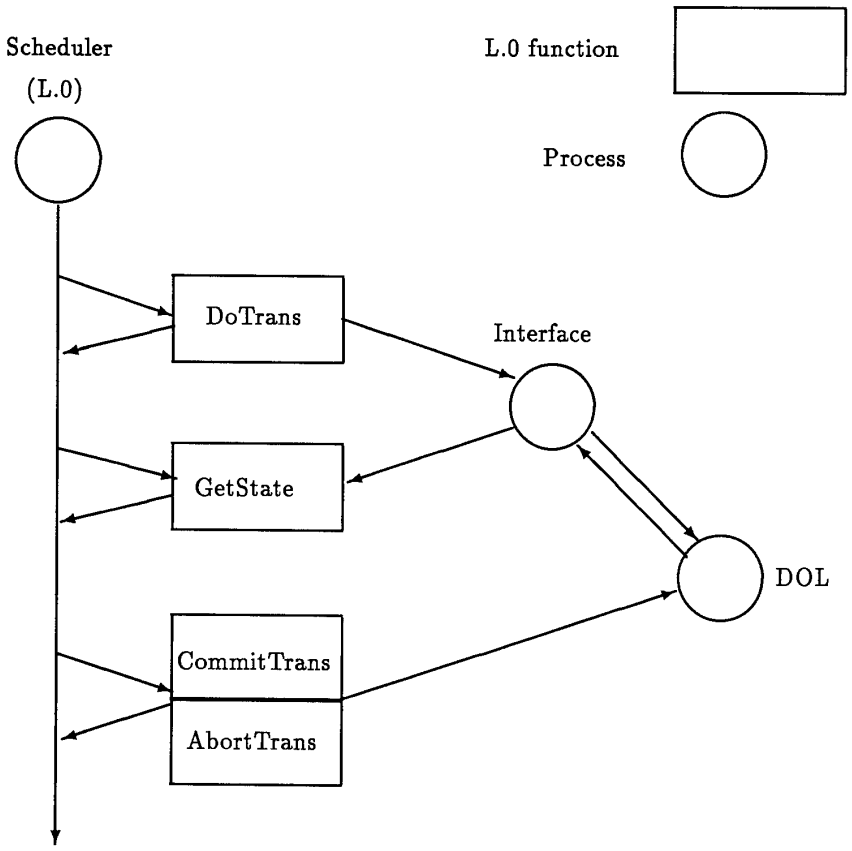

Figure 5. Interfacing L.0 to DOL

the socket. If there is any new result of the executing subtransaction, that is *LocalCommit*, *LocalAbort* or *PreparedToCommit*, it reports it back to L.0. Otherwise, it reports *Executing* as the current state of the subtransaction. If the subtransaction has locally committed or aborted, it closes the socket so that it can be reused. However, if the state is prepared to commit, it keeps the channel alive, so that it can be used for a commit or abort signal later in the program.

CommitTrans and AbortTrans are used for the subtransactions which are in the prepared to commit state. If the scheduler decides to commit the pending subtransaction, it calls the CommitTrans. The CommitTrans uses the already established channel to signal the subtransaction to commit. Similarly, AbortTrans is used if the scheduler decides to abort the pending subtransaction.

The DOL Interface is a process started by DoTrans. It creates a child process to execute a subtransaction and waits for the result. It communicates with its child through a pipe. If a subtransaction fails or succeeds without waiting in a prepared to commit state, the DOL Engine reports the state of the subtransaction back to the Interface. The DOL Engine reports any kind of failure such as failing to connect to a service or aborting a subtransaction in the DBMS as *Failure*. In case of success, Interface records any output in a file and reports local commit to the L.0. In each case, Interface plays the role of a filter. The implementation permits complex filtering depending on data returned by a subtransaction as well as on the state information.

If a subtransaction reaches a prepared to commit state, LAM reports this to L.0. At this point, the subtransaction waits for a commit or abort signal from the scheduler before it resumes its execution.

4 Conclusions

In this paper, we have described the specification and execution of Flexible Transactions. L.0, an executable temporal logic language, is used for specifying a Flexible Transaction and scheduling its subtransactions. DOL is used for executing each scheduled subtransac-

tion on databases which are accessed through LAMs.

L.0 supports synchronous execution of quantified guarded predicates. The guarded predicates preceded by the keyword **whenever**, contain the preconditions specifying dependencies among subtransactions. All such guarded predicates are evaluated within the scope of a **forall** command that allows simultaneous scheduling of all subtransactions whose preconditions are satisfied. This allows maximal parallelism permitted by the specification of the Flexible Transaction. The **until** deactivator is used to determine global success and failure of a Flexible Transaction and to terminate it. L.0's data structure allows straight-forward specification of a Flexible Transaction. Since L.0 provides the facility to call functions written in C, the interface between L.0 and communication modules as well as DOL (including the Local Access Manager interfaces to heterogeneous DBMSs) are easily specified. The complete L.0-DOL interface described in this paper has been implemented.

This work has led us to a better understanding of the Flexible Transaction model. We have extended the Flexible Transaction model in some aspects. By *clustering* subtransactions, we can simplify the specification of dependencies in cases where several subtransactions are intended to achieve the same subgoal. By using *Must Fail* as a state of a subtransaction, we allow more concurrency among alternative subtransactions that achieve the same subgoal.

Currently, we are investigating use of the multidatabase transaction paradigm for meeting transaction management needs of a telecommunication application. Our implementation will serve as a prototyping tool.

References

[CCG+91] E. Cameron, D. Cohen, T. Guinther, W. Keese, L. Ness, C. Norman, and H. Srinidhi. The L.0 Language and Environment for Protocol Simulation and Prototyping. *Transactions On Computers*, April 1991.

[CNS91] E. Cameron, L. Ness, and A. Sheth. An Executor for Multidatabase Transactions which Achieves Maximal Parallelism. *Proceedings of the First Int'l Workshop on Interoperability in Multidatabase Systems*, April 1991.

[GMS87] H. Garcia-Molina and K. Salem. SAGAS. *Proceedings of ACM SIGMOD Conference on Management of Data*, 1987.

[Gra78] J. N. Gray. Operating Systems: An Advanced Course. *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[Gra81] J.N. Gray. The Transaction Concept: Virtues and Limitations. *Proceedings of the 7th International Conference on VLDB*, September 1981.

[HM85] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. *ACM Trans. on Office Info. Syst.*, 3(3), July 1985.

[KPE91] E. Kuehn, F. Puntigam, and A. Elmagarmid. Transaction Specification in Multidatabase Systems Based on Parallel Logic Programming. *Proceedings of the First Int'l Workshop on Interoperability in Multidatabase Systems*, April 1991.

[LA86] W. Litwin and A. Abdellatif. Multidatabase Interoperability. *Computer*, 19(12), December 1986.

[LEB90] Y. Leu, A. Elmagarmid, and N. Boudriga. Specification and Execution of Transactions for Advanced Database Applications. Technical Report CSD-TR-1030, Computer Sciences Department, Purdue University, October 1990.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for Interbase. *Proceedings of the 16th Int'l Conference on Very Large Databases*, August 1990.

[Nes90a] L. Ness. A Quantified Synchronous Model of Computation which Permits Checking of a Class of Extended Temporal Logic Assertions. Technical Report TM-ARH-017230, Bellcore Technical Memorandum, 1990.

[Nes90b] L. Ness. Issues Arising in the Analysis of L.0: A Synchronous Executable Temporal Logic Language. *Proceedings of the Workshop on Computer-Aided Verification*, (DIMACS Technical Report 90-31), June 1990.

[RELL90] M. Rusinkiewicz, A. Elmagarmid, Y. Leu, and W. Litwin. Extending the Transaction Model to Capture more Meaning. *SIGMOD Record*, August 1990.

[ROEL90] M. Rusinkiewicz, S. Ostermann, A. Elmagarmid, and K. Loa. The Distributed Operational Language for Specifying Multisystem Applications. *Proceedings of the First Int'l Conference on Systems Integration*, April 1990.

[SL90] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3), September 1990.

Appendix A: Specification of a Flexible Transaction in L.0

In this appendix, the L.0 specification of the Flexible Transaction of Figure 1 is given. The global transaction specification consists of specification of each subtransaction, each compensating subtransaction, dependencies among subtransactions, and the set of acceptable states. The specification of a subtransaction consists of its code and its compensation information. The code for each subtransaction is stored in a file (e.g., dp*i* contains the code for subtransaction ST*i*). These files are not accessed during scheduling but are needed by DOL for executing a subtransaction after it has been scheduled.

Specification of subtransactions:

```
T1=:{ ST1:{filename:dp1;type:"Compensable";CompensableBy:CT11};
      ST2:{filename:dp2;type:"Compensable";CompensableBy:CT12};
      ST3:{filename:dp3;type:"NonCompensable"};
      ST4:{filename:dp4;type:"Compensable";CompensableBy:CT14};
      ST5:{filename:dp5;type:"NonCompensable"};
      ST6:{filename:dp6;type:"NonCompensable"};
      ST7:{filename:dp7;type:"Compensable";CompensableBy:CT17};
      ST8:{filename:dp8;type:"Compensable";CompensableBy:CT18} };
```

Specification of compensating transactions:

```
CompensatingTrans=:{ CT11:{filename:dp11};
                     CT12:{filename:dp12};
                     CT14:{filename:dp14};
                     CT17:{filename:dp17};
                     CT18:{filename:dp18} };
```

Dependency specification (see Figure 4):

```
Dependency=:{ ST1:"None";
              ST2:{dependent1:{ST1:S}};
              ST3:{dependent1:{ST2:F}};
              ST4:{dependent1:{ST3:S}};
              ST5:{dependent1:{ST2:S}; dependent2:{ST4:S}};
              ST6:"None";
              ST7:{dependent1:{ST6:S}};
              ST8:{dependent1:{ST7:F}} };
```

Set of acceptable states:

```
SetAcceptables=:{ state1:{S;S;N;N;S;M;M;M};
                  state2:{S;F;S;S;S;M;M;M};
                  state3:{M;D;D;D;M;S;S;N};
                  state4:{M;D;D;D;M;S;F;S} };
```

Appendix B: Scheduling Flexible Transactions

In this appendix, parts of the code for the scheduler are presented. In Section B.1, the code for scheduling subtransactions is shown. In Section B.2, the code for determining global success or failure is shown.

B.1: Scheduling Subtransactions

Whenever clause expresses precondition (*cause* in L.0) for each subtransaction, **then** clause (*effect* in L.0) schedules the subtransaction whose precondition is satisfied. (see subsection 3.4).

```
forall (?subtrans) st {exists (Trans:?subtrans);}
{
   {* If the subtransaction's execution guard is true,
      execute it, if it has not already been executed *}
   whenever
      (Trans:?subtrans:state=N) &
      ( (forsome (?PrimaryTrans) st {exists (Dependency:?PrimaryTrans:None);}
```

```
      { ?PrimaryTrans=?subtrans } ) |
     (forsome (?dep) st {exists (Dependency:?subtrans:?dep);}
      { forall (?subdep) st {exists (Dependency:?subtrans:?dep:?subdep);}
        { Dependency:?subtrans:?dep:?subdep=Trans:?subdep:result }}))
then
   {  Trans:?subtrans:state=E;
      Trans:?subtrans:CommInfo=DoTrans(Trans:?subtrans:filename'); };

{* Check to see if the result of the subtransaction is arrived yet and if so,
   receive the new state (local commit or abort, or prepared to commit) *}
whenever
   Trans:?subtrans:state=E
then
   Trans:?subtrans:state=GetState(Trans:?subtrans:CommInfo:socketno');

{* If the state is prepared to commit or local commit, set the result to success *}
whenever
     ( (Trans:?subtrans:state="PreparedToCommit") |
     (Trans:?subtrans:state="LocalCommit") ) &
     (Trans:?subtrans:result="None")
then
   Trans:?subtrans:result=S;

{* If the state is local abort, set result of subtransaction to failure *}
whenever
   (Trans:?subtrans:state="LocalAbort") &
   (Trans:?subtrans:result="None")
then
   Trans:?subtrans:result=F; };
```

B.2: Determining Global Success or Failure

The code for determining global success or failure is shown below.

Global Success predicates:

```
{* If an acceptable state has been satisfied, conclude the state transition
   of subtransactions (Figure 3), and then commit the Flexible Transaction.*}

until
   (forsome (?AcceptState) st {exists (SetAcceptables:?AcceptState);}
       { ( (forall (?SuccTrans) st {exists (SetAcceptables:?AcceptState:?SuccTrans:S);}
            { Trans:?SuccTrans:result=S }) |
           (forall (?SuccTrans) st {exists (SetAcceptables:?AcceptState:?SuccTrans);}
            { ~(SetAcceptables:?AcceptState:?SuccTrans=S) })) &
         ( (forall (?FailTrans) st {exists (SetAcceptables:?AcceptState:?FailTrans:F);}
            {  Trans:?FailTrans:result=F }) |
           (forall (?FailTrans) st {exists (SetAcceptables:?AcceptState:?FailTrans);}
            { ~(SetAcceptables:?AcceptState:?FailTrans=F) })) &
         ( (forall (?NotExecTrans) st {exists
                   (SetAcceptables:?AcceptState:?NotExecTrans:N);}
            { Trans:?NotExecTrans:state=N }) |
           (forall (?NotExecTrans) st {exists
                   (SetAcceptables:?AcceptState:?NotExecTrans);}
            { ~(SetAcceptables:?AcceptState:?NotExecTrans=N) }))})
then
   { {* Conclude the state transition of subtransactions (Figure 3) and commit *} };
```

Global failure predicate:

```
{* If no more subtransaction can be scheduled and no more is executing and
   no acceptable state is satisfied then conclude the state transition of
   subtransactions, and abort the Flexible Transaction. *}
```

```
until
      {* No subtransaction is executing *}
      (forall (?SubTransaction) st {exists (Trans:?SubTransaction:state);}
       { ~(Trans:?SubTransaction:state=E) &
         ~(((Trans:?SubTransaction:state="LocalCommit") |
            (Trans:?SubTransaction:state="LocalAbort")  |
            (Trans:?SubTransaction:state="PreparedToCommit") ) &
            (Trans:?SubTransaction:result="None"))})

      & {* No subtransaction can be scheduled *}
      (forall (?SubTransaction) st {exists (Dependency:?SubTransaction:None);}
       { ~(Trans:?SubTransaction:state=N) }) &
      ( (forall (?SubTransaction) st {exists (Trans:?SubTransaction:state);}
         { ~(Trans:?SubTransaction:state=N) }) |
        (forall (?SubTransaction) st {exists (Trans:?SubTransaction:state:N);}
         { ~(forsome (?dep) st {exists (Dependency:?SubTransaction:?dep);}
            { forall (?subdep) st {exists (Dependency:?SubTransaction:?dep:?subdep);}
               { Dependency:?SubTransaction:?dep:?subdep= Trans:?subdep:result }})}))

      & {* No acceptable state is satisfied *}
     ~(forsome (?AcceptState) st {exists (SetAcceptables:?AcceptState);}
       { ( (forall (?SuccTrans) st {exists (SetAcceptables:?AcceptState:?SuccTrans:S);}
            { Trans:?SuccTrans:result=S }) |
           (forall (?SuccTrans) st {exists (SetAcceptables:?AcceptState:?SuccTrans);}
            { ~(SetAcceptables:?AcceptState:?SuccTrans=S) })) &
         ( (forall (?FailTrans) st {exists (SetAcceptables:?AcceptState:?FailTrans:F);}
            { Trans:?FailTrans:result=F }) |
           (forall (?FailTrans) st {exists (SetAcceptables:?AcceptState:?FailTrans);}
            { ~(SetAcceptables:?AcceptState:?FailTrans=F) })) &
         ( (forall (?NotExecTrans) st {exists
                   (SetAcceptables:?AcceptState:?NotExecTrans:N);}
            { Trans:?NotExecTrans:state=N }) |
           (forall (?NotExecTrans) st {exists
                   (SetAcceptables:?AcceptState:?NotExecTrans);}
            { ~(SetAcceptables:?AcceptState:?NotExecTrans=N) }))})
then
   { {* Conclude the state transition of subtransactions (Figure 3) and commit *} };
```

Appendix C: A Subtransaction expressed in DOL

The following is an example of expressing a subtransaction as a DOL program. This subtransaction can be executed at site climax. The service is requested for Ingres DBMS.

```
BEGIN
OPEN (ingres) at climax as A
 SCOPE A
   <subtransaction>
 ENDSCOPE to output
CLOSE A
END.
```