Using Flexible Transactions to Support Multi-system Telecommunication Applications

Mansoor Ansari*
Dept. of Computer Science
University of Houston
Houston, TX 77204
ansari@cs.uh.edu

Linda Ness
Bellcore
445 South St.
Morristown, NJ 07962
linda@flash.bellcore.com

Marek Rusinkiewicz*
Dept. of Computer Science
University of Houston
Houston, TX 77204
marek@cs.uh.edu

Amit Sheth
Bellcore
444 Hoes Lane
Piscataway, NJ 08854
amit@ctt.bellcore.com

Abstract

Service order provisioning is an important telecommunication application that automates the process of providing telephone services in response to the customer requests. It is an example of a multi-system application that requires access to multiple, independently developed application systems and their databases. In this paper, we describe the design and implementation of a prototype system[1] that supports the execution of the Flexible Transactions and its use to develop the service order provisioning application. We argue that such approach may be used to support the development of multi-system, flow-through processing applications in a systematic and organized manner. Its advantages include fast and easy specification of new services, support for testing of the declaratively specified work-flows, and the specification of potential concurrency among the tasks constituting an application.
Keywords: multi-system application, work-flow control, service order provisioning, Flexible Transactions

*M. Ansari and M. Rusinkiewicz were supported, in part, by grants from Bellcore, MCC, and by the Texas Advanced Technology program under Grant No.3652008.

[1]The term "prototype" refers to a "software prototype" throughout this paper.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

1 Introduction and Motivation

In many industrial computing environments, dedicated application systems were developed to automate various organizational functions. These systems were usually designed independently and used their own databases to store the data they needed. As the scope of automation was expanded, the needs for new multi-system applications that require access to multiple autonomous systems began to grow. Such needs have usually been addressed by developing dedicated software systems which are said to control *work-flows*. When the work-flows are completely automated, i.e., do not require manual processing steps or human interventions, the system is said to support *flow-through* processing.

In current systems supporting flow-through processing, the integration of component systems has been usually implemented in an *ad hoc* manner, with the work flows hard-coded in the application. To develop robust multi-system applications, we need a model that allows flexible specification of work-flows and efficient control of their execution. Multidatabase transactions provide a technology that can be used to address these issues.

The multidatabase approach assumes the existence of multiple and possibly heterogeneous databases in which component database systems maintain their autonomy upon integration [HM85, LA86, SL90]. In managing multidatabase transactions, the problems associated with preserving the autonomy of participating database systems are aggravated by the fact that multidatabase transactions are often long running activities. This is inconsistent with the assumption of the traditional transaction model that transactions are short lived. To address this situation, several models have been proposed in the literature to relax some requirements of the traditional trans-

action model, such as atomicity or isolation, that may be too restrictive in a multidatabase environment [GS87, Elm92]. The management of work-flows and long running activities has been addressed, among others, in [Reu89, GGK+90, DHL91, Kle91].

During a cooperative research project involving Bellcore and the University of Houston, a prototype transaction processing system based on an extended transaction model was developed. The system was then used to implement a telecommunication application that served as a testbed for the prototype. The application we have selected is service order provisioning, which is the automated process of providing telephone services to customers. A service request is processed by multiple and heterogeneous systems that are responsible for performing optimal line and equipment assignments and updating the facility, customer and billing information databases. The systems involved in processing of a customer request have various functionalities and are, to a large extent, autonomous. In order to provide a service, the execution of these systems must be coordinated and data may need to be exchanged between them. This problem is fundamental to the telecommunications industry since its business consists of efficiently providing telecommunications services to customers.

The objective of the project was to determine whether the application of multidatabase transaction models might permit flow-through processing applications to be defined and supported quickly, flexibly, and efficiently. The project provided insight into both the strengths and weaknesses of the approach for this class of applications.

The transaction processing system described in this paper is based on the Flexible Transaction paradigm [RELL90, ELLR90]. A Flexible Transaction is a collection of subtransactions related by a set of *execution dependencies* among them. Associated with each Flexible Transaction is *a set of acceptable states* defining the conditions for the success of the global transaction. Therefore, the success of all subtransactions may not be necessary for the success of the global transaction. This characteristic of the Flexible Transactions provides flexible atomicity, by permitting specification of the subsets of subtransactions that should be treated as units of atomicity, rather than requiring the *all or nothing* property. In addition, partial results of the Flexible Transactions (the results of committed subtransactions) are visible to other transactions, if the subtransactions are declared to be compensable. This characteristic provides flexible isolation. Furthermore, declaring the execution dependencies of the subtransactions permits specification of intra-transaction parallelism.

The project consisted of two phases: (a) development of a prototype for executing Flexible Transactions, and (b) specification of provisioning for a class of service requests as a Flexible Transaction and executing it on the prototype. The prototype consists of a scheduler, which decides when the subtransactions of a Flexible Transaction should be submitted for execution and an Execution Monitor which submits each scheduled subtransaction to the appropriate DBMS, monitors its progress, and relays the information back to the scheduler. The second phase of our project involved modeling of a substantial application system and study of the applicability of the model in industrial context.

The rest of the paper is organized as follows. In section 2, the Flexible Transaction model is reviewed. Some extensions to the original model that were proposed as a direct result of this project, are provided in this section. Section 3 reviews service order provisioning and describes how a class of service orders can be modeled as a Flexible Transaction. Section 4 presents the steps in implementing this multi-system application using the Flexible Transaction paradigm and describes the architecture of our prototype for processing Flexible Transactions. Section 5 presents conclusions and future work.

2 Flexible Transaction Model

A Flexible Transaction is specified by providing the following information [ELLR90]: (a) a set of subtransactions, (b) a set of intra-transaction execution dependencies, and (c) a set of acceptable states defining the conditions for the success of the Flexible Transaction.

2.1 The Set of Subtransactions

Each subtransaction is a logical unit of work that performs some operations at a particular site. A subtransaction can be either compensable or noncompensable [Gra81, KLS90].

Most nested transaction models use *commitment protocols* to assure that all subtransactions constituting a global transaction are either committed or aborted. Some models assume the existence of a *prepared to commit* state. A subtransaction that has finished all its operations can wait in this state for a commit or abort

decision from the global transaction manager. However, since multidatabase transactions are frequently long-running activities, holding the locks on data by the subtransactions waiting in the prepared-to-commit state, may lower the availability of the data. Therefore, transaction models have been proposed that may take advantage of compensation to increase the availability of data and decrease the possibility of a deadlock [GS87, LKS91]. However, the applicability of these models is limited by the fact that not all subtransactions can be compensated (especially in a multidatabase environment). By allowing both compensable and noncompensable subtransactions to coexist within a single Flexible Transaction, the visibility of the subtransactions can be controlled as follows. A compensable subtransaction can commit locally and make its results visible to other (sub)transactions, assuming that it can be compensated, if necessary. A noncompensable subtransaction must wait in a prepared to commit state for a commit decision from the global transaction manager before it can make its results visible to others.

2.2 The Intra-transaction Execution Dependencies

One characteristic that most multi-transaction models share is that operations are grouped to form one or more subtransactions. Often a subtransaction can have an *execution dependency* as a condition to start a subtransaction, a condition to resume the execution of a halted subtransaction, or a condition to terminate a subtransaction [CR90]. The condition can be specified based on the *execution state* of one or more subtransactions, based on the *output* generated by other subtransactions, or based on *time*. A comprehensive list of executing dependencies have been discussed in [Ans92]; below we present a summary.

The execution state of a subtransaction can be *not-executed, executing, prepared-to-commit, committed, aborted,* or *compensated.* A number of execution dependencies involving the execution state of one or more subtransactions can be defined. The execution dependencies can be interpreted and defined from two different perspectives. One is from a historical perspective, i.e., whether a given history has maintained the execution dependencies of scheduled (sub)transactions [CR91]. It is useful when the inter-transaction dependencies of the subtransactions are being studied to establish a correctness criterion for the concurrent execution of the global transactions. The other is from a postulative perspective, i.e., to define the intra-transaction dependencies of the subtransactions in the specification of a global transaction [RELL90]. It is useful when the intra-dependencies of the subtransactions are being studied for constructing requirements for the execution of the global transactions. Below, different types of the execution dependencies are listed:

- The execution dependencies based on the execution state. These dependencies have been discussed frequently in the literature and include start dependencies (*start-start* dependency, *commit-start* dependency, *prepared-to-commit-start*, and *abort-start* dependency), commit dependencies (*commit-commit* dependency, *prepared-to-commit-commit* dependency, and *abort-commit dependency*), and abort dependencies (*weak-commit-abort*, and *weak-abort-abort* dependency).
- The execution dependencies based on the output of other subtransactions. These dependencies are sometimes referred in the literature as value dependencies [DE89].
- The execution dependencies involving time. The execution dependency of a subtransaction based on time can be *temporal-start dependency, temporal-commit dependency* [LT88], or *temporal-abort dependency.*

The execution dependencies listed above can be combined to express more complex semantics. To simplify the specification of the subtransactions that are assigned to perform a common subgoal of a global transaction (e.g. the set of subtransactions that can be used to reserve a car)[2], they can be grouped into *clusters.* Thus, execution dependencies may involve individual subtransactions or clusters. Associated with each cluster, is a condition for its success. A subtransaction with an execution dependency on a cluster can be executed only after success or failure of the cluster.

2.3 The Set of Acceptable States

The conditions for the success of a Flexible Transaction can be specified by providing a *set of acceptable states*, defined as a combination of the execution states of the subtransactions. If an acceptable state is reached

[2] In [LER90], this is referred to as function replication. For a function T, the local database systems which can be used to implement T are said to be *functionally replicated* for T.

during the scheduling of the subtransactions, no additional subtransaction need to be scheduled and the Flexible Transaction can terminate successfully.

Each acceptable state is specified as a conjunction of the *subtransaction states (s-states)*. The s-states corresponding to the execution states are presented in Table 1.

As an example, let's consider a Flexible Transaction consisting of three subtransactions. A set of acceptable states for this transaction can be specified as:

{(S,F,S) or (F,S,S) or ...}

The first acceptable state indicates that the Flexible Transaction can successfully complete if the first and the third subtransactions succeed and the second subtransaction fails. The second acceptable state indicates that the Flexible Transaction can complete successfully if the first subtransaction fails and the second and the third subtransactions succeed.

A Flexible Transaction may have a subset of subtransactions that can be alternatively used to achieve the same subgoal (such as the first and the second subtransaction in the above example). Such subtransactions can run concurrently; however, precautions must be taken, by the appropriate specification of the acceptable states, to ensure that only one of them is allowed to succeed. This is assured by the termination protocol, described in the next section.

2.4 The Execution of Flexible Transactions

The execution of a Flexible Transaction consists of a *scheduling phase* and a *terminating phase*. In the scheduling phase, subtransactions are scheduled according to their execution dependencies. All subtransactions whose execution dependencies are satisfied can be scheduled concurrently. Whenever the execution state of a subtransaction changes, the scheduler checks if an acceptable state has been reached. This phase ends if one of the following events occurs:
(a) One of the acceptable states is reached. An acceptable state is reached when all subtransactions whose success is required reach the S state[3]. This state is referred to as the *accepted state*. In this case, the Flexible Transaction has succeeded and therefore becomes *ready to commit*.
(b) No subtransaction is executing, no more subtransaction can be scheduled, and no acceptable state is reached. In this case, the Flexible Transaction has failed and therefore becomes *ready to abort*.

In the terminating phase, the execution of the Flexible Transaction is completed as follows:
(a) If the Flexible Transaction is ready to commit, the accepted state is used to determine which subtransactions must be committed, aborted, or compensated:

- All the subtransactions that are still executing, are aborted.
- All the compensable subtransactions that are committed, but their failure is required in the accepted state, are compensated.
- All the subtransactions that are in the prepared-to-commit state, are either committed or aborted, as required by the accepted state.

(b) If the Flexible Transaction is ready to abort, all the subtransactions in the prepared-to-commit state are aborted, and all the committed subtransactions are compensated[4].

3 Service Order Provisioning

Service order provisioning is the automated process of providing a telephone service to a customer. The process of providing a service is carried out in a distributed environment and requires access to multiple heterogeneous databases. The size and the complexity of the components and the fact that they maintain a real time inventory are some of the reasons for preserving the autonomy of the component systems. In the discussion below, we will refer to the real applications and databases[5]. We will attempt to extract general characteristics of such applications, including the execution dependencies that exist among various tasks, omitting unnecessary details whenever possible.

A customer's request for a service is entered into the Service Order Processor (SOP) which reads the *service request* and converts it to the *service order*. From there, the service order is sent to different business units within the company for further processing. The order is sent to the provisioning unit where it is

[3] If more than one acceptable state is reached simultaneously, one of them is selected. The choice can be either arbitrary or based on additional specification of preference among the acceptable states.

[4] Noncompensable subtransactions are not allowed to commit in the scheduling phase.

[5] NOTICE: Description of applications, databases, and systems is abstract and for illustration only. Bellcore and Bellcore Client Companies may not be supporting these products or may not be using them in the way described in this paper.

s-states	corresponding execution states
N	not-executed
S (Success)	committed, prepared-to-commit
F (Failure)	aborted, compensated
*	The execution state of the subtransaction has no effect on reaching the corresponding acceptable state.

Table 1: List of possible s-states in an acceptable state

analyzsed and processed for line and equipment assignments. After completing the assignments, SOP receives the final status of the provisioning.

3.1 Provisioning Environment

Provisioning[6] often involves three distinct types of areas in which different types of assignments may take place (see Figure 1). One type of area, called *local loop*, is from customer premises to the *central office*. All local loops in a geographic area terminate in the central office where the necessary equipment for switching between cables and wires reside. The second type of area is within a central office. The third type of area is between central offices and is called inter-office or trunk area. Figure 1 shows two local loops, two central offices, and one inter-office area. Usually one application system, called operation support system, is dedicated to assignments in each area.

The assignment of a *local loop*, the transmission path between central office and customer premises, is done by Loop Facility Assignment and Control System (LFACS). LFACS is a Bellcore provided operation support system for the loop assignment center. The inventory for local loop facilities is maintained in the LFACS database and include, information about *cables, cable pairs, distribution terminals, cross box terminals, binding posts, living units, customer service wires, remote switches*, etc.

The assignment of the line-side equipment within the central office (the equipment dedicated to the local loop) is done by the COmputer System for Main frame Operation System (COSMOS). Its primary job is to assign the best possible line equipment in the central office (*originating equipment*) to the outside plant equipment (typically *cable pair*). A cable pair is assigned by LFACS and passed to COSMOS through the information flow controller module. The corresponding originating equipment is assigned by COSMOS based on the central office load balancing and other criteria determined by the telephone operating company. COSMOS maintains the inventory for local wire center facilities and circuits in its database.

The assignment of the trunk-side equipment (the equipment dedicated to the trunks) within the central office and the assignments of the facilities between central offices is done by the TIRKS®system. The TIRKS system is composed of several software modules performing a variety of functions. It supports the facilities between central offices and the associated equipment within a central office, required to make the facilities work. It also supports equipment inventory, facility inventory, and circuit design.

Other systems exist in the provisioning environment that perform various tasks, in addition to facility assignments. The Service Order Analysis and Control system (SOAC) controls the work flow and orchestrates the flow of information among the components of the system. SOAC is a transaction based system and holds only temporary data. Routing and dispatching of work force is done by WFA/DO. It handles the routing and dispatching installation maintenance forces for the field, maps the job location, estimates the time to perform the work, categorizes the work to match technician skill, calculates dispatchs start date, and prioritizes dispatch jobs. Another system called the MARCH®system serves as a vehicle to associate service and/or custom calling features (e.g. 3 way calling) with the telephone number of the customer. The rest of the operation support systems are beyond the scope of this paper.

[6] The discussion here is that of a typical case.

TIRKS is a registered trademark of Bellcore.
MARCH is a registered trademark of Bellcore.

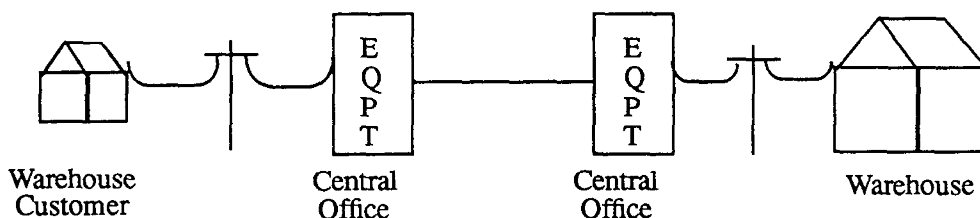

Figure 1: Foreign Exchange Service

3.2 Foreign Exchange Service: A class of service orders

The foreign exchange service is one example of a *special service order*. It differs from an ordinary service order in that it has two terminating ends (customer premise, and caller to the customer premise). Also it uses circuits and trunks that are dedicated to special service circuits. To understand the foreign exchange service, let's consider the following scenario.

The owner of a warehouse in Piscataway has customers in Morristown (Figure 1). Although both Piscataway and Morristown are within the service area of the same telephone operating company, the call between them is a toll call, and the warehouse customers would have to pay the usage sensitive fee. The warehouse owner requests a Foreign Exchange Service from the telephone company to arrange a special service so that the warehouse customers in Morristown would not be charged for their calls to the warehouse. Instead, the warehouse owner will pay a monthly fee for the special service.

The Foreign Exchange Service request is processed under the supervision of SOAC which interacts with multiple operation support systems. Figure 2 illustrates the simplified flow information. Basically, the process consists of submitting appropriate allocation and update requests to the various component systems and passing the information between the components through the controller. Complex execution dependencies exist between the individual requests executed by the independent but cooperating systems. After the assignments and the physical connections are made, the special service is available and the telephone company starts to bill the customer.

3.3 Modeling the Foreign Exchange Service Order Provisioning as a Flexible Transaction

A service request (transaction), which is forwarded to SOAC, has the characteristics of a Flexible Transaction. The foreign exchange service transaction corresponding to Figure 2 consists of ten subtransactions. A flexible transaction can be represented by a transaction graph, such as the one shown in Figure 3. The nodes of the transaction graph correspond to subtransactions ST_i of flexible transactions. A directed edge is drawn from ST_i to ST_j if an execution dependecy exists, requiring that ST_j cannot be exuted until ST_i completes successfully. Transactions with no incoming edges have no execution dependencies and are designated as *primary subtransactions*.

In our example, ST_1, ST_2 and ST_3 are primary subtransactions. Each subtransaction corresponds to one of the messages (with its corresponding response/reply message) shown in Figure 2). ST_1 is the Planning MSG to WFA/DO, ST_2 is MSG1 and its reply MSG1R involving the TIRKS system, and ST_3 is the Assignment Request to LFACS (that performs the assignments related to the loop serving the warehouse) and the reply response from LFACS. Upon the success of ST_2, ST_4 is scheduled. ST_4 is MSG2 to the TIRKS system and its response MSG2R. Upon the success of ST_3, ST_5 and ST_6 are triggered to provide Assignment Request to two COSMOSs corresponding to the two central offices. ST_5 and ST_6 are prepared based on the output of the ST_3 which is the assignment done by LFACS. Upon the success of ST_1, ST_4, ST_5, and ST_6, ST_7 through ST_{10} are scheduled for execution. ST_7 is the MSG3 to the TIRKS system, ST_8 is the Assignment MSG to WFA/DO, ST_9 is the Translation Packet to the MARCH system, and ST_{10} is the Assignment Section to SOP. Upon the success of ST_7 through ST_{10}, the Flexible Transaction commits. Therefore, this Flexible Transaction has one acceptable state which specifies the success of all ten subtransactions as the condition for its success.

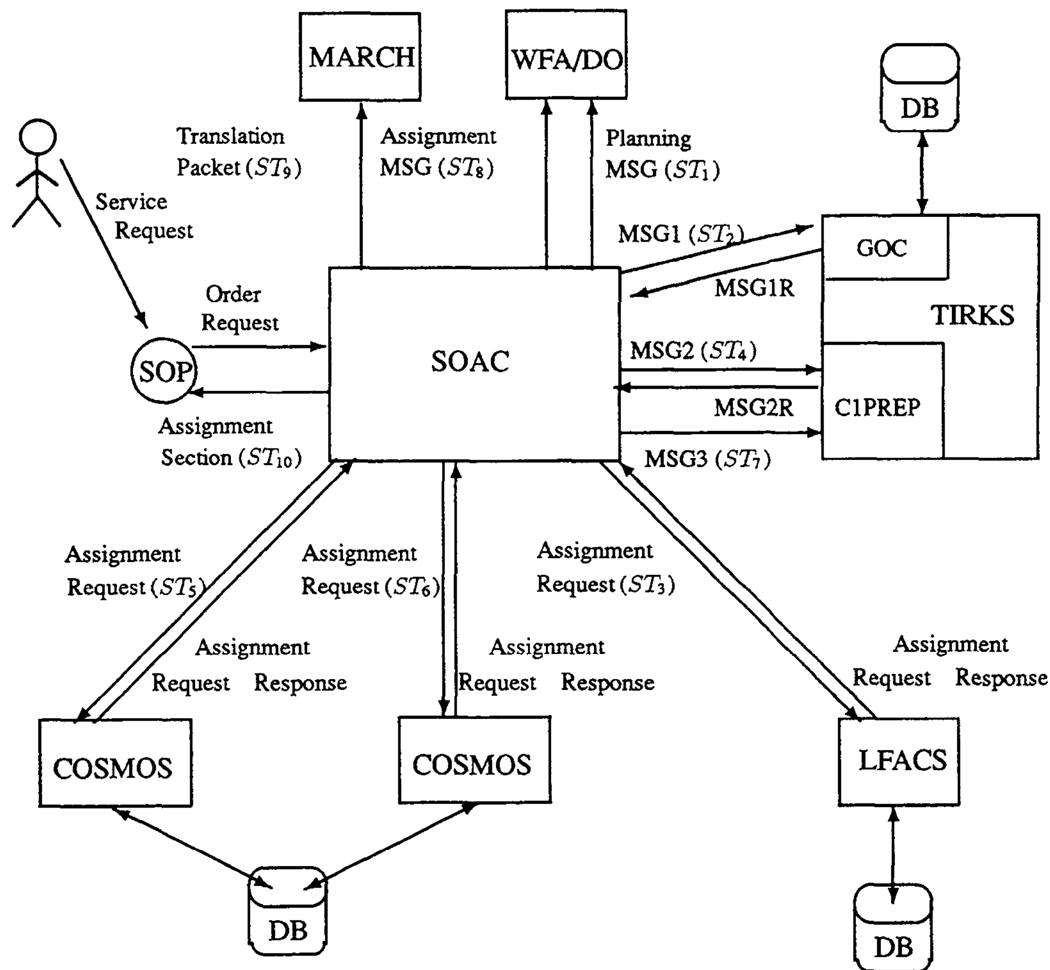

Figure 2: Foreign Exchange Service Order Provisioning

4 Implementing a Class of Service Orders Using the Flexible Transaction Paradigm

Our prototype testbed for foreign exchange service provisioning was developed in two steps. In the first step, a processing system for Flexible Transactions was implemented to receive the specification of a Flexible Transaction, schedule subtransactions and manage the control and data flow among subtransactions until the Flexible Transaction commits or aborts. This implementation is independent from the application discussed in Section 3 and is capable of processing any Flexible Transaction. A model for executing Flexible Transactions using a parallel Prolog-based query language is described in [KPE92]. However, no information about the actual implementation involving multiple application systems or databases is provided there. Implementation of our scheduler and the rest of the prototype is described in Section 4.

In the second step, a program module was implemented to analyze a telephone service request, generate the specification of a Flexible Transaction, and generate a set of subtransactions for local systems. Also, since the actual database systems used by the components of the service order processor were not accessed by the prototype implementation, models of these databases were created under a commercial relational database management system. These models captured only the data that were relevant to our application. Each of these databases are autonomous and are treated independently from other databases. The processing system for Flexible Transactions resides on a separate machine and no database has direct communication with other databases.

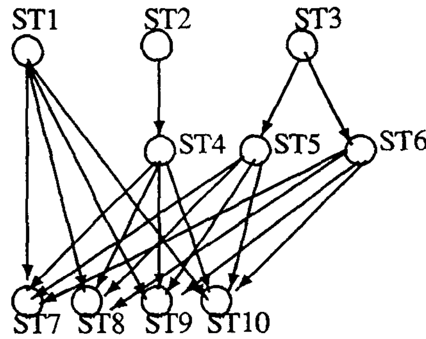

Figure 3: The intra-transaction execution dependency of a foreign exchange service transaction

4.1 Software Architecture

There are two major tasks in the implementation of a transaction processing system for Flexible Transactions [ARNS92]. The first task is to schedule subtransactions and determine the success or failure of the global transaction. The second task is to execute subtransactions in the local database systems. Figure 4 illustrates the architecture of PROMT (a PROcessing system for Multidatabase Transactions) prototype, which was developed to perform the above tasks.

The scheduling algorithm for Flexible Transactions is implemented using L.0 [CCG+91, Nes90], a language that allows concise specification of the scheduling constraints on the subtransactions. The scheduler receives the specification of a Flexible Transaction consisting of a set of subtransactions, their dependency set, and the set of acceptable states. The specification of a Flexible Transaction can be expressed in a pseudo language or through a graphical interface. In this case, the specification must be translated by a module to L.0 before it is passed to the scheduler.

To supervise the execution of the scheduled subtransactions, the Distributed Operation Language, DOL, is used [HAB+92]. DOL is designed to provide access to multiple and heterogeneous hardware and software systems. By interfacing L.0 and DOL, the scheduler can cooperate with the execution monitor in processing Flexible Transactions.

4.2 The Scheduler

L.0 is a rule-based language, which was designed to allow fast development of prototypes for software and hardware protocols [CCG+91, Nes90]. Such protocols constrain the behavior of a number of different agents or components to achieve a common goal. Among such common goals are reliable transmission of data, fair resource allocation, recovery from an error state, correct execution of a hardware circuit, and success or failure of a Flexible Transaction.

Often these protocols are stated as sets of *guarded commands* [Dij75] (or rules). Each set of guarded commands specifies the behavior of a particular agent or component. In the case of Flexible Transactions, each subtransaction may be viewed as an agent. The Flexible Transaction itself may be viewed as a protocol for coordinating the behavior of each of these subtransactions. Thus, the algorithm for processing Flexible Transactions can be implemented via a parameterized set of guarded commands, which is instantiated once for each specification of a Flexible Transaction.

The fundamental semantics of L.0 is the *synchronous* execution of the guarded commands. Each guarded command is composed of a guard (a set of predicates), and a set of actions to be taken once the guard becomes true. The guards in L.0 are referred to as *causes*, and the actions are referred to as *effects*. L.0 provides primitives to activate or deactivate a set of cause-effect rules in each *synchronization step*.

Each synchronization step is composed of two *phases*. In phase one, all the causes in the set of *active* cause-effect rules are evaluated. The effects of the *true* causes are executed at the next synchronization step(s). In phase two, all of the effects with true causes (which are evaluated in previous steps) are executed. The execution of these effects appear to be simultaneous to the user.

For example in the scheduler of Flexible Transactions, the execution dependencies of the subtransactions are implemented using **whenever** cause-effect rules. For each subtransaction, there is one guarded command of the form:

```
whenever
  <precondition for execution> &
```

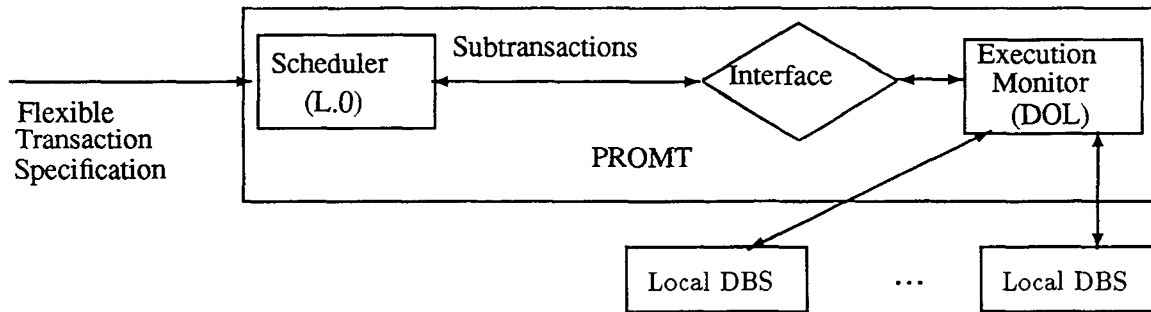

Figure 4: The Architecture of PROMT

```
  <the execution state of the
   subtransaction is "not_executing">
then
  <assign state of subtransaction
   to be "executing"> &
  <invoke DOL to start execution
   of that subtransaction>;
```

This type of cause-effect rule implies that at each step, all subtransactions which have not been executed and whose preconditions (their execution dependency) are true, are scheduled for execution. By using a *quantifier* for the above cause-effect rule, the cause-effect rule is executed for all subtransactions.

The predicate to determine the success or failure of the Flexible Transactions is implemented using the **until** deactivators. The **until** deactivator, which is a cause-effect rule, can be used to remove one set of rules and activate other set(s) of rules. In the scheduler, two **until** deactivators are used. The cause of one deactivator specifies the conditions for the success of the Flexible Transaction. The cause of the other deactivator specifies the conditions for the failure of the Flexible Transaction. Once the cause of one of the two **until** deactivators becomes true, all other rules specified by the **whenever** rule and the other **until** rule, are deactivated. Upon the completion of the effect of the **until** rule, it is also removed from the set of active rules and the execution stops.

Thus, the scheduler of the Flexible Transactions is implemented using a quantified **whenever** rule and two **until** rules. These rules form an L.0 procedure. To process a Flexible Transaction, the specification of the Flexible Transaction, specified in (or translated to) L.0, is passed to the scheduler capsule. The scheduler capsule then processes the transaction, and upon termination of the transaction, it returns the status of the transaction. The status specifies whether the Flexible Transaction has committed or aborted.

Using L.0 to implement the scheduler has several advantages. The basic idea underlying L.0 is synchronous execution of quantified guarded commands. The synchronous execution allows modeling of maximal parallelism [CNS91]. The parallelism may further be restricted according to the dependency constraints of the subtransactions and the limitations of the execution environment for the subtransactions.

Another important advantage is that it simplifies the implementation. 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 can be expressed easily using L.0 data structure. Interfacing L.0 to DOL is straight-forward, since L.0 provides the facility to call functions written in C.

4.3 Subtransaction Execution Monitor

The subtransactions are described using the Distributed Operation Language (DOL) [HAB+92]. DOL can be used to specify a distributed execution of a global application in a heterogeneous computing environment. A DOL execution environment consists of *Execution Engine, Service Directory,* and *Local Access Managers (LAMs).* The architecture of the DOL system is illustrated in Figure 5.

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

A LAM acts as a proxy user for the software sys-

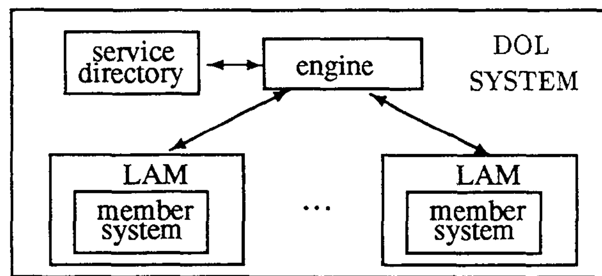

Figure 5: The Architecture of DOL system

tem 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, a LAM must be designed for it and its access information, such as its network address, must be added to the Service Directory.

4.4 Interfacing the Scheduler and the Execution Monitor

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 6.

Four C functions, *DoTrans*, *GetState*, *CommitTrans*, and *AbortTrans* were developed to interface L.0 and DOL. *DoTrans* starts the Interface process and establishes the communication channel with it. It then 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 Interface process to finish, it returns back to the scheduler. It returns the communication information of the established channel.

GetState reports the current state of a subtransaction, upon a request from the scheduler. It checks whether there is any returned result from the Interface or the LAM. If there is any new result of the executing subtransaction, that is *LocalCommit*, *LocalAbort* or *PreparedToCommit*, it reports it back to the scheduler. Otherwise, it reports *Executing* as the current state of the subtransaction. If the subtransaction has locally committed or aborted, it closes the channel 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 to send a commit or abort message later.

CommitTrans and *AbortTrans* are used for the subtransactions that 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 Interface is a process started by *DoTrans*. It creates a child process to execute a subtransaction and waits for the result. 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 the case of a success, Interface records any output in a file and reports local commit to the scheduler. 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.

5 Conclusions and Future Work

Many applications and databases that were designed to operate as stand-alone systems need to become interoperable to support multi-system applications. Service order provisioning is one example of such a multi-system application which is fundamental to the telecommunications business. The automation of the provisioning will allow telephone operating companies to make new services available to customers

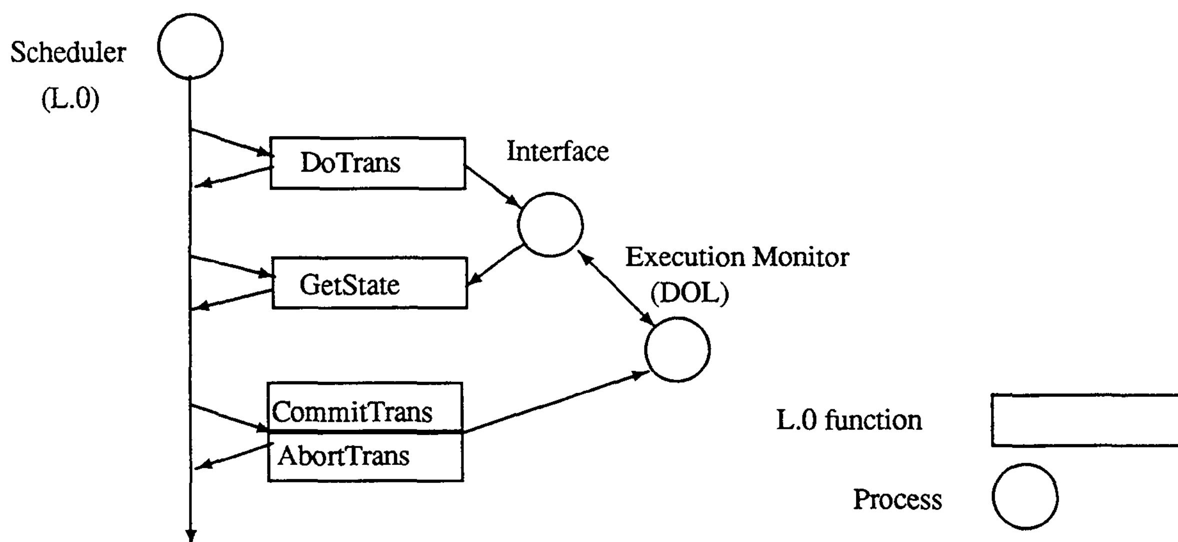

Figure 6: Interfacing L.0 and DOL

faster, i.e. in minutes, rather than in days or weeks.

One conclusion of the project described in this paper is that the Flexible Transaction model was found to be useful for specifying the control and data flow in service order provisioning. Work flows for many telecommunications services are qualitatively similar to the work flow considered here; hence Flexible Transactions for many services can be defined similarly. The use of this model facilitates specification of provisioning at a higher level of abstraction, making the provisioning of various services easier to understand. The declarative and high-level specification of work-flow control as dependencies and success/failure conditions, as compared to hard coded flows in application code has significant advantages. The work-flow can be changed independently of the subtransactions that perform a specific activity or request specific functions from other systems. A subtransaction that performs one type of activity can be used in multiple work flows that require performing the same type of activity. Since the Flexible Transaction model permits concurrent execution of subtransactions at different sites, it promises improved efficiency by exploiting parallelism among subtransaction executions. The model also allows easy addition of more systems and tasks to service order provisioning with new demands. The abstraction of the work and data flow could, in the future, permit automated verification of correctness.

Another conclusion that might be drawn from this project is that proposed multidatabase transaction models (e.g. the Flexible Transaction model), which relax traditional requirements such as atomicity and isolation and even the correctness criterion of serializability are practical, and have, in fact, been in use informally for years by "industrial" applications.

One of the obstacles to efficient provisioning of services is that requests for manual assistance due to errors, failures and data inconsistencies may be generated throughout the provisioning process. These scenarios must be identified and their handling must be automated. Our experiences indicate that provisioning and many other flow-through processing applications could be defined and supported efficiently using multidatabase transaction models. However, it remains an open research problem to determine if the Flexible Transaction Model is expressive enough to naturally remove needs for manual assistance.

The project is currently continued and its scope has been expanded. In particular, we are studying a number of research issues including the definition of a suitable correctness criterion for concurrent execution of multiple Flexible Transactions and the development of a recovery mechanism for Flexible Transactions.

Acknowledgement: We wish to thank Peter J. Stein from Bellcore for his invaluable assistance. He provided us with an abstract view of service order provisioning and suggested the foreign exchange service as an appropriate example. We thank Len Castelli, Steve Melville, and Patty Murray for their helpful comments on earlier drafts. We also thank James McKenna for his support of this project.

References

[ARNS92] M. Ansari, M. Rusinkiewicz, L. Ness, and A. Sheth. Executing Multidatabase Transactions. *Proceeding of 25th International Conference on System Sciences*, Vol.II, January 1992.

[Ans92] M. Ansari. Executing Flexible Transactions in a Multidatabase Environment. Master thesis, Department of Computer Science, University of Houston, 1992.

[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 International Workshop on Interoperability in Multidatabase System*, April 1991.

[CR90] P. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. *Proceedings of ACM SIGMOD*, May 1990.

[CR91] P. P. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Models. *Proceedings of the 17th VLDB*, September 1991.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. A Transactional Model for Long-Running Activities. In *Proceedings of the 17th VLDB*, September 1991.

[Dij75] E. Dijkstra. Guarded Commands. *Communications of ACM* 18, 8, August 1975.

[DE89] W. Du and A. Elmagarmid. Quasi Serializability: A Correctness Criterion for Global Concurrency Control in InterBase. *Proceedings of the 15th VLDB*, August 1989.

[Elm92] A. Elmagarmid, Ed. Transaction Models for Advanced Database Applications, Morgan-Kaufmann, February 1992.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for Interbase. *Proceedings of the 16th VLDB*, August 1990.

[GGK+90] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating multi-transaction activities. Technical Report CS-TR-247-90, Princeton University, February 1990.

[GS87] H. Garcia-Molina and K. Salem. Sagas. *Proceedings of ACM SIGMOD*, 1987.

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

[HAB+92] 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 Multi-System Applications. *Proceedings of the Second International Conference on Systems Integration*, 1992.

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

[Kle91] J. Klein. Advanced Rule Driven Transaction Management. *Proceedings of the COMPCON Spring*, February 1991.

[KLS90] H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. *Proceedings of the 16th VLDB*, August 1990.

[KPE92] E. Kuehn, F. Puntigam, and A. Elmagarmid. An Execution Model for Distributed Database Transactions and Its Implementation in VPL. *Proceedings of the Third International Conference on Extending Database Technology*, March 1992.

[LER90] Y. Leu, A. Elmagarmid, and M. Rusinkiewicz. An Extended Transaction Model for Multidatabase Systems. Technical Report, Computer Sciences Department, Purdue University, 1989.

[LKS91] E. Levy, H. Korth, and A. Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. *Proceedings of the ACM SIGMOD*, May 1991.

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

[LT88] W. Litwin and H. Tirri. Flexible Concurrency Control Using Value Dates. Technical Report 845, INRIA, May 1988.

[Mos85] J. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press, Cambridge, MA, 1985.

[Nes90] L. Ness. Issues Arising in the Analysis of L.0: A Synchronous Executable Temporal Logic Language. *Proceedings of the Workshop on Computer-Aided Verification*, June 1990.

[Reu89] A. Reuter. Contracts: A Means for Extending Control Beyond Transaction Boundaries. *Presentation at Third International Workshop on High Performance Systems*, September 1989.

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

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