1990

A Parallel Logic Language for Transaction Specification in Multidatabase Systems

Eva Kuhn

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Yungho Leu

Nourreddine Boudriga

Report Number:
90-1031

A PARALLEL LOGIC LANGUAGE FOR TRANSACTION SPECIFICATION IN MULTIDATABASE SYSTEMS

Eva Kuhn
Ahmed K. Elmagarmid
Yungho Leu
Noureddine Boudriga

CSD-TR-1031
October 1990

A Parallel Logic Language for Transaction Specification in Multidatabase Systems

Eva Kuhn
Institute of Practical Informatics
Technical University of Vienna

Ahmed. K. Elmagarmid and Yungho Leu
Department of Computer Science
Purdue University

Noureddine Boudriga
Faculty of Science of Tunis
University of Tunis II

Abstract

The realization of truely heterogeneous database systems is hampered among others by two obstacles. One is the unsuitability of traditional transaction models, this has lead to the proposal of a new more flexible transaction model. The second is the fact that none of the existing language and system support such a flexible model. This paper addresses these two issues by proposing a logic approach to the integration of database systems.

1 Introduction

One of the consequences of the information explosion taking place in society is the emerging need to access heterogeneous and isolated data repositories. Because of this isolation, it becomes more difficult for programmers to write global applications which make full use of the data and resources at their disposal because the systems that they need to use are not integrated.
Flex transactions provide for additional capabilities originally not foreseen in traditional transactions. These new capabilities are required in order to describe applications in a multidatabase environment [ELLR90] [LEB90]. Flex transactions provide the following features:

Alternative: Alternative ways by which a specific task can be performed can be conveniently stated in the Flex model. Traditionally, all tasks stated as a part of a transaction must be performed. Using alternatives more than one equivalent task are stated and it is left to the application designer to finally choose which one to finally commit.

Control isolation: Transactions are traditionally non-compensatable and once they are committed, their effects are preserved by the system. Flex transaction model allows transactions to include some subtransactions that are compensatable and others which are not. This results in what we call mixed transactions. By properly mixing both compensatable and non-compensatable subtransactions, we can control the isolation of the global transaction to

any desired granularity. Those subtransactions which are non-compensatable must run in isolation of the rest of the system, while the compensatable subtransactions can reveal their results before their parent global transaction commits.

Dependency: The model allows for specifying functions that can be used to influence the execution of the transaction. These functions consider external parameters to the transaction or subtransactions. The model also allows for specifying dependency among subtransactions.

In order for this new transaction model to be implementable in a multidatabase system, a language that can be used for describing it must be introduced. It must be expressive and includes constructs to allow the specification of predicates.

Several languages for describing multidatabase activities exist in the literature. A Distributed Operational Language (DOL) was introduced in [ROEL90]. It serves as a task specification and execution language for multidatabase activities. DOL is intended as a low level language. DOL along with the execution environment built for it [ER90] constitutes a low level machine that can be used to perform tasks distributed over disparate and autonomous sites. In [LAN+87], a multidatabase language is described. MSQL is an extension of SQL which allows a user to specify queries on multiple autonomous databases. Though no full implementation of MSQL exists, it constitute the first high level language that supports fully autonomous databases without requiring global schema to be constructed. MSQL like SQL does not provide for the expression of complex transactions like those envisioned by Flex. In [KL88], a system based on logic programming was described. VIP-MDBS (Vienna Integrated Prolog Multidatabase) extends Prolog by using semantic relations. It provides all features available in MSQL in addition to recursive data representation and transitive closure. VIP-MDBS is used to describe activities over autonomous databases.

These multidatabase languages do not support transaction specification and do not provide parallel constructs. The objective of this paper is to define a language for the specification of Flex transactions. The language can be considered as an extension to the ones described above and is directly executable. This paper goes beyond the introduction of the parallel language. It describes a multidatabase architecture and the execution environment built for this language. This implementation is currently underway.

This paper is organized as follows. Section 2 presents background material for Flex transactions and logic. In section 3, a parallel logic language for specification of Flex transactions is defined. Section 4 constructs a syntactic mapping for the design of parallel logic transaction programs. The implementation of Parallel Logic Transaction Language (PLTL) and execution of a Parallel Logic Transaction Program are summarized in section 5. Section 6 concludes this paper.

2 Background

2.1 Flex transaction issues

As it is defined in [LEB90], a Flex transaction FT is a set of subtransactions, say

$$FT = \{t_1, t_2, \cdots, t_n\}$$

accompanied with the following predicates and functions:

1. A (partial) ordering relation on FT, called success order and denoted by $\prec_S$, is defined by:

   $\forall$ *i, j:* $t_i \prec_S t_j$ *iff the execution of* t_j *depends on the success of* t_i.

2. A (partial) ordering relation on FT, called failure ordering and denoted by $\prec_F$, is defined by:

   $\forall$ *i, j:* $t_i \prec_F t_j$ *iff the execution of* t_j *depends on the failure of* t_i.

3. External predicates such as time or cost. These predicates define the execution dependency of subtransactions on external parameters.

4. An acceptability function which is an n_ary predicate on FT. This function allows the user to specify what subtransactions he accepts for commitment.

To illustrate the notion of Flex transaction, we use the following travel agent example.

Example 2.1 *When scheduling a travel package for a user, the travel agent has to performs three tasks:*

1. TA negotiates with airlines for flight tickets.

2. TA negotiates with car rental companies for car reservations.

3. TA negotiates with hotels to reserve rooms.

These three tasks can be decomposed respectively as the three sets of subtransactions $\{t_1, t_2\}$, $\{t_3\}$ *and* $\{t_4, t_5\}$, *where*

t_1 *Order a ticket at Northwest Airlines;*
t_2 *Order a ticket at United Airlines,*
t_3 *Rent a car at Hertz;*
t_4 *Reserve a room at Hilton,*
t_5 *Reserve a room at Sheraton.*

The constraints that we assume in executing the subtransactions are (1) t_1 depends on the failure of t_2; (2) t_3 depends on the success of both t_1 and t_2 and (3) t_1 and t_2 must be executed between 8 AM and 5PM. We assume that customers accept a travel package if he/she can get a ticket, a car reservation and a hotel reservation.

This transaction can be specified in the Flex transaction model as follows.
$FT = \{t_1, t_2, \cdots, t_5\}$,

Internal dependency $\{\prec_S, \prec_F\}$, *External dependency* $\{p\}$,
Where
$\prec_S$: $t_1 \prec_S t_3$, $t_2 \prec_S t_3$,
$\prec_F$: $t_1 \prec_F t_2$,
p: $p(t_i)$ *is true if* $i = 1$ *or* 2 *and* $8:00 < actualtime(t_i) < 17:00$
$f(t_1, t_2, \cdots, t_5) = ((t_1 = S) \vee (t_2 = S)) \wedge (t_3 = S) \wedge ((t_4 = S) \vee (t_5 = S))$
The expression $(t_i = S)$ *stands for "*t_i *is successfully executed".*

In the definition of a Flex transaction, the notion of execution state appears important for capturing the notion of dependency. We define the execution state x_i for subtransaction t_i as

$$x_i = \begin{cases} N & \text{if subtransaction } t_i \text{ has not been submitted for execution;} \\ E & \text{if } t_i \text{ is currently being executed;} \\ S & \text{if } t_i \text{ has successfully completed;} \\ F & \text{if } t_i \text{ has failed or completed without acheiveing its objective;} \end{cases}$$

An execution state of a Flex transaction FT is an n-tuple $(x_1, \cdots, x_n)$ representing the execution state of FT. An execution of FT is a sequence of execution states beginning with the state $(N, \cdots, N)$ leading to an acceptable state x (with $f(x) = 1$), where n is assumed to be the number of subtransactions in FT.

2.2 Logic Language Issues

Logic offers a framework for concise description of database transactions, provides tools for reasoning on them and supports dependency among these transactions. The logic description of subtransactions is easy because they consist of sequences of reading and writing actions.

Simply, a logic program consists of a set of logic axioms of the form

$$A :- B_1, B_2, \cdots, B_n \quad n \geq 0$$

Such an axiom can be read in two manners. First, declaratively, saying that A is true if B_1, B_2, $\cdots$, B_n are true. Second, procedurally, saying that to prove A one can prove B_1, B_2, $\cdots$,

B_n. An execution is a proof of existentially quantified goal statement using these axioms. If all solutions which satisfy the goal are required, it is necessary to leave open deterministic choices when examining one possible solution.

Sequential Prolog uses the order of goals in a clause and the order of clauses in the program to control the search for a proof. In sequential Prolog, the leftmost goal is always chosen, and the non-deterministic choice of clauses is done by sequentially searching and backtracking through the execution of the program. This description suggests the following form of parallel execution:

AND parallelism: the reduction of several atoms can be done in parallel.

OR parallelism: the search of clause can be done in parallel.

Two approaches can be used to exploit the parallelism of a logic program. The first approach consists of parallelizing the interpreter. In the second approach, the programmer uses explicit language constructs to express concurrency and synchronization. The semantics of such languages differs from the semantics of sequential Prolog.

In general, a parallel Prolog program is a finite set of guarded clauses [Ehu87]. A guarded clause is a universally quantified axiom of the form:

$$Head \text{ :- } G_1, G_2, ..., G_n \mid B_1, B_2, ..., B_m,$$

An execution of a parallel program on a goal G proceeds by reducing subgoals of G by searching concurrently the clauses of the program that match with a clause subgoal. At each choice point, it is possible to decide among alternatives: however, having made a choice (or commitment), no other alternative can be used. In conclusion, no backtracking operator is used. For this reason, we call the | operator the *don't care* committment operator. The parallel Prolog language described above has a lot of advanced features, but also deficiencies. In particular, the semantics of the *don't care* commitment operator may cause problems—especially in the database context. When retrieving tuples of a relation, in most cases, it is not sufficient to just produce one solution; backtracking over all solutions is usually required. In particular, to capture the concept of alternatives in Flex transactions, it is necessary to try several clauses that are able to perform the same task. A solution to reduce this deficiency is to include a commitment operator that allows backtracking by not killing all other candidates running using the OR-parallel construct. In [Sar88] such a non-deterministic *don't know* commitment operator has been investigated and its semantics have been described.

Another disadvantage of the don't care commitment operator is that the semantics of "the first clause that reaches its commit point is taken" is sometimes hard to understand, especially in case of non-disjoint guards. Nevertheless, the don't care commitment operator has its own merits. First, it may be considered as a symmetric cut operator to control the procedural semantics. Second, it is cheaper to implement than the don't know commitment operator. Finally, it avoids backtracking which is a non-logical feature.

Our approach in this paper is to define an extension of Prolog with sufficiently powerful constructs. We adopt the *don't know* operator and use a feedback construct that allow the user to

decide a solution found to be acceptable.

3 Parallel Logic Transaction Language

3.1 Basic Concepts and Syntax

In order to have a language that is powerful enough to express any kind of control flow, e.g. Flex transaction model as defined by [LEB90], the language must provide means to represent sequential as well as parallel execution of processes. The Parallel Logic Transaction Language that we present below extends sequential Prolog and allows the user to compose MDBS queries. The syntax of the PLTL is a superset of the syntax of Prolog. It includes the following well known constructs:

'&&' ..... parallel AND operator in the guard or body of a clause
';;' ..... parallel OR operator in the guard or body of a clause
'||' ..... non-deterministic don't know commitment operator
'?' ..... read-only annotation for a variable
'↓' ..... write-only annotation for a variable (optional)

In addition to the predicates supported by Prolog, the following predicate symbols are proposed:

'Remote_call'... communication primitive,
'Feedback' ... primitive to allow user to select alternatives.

As it is explained in the following subsections, remote_call serves to access local sites and feedback serves to stop the search for solutions whenever an acceptable solution is reached.
A clause in the proposed PLTL is a universally quantified formula of the form:

$$A \ :- \ G_1, G_2, \cdots, G_m \ || \ B_1, B_2, \cdots, B_n$$

Where A and B_is are atoms.

In order to explain the meaning of PLTL syntax, we consider the following example where each local database provides some predicates.

Example 3.1 *The parallel logic transaction language that we present in this example describe a travel agent transaction. It is constituted by five local systems and allows customers to book flight, rent cars and reserve hotel room.*
Predicate Symbols

DB1 / United Airlines database */*
flights(From, To, FlightNr, Price).
seats(FlightNr,SeatNr,ClientName, Date).

DB2 / NorthWest Airlines database */*
ftab(From, To, FlightNR).
prices(FlightNr, Price).
seats(SeatNr,FlightNr,ClientName, Date).
foreign_flights(Airline, Flno, From, To).

DB3 / Hertz car rental company database */*
cars(CarType, CarNr, Price).
book_car(CarNr, ClientName, Date).

DB4 / Hilton hotel database */*
rooms(RoomId, Price).
reserve(RoomId, Date, ClientName).

DB5 / Ramada hotel database */*
rooms(RoomId, Price).
reserve(RoomId, Date, ClientName).

This example presents predicates used by a multidatabase system (with five databases). These predicates access (reading or updating) the databases. Reading predicates, such as flights(From, To, FlightNr, Price) in DB1, have only reading variables. While updating predicates, such as seats(FlightNr,SeatNr,ClientName, Date) can update a database via a writing variable (e.g. seats predicate inserts a tuple in the table (FlightNr,SeatNr,ClientName, Date) of DB1).

3.2 Operational Semantic

A PLT program is a set of guard clauses of the form

$$A \ :- \ G_1, G_2, \cdots, G_m \ || \ B_1, B_2, \cdots, B_n$$

The execution of a PLT program on goal G proceeds as follows

- a subgoal of G is chosen,
- parallel search for matching clauses with the subgoal is done,
- all matching clauses with true guards are considered,
- one clause is chosen to reduce the subgoal, and the other queued.

When the reduction is done, the new goal is stored instead of G. The process is then repeated. Whenever a solution is found or the set of matching clause is empty a backtracking is used to search other solutions.
We explain, now, the meaning of constructs that extend Prolog into PLTL. Especially, we describe the remote_call and feedback predicate.

The primitive remote_call serves to send a subtransaction to a local site for execution. It is a three argument predicate of the form

$$remote\_call(DBName, stdin([t]), stdout(t)).$$

The first argument of this predicate designates the database system where a subtransaction is executed. The second argument, is a list where the subtransaction is written in Prolog atoms or SQL primitives. The last argument, stdout(t) is a list with head equal to s if the execution of the subtransaction is successful and f otherwise; the rest of the list contains the reply or an error messages. An example of the use of remote_call predicate is given by

$$remote\_call(DB2, stdin([ftab(chicago, indianapolis, N), price(N, P), P \leq 100]), Y)$$

is true if Y is a list of the form $[s, (chicago, indianapolis, n, m)]$. Where n is the flight number of a specific flight from Chicago to Indianapolis with cost $m \leq \$100$, or $Y = [f]$.

The *feedback* predicate is a two argument predicate of the form

$$feedback(One\_solution, Commit).$$

It allows the user to decide, whenever a solution is found, to accept the solution and to stop searching for another solution. The argument *Commit* is a boolean variable handled by the user. The following example shows the use of a feedback predicate.

Example 3.2 *The Prolog clause*

$$feedback(X, C) : -r(X), write(X), user\_decision(C)$$

can be read as follows: X is an acceptable solution if X is a solution of predicate r and the user decision is equal to 1 and *write* is a predefined predicate allowing the display of X.

4 Parallel Logic Transaction Programming

4.1 Logic model for Flex transactions

As it is assumed in the preceding section, write and read action are performed by the use of local predicates belonging to the local site. Because of the definition of subtransactions, we can assume, without loss of generality, that a subtransaction is a conjunction of atoms. For example, the conjunction

$$cars(ford, CarNr, Price) \wedge book\_car(CarNr, abc, Date) \wedge (price < 100)$$

means "rent a Ford car for customer abc with price less than $100.
Let $(FT, \prec_S, \prec_F, p, fu)$ be a Flex transaction where FT is a set of subtransactions, say $FT =$

$\{t_1, t_2, \cdots, t_n\}$, $\prec_S$ is the success ordering on FT, $\prec_F$ is failure ordering on FT, p is the set of external predicates on FT and fu is the acceptability function. The definition of a parallel logic transaction program for a Flex transaction can be constructed by the following four steps:

1. represent a subtransaction in PLTL.
   A subtransaction t_i is represented by

   $$remote\_call(DBNr, stdin([t_i]), stdout(t_i))$$

   where DBNr is a local system in which t_i has to be executed, stdin($[t_i]$) is a list where the atoms of t_i occur, and stdout(t_i) is a list of arguments. The first argument reveals the execution state of t_i, the remaining arguments contain an output as the reply to the request of stdin($[t_i]$).

2. represent external dependency.
   Let p be an external predicate on a Flex transaction. For simplicity, the evaluation of p on a subtransaction t_i is denoted by $p(t_i)$. We represent a subtransaction t_i with an external predicate p by:

   $$remote\_call(DBNr, stdin([t_i, p(t_i)]), stdout(t_i)).$$

   where list [t_i, p(t_i)] is the concatenation of atoms of t_i followed by atom p(t_i).

   Example 4.1 *The expression*

   $$remote\_call(DB2, stdin[ftab(From, To, FlightNr, Price), (price < \$100]), stdout(t_i))$$

   indicates a price ceilling of $100 for the flight.

   Because p(t_i) appears in the stdin argumentlist, the above expression means that the local system where subtransaction t_i is executed has to enforce the predicate p(t_i). Whereas in the following expression

   $$p(t_i) \parallel remote\_call(DBNr, stdin([t_i]), stdout(t_i)).$$

   indicates that p(t_i) is enforced by the interoperability component.

3. represent success and failure dependencies.
   Consider the success order. Suppose that t_i is success dependent on subtransaction t_j and failure dependent on subtransaction t_k. We construct the following expression.
   (stdout(t_j) = s), (stdout(t_k) = f) || remote_call(DBNr, stdin[t_i], stdout(t_i)),
   where stdout[t_j] = s is true if the head of [t_j] is s.

4. represent the acceptability function.
   We suppose that fu is written in its normal conjunctive form (we did not consider the disjunctive form because of its similarity). Let $S_1, S_2, \cdots, S_k$ be the disjunctive components of fu, i.e.

   $$fu(t_1, t_2, \cdots, t_n) = S_1 \wedge S_2 \wedge \cdots \wedge S_k$$

Where

$$S_i = (stdout(t_{i1}) = s) \vee \cdots \vee (stdout(t_{is}) = s).$$

We then associate with fu the following universally quantified predicate:

$$ta\$transaction() : -ta\$task\_S_1, ta\$task\_S_2, \cdots, ta\$task\_S_k()$$

With each S_i, we associate the following clauses:

$$ta\$task\_S_i : - \ G_1 \ || \ remote\_call(DBN_{i1}, stdin([t_{i1}]), stdout(t_{i1})).$$
$$\vdots$$
$$ta\$task\_S_i : - \ G_s \ || \ remote\_call(DBN_{is}, stdin([t_{is}]), stdout(t_{is})).$$

where $t_{i1}, \cdots, t_{is}$ are the predicate occuring in S_i and DBN_{ij} is the name of the local site where t_{ij} is executed. G_k is the conjunctions of atoms containing the internal dependency on which t_i depends (and possibly external predicates that have to be checked by the interoperability component).

4.2 Example

In this subsection we reconsider the travel agent transaction presented in example 2.1. First, a partial solution is given where all dependencies are omitted. Second, a complete solution is presented.

Example 4.2 *Solution without dependencies:*

```
ta$reservation(Clientname,Date,From,To,Fprice,Cprice,Hprice,Car) :-
        ta$book_flight(ClientNm,Date,From,To,Fprice),
        ta$rent_car(ClientNm,Date,From,To,Car,Cprice),
        ta$reservate_hotel(ClientNm,Date,From,To,Hprice),

ta$book_flight(Nm,D,F,T,P) :- true ||
        remote_call(DB1, stdin([flights(F,T,Fno,P),seats(Fno,Nm,D)]),stdout(t1)).
ta$book_flight(Nm,D,F,T,P) :- true ||
        remote_call(DB2, stdin([ftab(F,T,Fno), prices(Fno,P), seats(F,Nm,D)], stdout(t2)).

ta$rent_car(Nm,D,From,To,Clyp,Price) :- true ||
        remote_call(DB3, stdin([cars(Clyp,Cnr,Price), book_car(Cnr,Nm,D)], stdout(t3)).

ta$reservate_hotel(Nm,Date,From,To) :- true ||
        remote_call(DB4, stdin([rooms(Rid,Price),reserve(Rid,Date,Nm)]),stdout(t4))
ta$reservate_hotel(Nm,Date,From,To) :- true ||
        remote_call(DB5, stdin([rooms(Rnr,Price),book_room(Rnr,Date,Nm)]),stdout(t5)).
```

Example 4.3 *Solution with internal dependencies:*

We assume that internal dependencies in the Flex transaction are those considered in 2.1 the PLTP associated to this Flex transaction is

ta$reservation(Clientname,Date,From,To,Fprice,Cprice,Hprice,Car) :-
ta$book_flight(ClientNm,Date,From,To,Fprice),
ta$rent_car(ClientNm,Date,From,To,Car,Cprice),
ta$reservate_hotel(ClientNm,Date,From,To,Hprice).

ta$book_flight(Nm,D,F,T,P) :- actualtime(T), 8≤T, T≤ 17 ||
remote_call(DB1,stdin([flights(F,T,Fno,P),seats(Fno,Nm,D)]),stdout(t_1))
ta$book_flight(Nm,D,F,T,P) :- actualtime(T), 8≤T, T≤ 17, (stdout(t_1) = f) ||
remote_call(DB2, stdin([ftab(F,T,Fno), prices(Fno,P), seats(F,Nm,D)], stdout(t_2)).

ta$rent_car(Nm,D,From,To,Ctyp,Price) :- (stdout(t_1) = s)||
remote_call(DB3, stdin([cars(Ctyp,Cnr,Price), book_car(Cnr,Nm,D)], stdout(t_3)).
ta$rent_car(Nm,D,From,To,Ctyp,Price) :- (stdout(t_2) = s)||
remote_call(DB3,stdin([cars(Ctyp,Cnr,Price), book_car(Cnr,Nm,D)], stdout(t_3)).
with the ta$reservate_hotel clause of the above example kept unchanged.

Reading procedurally the first clause of this program gives: a reservation can be made if a flight is booked, a car is rent and a hotel is reserved. Reading the third clause gives: flight is booked from United airline if a subtransaction is executed between 8Am and 5PM, and a seat is allocated in some flight.

4.3 Extensions

In this subsection, we show how the notion of commitment and compensatability can be captured in our logic framework. These predicates are of some importance. For this purpose, we use t_i^C to denote a compensatable subtransaction, and $\text{-}t_i$ to denote the compensating transaction of t_i when t_i is compensatable.
We define:

1. A predicate *commit* by
   $commit(t) \ :- \ remote\_call(db(t), stdin[t], stdout(t)), (stdout(s) = s), compensatable(t)$
   $commit(Flex\_transaction) \ :- \ feedback(X, C), C = 1$
   $commit(t) \ :- \ commit(flex\_transaction), \neg compensatable(t)$

   and saying first that a compensatable subtransaction has to commit when it is successfully executed; second that the Flex transaction commits if an acceptable solution is found; and

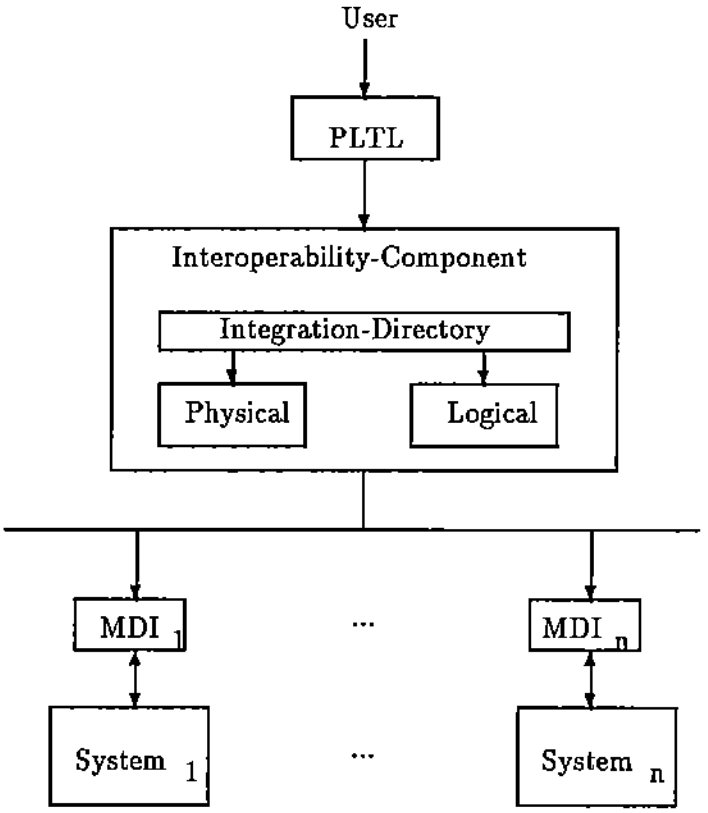

Figure 1: An architecture for multidatabase systems

last, a non-compensatable subtransaaction is committed if the Flex transaction is committed.

2. A predicate *compensate* saying that a compensatable subtransaction has to be compensated when a solution is found for the Flex transaction and the user rejects it.

5 PLTP-based system description

5.1 An Architecture for MDBS

A multidatabase system interconnects multiple pre-existing database systems in order to support global applications. This section presents the architecture used in the propsed system. This architecture is based on that originally used in the existing InterBase prototype [ER90].

As can been seen in the figure, the system is composed primarily of an interoperability component and an integration directory. On top of the existing systems to be integrated, the system incorporates an interface called the multidatabase interface (MDI). The PLT language is used as a high level user interface. The system componenets are briefly described in the following:

1. **Parallel Logic Transaction Language**: the syntax and operational semantics of PLTL have been explained in the previous sections. Multidatabase transactions are specified by the user in PLTP and then submitted to the interoperability component for execution.

2. The **interoperability component** (IC) is responsible for query processing and transaction management for global applications. PLTL is the implementation language of the interoperability component; this means that queries written in PLTL can be directly processed by the interoperability component and executed by local systems.

3. The **integration directory** (ID) contains static information that is used by the interoperability component for the integration of heterogeneous systems. It is subdivided into two subdirectories. The physical integration subdirectory contains information dealing with the physical heterogeneity of systems (e.g, login to a local system, networks/protocols, etc.). The logical integration subdirectory contains information about the semantic heterogeneity of systems (e.g: name mapping, multidatabase views, description of MDI features, compensation of actions, etc.).

4. A Multidatabase Interface (MDI_i) is a server process that accepts queries from the interoperability component, translates these queries into the local query language, submits them to local $system_i$; and returns the results, of the query to the interoperability component.

5. A component system is a local system available for integration. Component systems can be databases, Unix tools, expert databases or any other software systems.

5.2 Communication Primitives

Transactions specified in PLTP interact with the component systems through distinct primitives supported by the system. One such primitive is called remote_call. As it has been previously introduced, remore_call has three possible arguments. An input argument called *stdin(In_List)*, and one of two output arguments *stdout(Out_List)* and *stderr(Err_List)*. The query represented by *In_List* is passed to the MDI which forwards it to its local system. The local system returns either a Out_list or Err_List to the interoperability component containing the results of the execution [ROEL90].

Example 5.1 remote calls to local systems
remote_call(DB2, stdin(['/usr/spool/uucp/myfile']), stdout(File_Contents))
*remote_call(DB1, stdin(['SELECT * FROM flight_table',*
'WHERE from=vie AND to=jfk']),stdout(TupleList))

As shown in the above subsection, an MDI forms the interface between the interoperability component and a local system. In order to be able to communicate with a local system, the interoperability component has to establish an MDI server process for this system at the corresponding

site. This is achieved by the primitive

$$mdi\_start(Local\_System, Argumentlist).$$

We assume that the local system either commits after this command and that results are immediately visible, or it aborts the subtransaction. We distinguish two cases:

1. Commands may directly be written in the language of the local system. In this case the MDI only passes the commands to the local system without processing them.
2. Commands may be written in Prolog, in this case: (1) If the local system is able to process Prolog queries (this means that it is a coupled or an integrated Prolog database system), then the MDI may also pass the query as it is (as in the above example). (2) If the local system does not understand Prolog, then the MDI's will translate the Prolog query into the language of the local system.

Depending on the expressiveness and the degree of autonomy of the underlying local system, an MDI may also provide the primitive

$$mdi\_undo(Local\_System, Argumentlist)$$

in order to undo the effects of the subtyransaction on the local system.

6 Conclusion

Flex transactions extend traditional transactions to allow for flexible transaction execution in MDBSs. The semantically-rich transaction model calls for a powerful transaction specification language. In this paper, we have shown that parallel logic programming (PLP), with extended semantics, could be used as a specification language for Flex transactions. In fact, most of the features of Flex transactions can be naturally represented in this language.

Traditionally, the query processing and the transaction management are studied independently in MDBSs. It is our believe that they interact in subtle ways. With proper extensions, the semantic relations of PLP, initially designed for expressing queries can be used for specifying transactions. With the semantic relations, we are able to study query processing and transaction management in the same framework. An initial implementation of the PLP language has been done in the InterBase project. Currently, we are studying a logic framework which can be used to describe and to analyze different transaction models.

References

[Ehu87] Shapiro Ehud. *Concurrent Prolog.* The MIT Press, 1987. Collected Papers, Vol. 1 & 2.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. E. Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of 16th VLDB conference*, August 1990.

[ER90] A. K. Elmagarmid and M. Rusinkiewicz. Critical issues in multidatabase systems. *Information Science*, 1990.

[KL88] E. Kuhn and T. Ludwig. VIP-MDBS: A logic multidatabase system. In *International Symposium on Database and Distributed Systems*, 1988.

[LAN+87] W. Litwin, A. Abdellatif, B. Nicolas, Ph. Vigier, and A. Zerounal. MSQL: A multidatabase manipulation language. *Information Science*, June 1987. Special Issues on DBS.

[LEB90] Y. Leu, A. Elmagarmid, and N. Boudriga. Specification and execution of transactions for advanced database applications. Technical Report CSD-TR-1030, Purdue University, October 1990.

[ROEL90] M. Rusinkiewicz, S. Ostermann, A. Elmagarmid, and K. Loa. The distributed operational language for specifying multi-system applications. In *Proceedings of the 1st International Conference on Systems Integration*, 1990.

[Sar88] V. J. Saraswat. A somewhat logical formulation of CLP synchronization primitives. In *Proceedings of the 5th International conference and symposium of logic programming.* The MIT Press, 1988.