The InterBase Parallel Language: Supporting the Flex Transaction Model and Beyond

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

Jiansan Chen

Report Number:
92-017
THE INTERBASE PARALLEL LANGUAGE: SUPPORTING
THE FLEX TRANSACTION MODEL AND BEYOND

Ahmed K. Elmagarmid
Jiansan Chen

CSD-TR-92-017
March 1992
The InterBase Parallel Language:
Supporting the Flex Transaction Model and Beyond

Ahmed K. Elmagarmid and Jiansan Chen
Computer Sciences Department
Purdue University
West Lafayette, IN 47907
{ake, jchen}@cs.purdue.edu

Abstract

An important component of the InterBase project at Purdue University, the InterBase Parallel Language (IPL) is a declarative distributed language originally designed to support a flex transaction model [ELLR90]. IPL allows users to write global transactions by specifying all associated actions (grouped by subtransactions), and their sequence, as well as logical dependencies and data flows among subtransactions, without violating local autonomy. It also allows subtransactions of global transactions to be executed in parallel whenever possible. In this paper, we present the IPL language with examples to illustrate its support of flexible transactions, mixed transactions, and time-constrained transactions [ELLR90]. We also describe several additional features of IPL which are not supported by the Flex Transaction Model. Besides its transaction-oriented features, IPL can be used as a general purpose distributed programming language.

1 Introduction

The integration of computation and data is a complex process. A heterogeneous system integrates pre-existing systems to support global applications accessing more than one element system. For many applications, a heterogeneous system is an attractive alternative to a single system because it supports global applications accessing multiple systems simultaneously, thus enhancing performance. Unlike traditional homogeneous distributed systems, it interconnects element systems in a bottom up fashion, thereby allowing existing applications developed on the element systems to continue to be executable without modification. A multidatabase system (MDBS) is such a heterogeneous system. It integrates pre-existing local database systems (LDBSs) to support applications over underlying LDBSs. Important features of LDBSs are the atomicity, consistency, isolation, and durability (ACIDity) of their transactions. Although the ACIDity are the basic properties of LDBSs, their feasibility have come under question in advanced applications on MDBSs. It has recently been suggested, however, that a selective waiving of these properties in favor of increased
flexibility or enhanced performance should be made on a case-by-case application basis. This proposal has led to the development of several advanced transaction models [Eea92].

The InterBase project in the Department of Computer Sciences at Purdue University is devoted to the investigation of heterogeneous systems. In [ELLR90], a new advanced transaction model (called the Flex Transaction Model) was proposed to cope with the problems arising from transaction processing in heterogeneous systems. The concept of Flex Transactions is a generalization of traditional transactions and can be defined as follows [LEB92]:

**Definition 1.1** A flex transaction $T$ is a 5-tuple $(B, S, F, \Pi, f)$ where

- $B = \{t_1, t_2, ..., t_n\}$ is a set of typed subtransactions called the domain of $T$. The type of a subtransaction indicates whether or not it is compensatable, thereby allowing compensatable and non-compensatable subtransactions to coexist within a single global transaction. For example, a subtransaction reserving a seat on a flight can be compensated by another subtransaction which cancels the reservation. This property allows subtransactions to commit before the corresponding global transaction has committed.

- $S$ is a partial order on $B$ called the success order of $T$. It defines positive dependencies between subtransactions. A subtransaction $t_i$ is positively dependent on subtransaction $t_j$ if $t_i$ can be executed only after $t_j$ is successfully executed.

- $F$ is a partial order on $B$ called the failure order of $T$. It defines negative dependencies between subtransactions. A subtransaction $t_i$ is negatively dependent on subtransaction $t_j$ if $t_i$ can be executed only after $t_j$ is executed and failed.

- $\Pi$ is a set of external predicates on $B$ for other conditions in the scheduling of subtransactions. It allows greater flexibility in subtransaction processing. For example, the reservation of a seat for a flight must be made before the trip.

- $f$ is an $n$-ary boolean function defined on the set $\{1, 0\}$ and is called the acceptability function of $T$. It provides function replication allowing the composition of flexible transactions which can tolerate the failure of individual subtransactions by exploiting the fact that a given function can frequently be accomplished by more than one software system. For example, the transaction programmer may leave to the system the choice of renting a car from Hertz or Avis.

DOL [ROEL90], the first version of the Interbase Language, serves as a task specification and execution language for multidatabase activities. Users of DOL are able to define a global transaction consisting of several subtransactions on different local systems and to specify the message passing among them. However, DOL is inadequate to support the Flex Transaction Model, as it is not able to specify flexible, mixed, or time-constrained transactions [ELLR90]. These limitations led to the design of IPL, which maps the Flex Transaction Model into a programming language.

Investigations in the InterBase lab at Purdue University have succeeded in representing the Flex Transaction Model through the medium of the InterBase Parallel Language (IPL), which has
been designed as a transaction language for the MDBS environment. The major advantage of using such a language to represent flex transaction specifications is its semantic power and suitability. The IPL language permits the simultaneous execution of parallel transactions directed toward their goals, thus achieving a higher degree of parallelism in the MDBS environment. Such properties of the Flex Transaction Model as compensability, function replication, and function dependency are readily performed in IPL. Through dependency description, programmers in IPL can specify dependencies among subtransactions and thus maximize the flexibility of a global transaction. IPL permits the construction of a mixed global transaction by allowing the extent of compensatability within the flex transaction to be specified in the declarations of subtransactions. Commit and abort operations of subtransactions are deferred until their global transactions commit or abort, following a Semantic-based Commitment Protocol. The time constraint and guard options of IPL provide programmers with the capability to specify the preferred starting and ending times, as well as other execution conditions of a given subtransaction. IPL also provides an environment in which transaction management and query processing are integrated. Users can therefore define queries which may include complex subtransactions.

IPL can be considered a general purpose distributed programming language, because the text of transaction operations is not passed through the IPL interpreter. In this respect, IPL is similar to DOL [ROEL90]. Another language based on the Flex Transaction Model is Vienna Parallel Logic (VPL) [KPE91]. VPL is a concurrent Prolog language, in which the dependencies of subtransactions are specified by Prolog clauses. Backtracking, unification, cut operator, and other features of Prolog are extended to permit the specification of flexible and mixed transactions. Aside from superficial similarities of appearance, IPL and VPL differ in their methods of defining subtransactions and specifying message transfer between subtransactions and in their support of other transaction features.

An early version of IPL was introduced in [BCC+92]. That paper, however, did not include several new features that transcend the Flex Transaction Model, including the language construct supporting a Semantic-based Commitment Protocol and the Extended Boolean Type. While the concept of implicit dependencies among subtransactions was mentioned, its full significance was not explained. The present paper will provide a thorough discussion of these aspects found in the new version of the InterBase System.

The rest of the paper is organized as follows. A description of the components of IPL is provided in Section 2, while advanced features of the language are discussed in Section 3. Section 4 offers an example of IPL applications. Section 5 the outlines implementation of IPL in the new version of the InterBase System, and concluding remarks and an agenda for future work appear in Section 6. The syntax and semantics of IPL appear in the Appendix A and Appendix B, respectively.

2 IPL: Language Components

IPL contains four fundamental components: objects and types, subtransaction definitions, dependency descriptions among subtransactions, and acceptable sets. These components not only
support the Flex Transaction Model, but also provide extended capabilities.

2.1 Objects and Types

Objects in IPL serve as results of and arguments to subtransactions in an IPL program. Therefore, in IPL, each subtransaction is associated with a type. Types have unique names and are used to categorize objects into sets that are capable of participating in a specific set of subtransactions.

A type can be a basic type, i.e., an integer, a real, a charString, or a bitString; an aggregate type of homogeneous values, e.g., an array of integers; or an aggregate type of heterogeneous values, e.g., a class of mixed integer(s), real(s), and string(s). An element of a type can be another previously defined type. Types are organized in an acyclic type graph.

A type specifies the kind of result a successful subtransaction will produce. Because the result of a subtransaction is an object, it can be easily transferred to another context. For example, if desired, an IPL interpreter can reveal the result of a subtransaction to users, or it can be employed as an input parameter of another subtransaction or to evaluate the guard of another subtransaction. This flexibility is an advantageous feature of IPL which is not offered by the Flex Transaction Model.

For example, to define the type of an airline ticket, an IPL format could be:

```ipl
class ticket of
  number : integer;
  passenger_name: charString;
  origin: charString;
  destination: charString;
  departure_time: charString;
  arrival_time: charString;
  cost: real;
endclass;
```

ticket is used as the name of the type. class, of and endclass are used as keywords in IPL.

The above example illustrates the difference between the type of a subtransaction and of its output, while the former indicates whether it is compensable, the latter defines the type of the output of the subtransaction as an object.

2.2 Definition of Subtransactions

In IPL, a subtransaction is a task executable on a local software system which is carried on a computer system reachable via a computer network. The subtransaction may require the results of other subtransactions as its input. It may also be executed under particular time constraints or other conditions. A subtransaction is provided with an identifying name which should be unique within the context of a global transaction. The nature of a subtransaction may be specified by the following parameters:

1. the option of input parameters, each specified by the name of a subtransaction other than itself. As mentioned previously, the result of a subtransaction may become an input of other
subtransactions.

2. the type of its result. As discussed in section 2.1, a subtransaction yields a meaningful result.

3. the name of the software system on which the subtransaction must be executed.

4. the network name of the machine to which the subtransaction must be sent for execution.

5. time options, including starting time, ending time, valid time period of execution, and maximum execution time. This feature supports temporal predicates [ELLR90] and timeout constraints.

6. the option of a guard for other execution conditions, as defined by the user.

7. the body of the operations of the subtransaction.

8. the body of a commit operation. This option is executed when the subtransaction is instructed to commit by its governing global transaction.

9. the body of an undo operation, an option which is executed when the subtransaction is aborted.

The external predicates on a subtransaction can be expressed by parameters 5 and 6; that is, time constraints and guards. In Sections 3.1 and 4, we will show how the interaction of parameters 7, 8, and 9 support typed subtransactions.

For example, one can define the nature of a subtransaction for ordering an airline ticket as follows:

```
subtrans order_ticket (user_info) : ticket at order.aa.com use Sybase between 8:0:0 and 17:0:0 lasts 50 guard
  user_info.status is not suspectOrTerrorist;
beginexec
  the text of operations for reserving a ticket.
endexec
beginconfirm /* operations when it commits */
  the text of operations for confirming the reservation.
endconfirm
beginundo /* operations when it undoes */
  the text of operations for canceling the reservation.
endundo
endsubtrans;
```

In this example, the output type of subtransaction `order_ticket` is `ticket`, which indicates that an object `ticket` is returned as its result if the subtransaction succeeds. The local software system involved is the Sybase database management system, which runs on the machine named `order.aa.com`. The subtransaction is allowed to run between the hours of 8:00 and 17:00 with a maximum execution time of 50 seconds. If its execution time exceeds 50 seconds or 17:00, it will be aborted.
as if its execution had failed. The result of another subtransaction \textit{user\_info} becomes the input of this subtransaction, providing such user-related information as the name of the user and the origin and destination cities. The guard of the subtransaction prevents a suspected terrorist from booking a ticket; in such an instance, the execution of the subtransaction will be rejected as if it had failed. The body of the subtransaction operations, the body of the commit operation, and the body of the undo operation are defined by the IPL keywords \texttt{beginexec} and \texttt{endexec} (the construct \texttt{< executing >}), \texttt{beginconfirm} and \texttt{endconfirm} (the construct \texttt{< confirm >}), and \texttt{beginundo} and \texttt{endundo} (the construct \texttt{< undo >}), respectively.

2.3 Dependency Description

The dependency description, the third component of IPL, provides users with a mechanism for specifying the explicit dependencies among the subtransactions of a global transaction. That is, the execution order of subtransactions of a global transaction can be defined using the IPL dependency description. For example, given seven subtransactions $C_1, C_2, C_3, C_4, C_5, C_6$ and $C_7$, their execution order is defined:

1. $C_2$ will be executed only if $C_1$ succeeds.
2. $C_3$ will be executed only if $C_1$ fails.
3. $C_4$ will be executed only if $C_2$ or $C_3$ succeeds.
4. $C_5$ will be executed only if $C_2$ or $C_3$ succeeds.
5. $C_6$ will be executed only if $C_3$ and $C_4$ succeed or $C_5$ fails.
6. the global transaction will succeed if at least two of $C_4, C_5,$ and $C_6$ succeed.

The IPL dependency description for those subtransactions could be:

\begin{verbatim}
dependency
c1: c2;
not c1: c3;
c2 or c3: c4;
c2 or c3: c5;
c3 and c4 or not c5: c6;
(2: c4, c5, c6): accept;
enddep
\end{verbatim}

Each IPL program must have a dependency description beginning with the keyword \texttt{dependency} and ending with \texttt{enddep}. A dependency description includes one or more dependency pairs, each of which consists of a boolean expression or a partial success expression, a colon, and a subtransaction identifier. This subtransaction, which is dependent upon the boolean expression or the partial success expression, is eligible to be submitted for execution only after the expression upon which it depends becomes \texttt{true}. If its governing expression becomes \texttt{false}, its execution is rejected as if it had failed. If a subtransaction does not depend on any other subtransaction, it can
be submitted for execution immediately. In this example, c1 is such a subtransaction; therefore, it is executed first. If it succeeds, c2 is executed; otherwise, c3 is executed. If either c2 or c3 succeeds, c4 and c5 are executed simultaneously. If c3 and c4 succeed or c5 fails, c6 is executed. If two of c4, c5, and c6 succeed, the global transaction succeeds; otherwise, the global transaction fails. accept, a keyword of IPL, indicates the succeed/fail status of the global transaction (GT). If its value is true, then GT succeeds; if false, then GT fails; otherwise, GT keeps running until accept becomes true or false.

It can be readily demonstrated that the IPL dependency description supports success orders and failure orders of flex transactions. Furthermore, the IPL dependency description can describe more complicated explicit dependencies among subtransactions in a global transaction than can the Flex Transaction Model. This feature therefore represents another extension of IPL to the Flex Transaction Model.

2.4 Acceptable Sets

Acceptable sets, the fourth component of IPL, begin with the keyword acceptable_sets and end with the keyword endaccs. An acceptable set consists of a subtransaction list and a sufficient acceptable condition of the global transaction. When a global transaction reaches its final status, the user is asked to select a preferred acceptable set from an array of choices. All the subtransactions in an acceptable set in the array must be successful. Successful non-compensable subtransactions are maintained in an uncommitted state until the global transaction is completed. When the user chooses an acceptable set and the global transaction commits, the uncommitted subtransactions in the acceptable set then perform their commit operations, all other uncommitted subtransactions perform their abort operations, and the compensatable subtransactions not in the acceptable set perform their compensating operations. When the global transaction decides to abort, all the successful subtransactions perform their abort operations or compensating operations.

Acceptable sets reflect the acceptable functions of flex transactions and support function replication within them, enabling them to tolerate the failure of individual subtransactions by exploiting the ability of several software systems to accomplish a given function. For the example presented in section 2.3, the acceptable sets could be:

```
acceptable_sets
  (c1, c2, c4, c6), (c1, c2, c4, c6), (c3, c4, c5, c6),
  (c3, c4, c6), (c3, c4, c6)
endaccs
```

In this example, five acceptable sets are included, they are subtransaction sets (c1, c2, c4, c6), (c1, c2, c4, c6), (c3, c4, c5, c6), (c3, c4, c5), and (c3, c4, c6). The success of any of these five subtransaction sets will result in the success of the global transaction, thus providing function replication within the global transaction.

Although there are five acceptable sets of subtransactions in this example, there is exclusive relation between subtransactions c1 and c3 [BCC+92], indicating that, at a given time, at most
one of them is true. There are therefore at most three acceptable sets of subtransactions that can be listed for the user to choose among. Only those acceptable sets which include no false subtransactions can be listed for selection. The user must choose one of those acceptable sets as the final result. All the subtransactions in the chosen set can and must be committed, and all other subtransactions must be aborted.

3 Advanced Features of IPL

Several other significant features of IPL are not supported by the Flex Transaction Model. They will be discussed in this section.

3.1 Partial Subtransactions and a Semantic-based Commitment Protocol

As a preliminary step to an examination of partial subtransactions and the semantic-based commitment protocol, let us first define the fundamental concept of reserving states.

Definition 3.1 (Reserving states): A reserving state is a state from which a subtransaction can easily be undone without the creation of any inconsistencies.

For example, in an airline reservation system, one can either reserve or order a ticket. The reserving process requires that one must later confirm one’s reservation (i.e., order the ticket within a specific time period after the reservation), but there is no penalty for cancellation. In contrast, cancellation of an ordered ticket is either impossible or carries a penalty. “A ticket is reserved” is therefore an example of a reserving state for an ordering air-ticket subtransaction.

From its definition, we can conclude that the state of a subtransaction just before its execution, the state of a reversible subtransaction just after its execution, and any state of a read-only subtransaction can be a reserving state. Therefore, a subtransaction may have several reserving states. However, only one among them will not lead, upon the execution of the subtransaction, to any other reserving state. This gives rise to the following definition:

Definition 3.2 (The best reserving state): The best reserving state of a subtransaction is a reserving state that will not lead, upon the execution of the subtransaction, to any other reserving state.

For example, for an ordering air-ticket subtransaction, “a ticket is reserved” is the best reserving state for that subtransaction. It is clear that any subtransaction has only one best reserving state.

The best reserving state forms an axis which is critical to the following definition of three partial subtransactions.

Definition 3.3 For a given subtransaction, the executing partial subtransaction (EPS) is defined to include all operations from the beginning to the best reserving state, the confirming
partial subtransaction (CPS) to include those from the best reserving state to the commitment of the subtransaction, and the undoing partial subtransaction (UPS) to include those from the best reserving state to the abortion of the subtransaction.

For example, for an ordering air-ticket subtransaction, "reserve a ticket" is its EPS, "order the reserved ticket" its CPS, and "cancel the reserved ticket" its UPS.

In IPL, an <executing> construct indicates an EPS, a <confirm> construct a CPS, and an <undo> construct a UPS.

The best reserving state arises from the semantics, rather than from the syntax, of a subtransaction. It is therefore the user’s responsibility to define the best reserving state for a subtransaction, and, by extension, the pertinent EPS, CPS, and UPS. From these definitions, IPL can determine which kind of commitment protocol must be used for the subtransaction.

- For a compensatable subtransaction ($S_i$) [GM83], [KLS90], the best reserving state is, by definition, the state after the execution of $S_i$. Therefore, for $S_i$, the EPS is $S_i$ itself, and the UPS could be the compensating subtransaction for $S_i$. No CPS needs to be defined for $S_i$.

- For a read-only subtransaction ($S_i$), or if the data modified by $S_i$ need not be consistent, then only the EPS for $S_i$ must be defined. Clearly, in this case, the EPS is $S_i$ itself.

- The situation is more complex for a non-compensatable subtransaction ($S_j$) of a global transaction ($T_i$). After the EPS of $S_j$ reaches the best reserving state, two choices are presented to the user in defining the next step of the EPS and the ensuing executions of the CPS and UPS. The resolution of this choice generates different definitions for the EPS, CPS, and UPS in the two instances.

  - If $S_j$ must be run in isolation, the EPS commits at the best reserving state of $S_j$. During the final stage of the execution of $T_i$, the user chooses whether to commit or abort $S_j$. In the former case, the CPS is issued to bring $S_j$ to completion and to commit it; in the latter, the UPS is issued to undo the execution of the EPS.

  - If $T_i$ and $S_j$ are capable of intercommunication and need not be isolated completely, then at the best reserving state of $S_j$, after reporting the success information to $T_i$, the EPS waits for a commit/abort signal from the communication channel set between the EPS and $T_i$. Again, during the final stage of the execution of $T_i$, the user chooses whether to commit or abort $S_j$. In the former case, the CPS is issued to send the "commit" signal to the EPS, thus triggering the EPS to commit; in the latter, the UPS is issued to send the "abort" signal, triggering the EPS to abort. This is very similar to the Two Phase Commitment Protocol.

Regardless of the number of partial subtransactions defined for a subtransaction ($S_i$), IPL always execute its EPS first. At the final stage of the global transaction, if $S_i$ must be committed and its CPS is defined, the CPS is submitted for execution to commit $S_i$; on the other hand, if $S_i$ must be aborted and its UPS is defined, the UPS is submitted for execution to undo $S_i$.
For each subtransaction, the user defines the best reserving state, as well as the EPS, CPS, and UPS. Although the semantic structure of IPL delegates to users any necessary decisions regarding commitment, IPL can determine from the definition of a subtransaction what kind of commitment operation should be used.

In this approach, the commitment protocol for a subtransaction is based on the semantics of the subtransaction and is grounded on the commitment protocols of the component systems. Such a commitment approach is both flexible and simple and can be applied to a variety of application environments. We will present an application example in Section 4 to illustrate the effectiveness of the Semantic-based Commitment Protocol.

3.2 The Extended Boolean Type

Using the boolean expression to express the dependency relations among subtransactions in an IPL dependency description has the immediate advantage of simplicity. However, as the evaluation of a boolean expression returns either true or false, it carries the potential for later confusion. A truth value indicates that the dependent condition for the execution of a subtransaction is satisfied, and that, therefore, the subtransaction is eligible for execution (unless it is in violation of its temporal constraints and guard condition). A false value indicates that the dependent condition for the execution of a subtransaction is not satisfied; therefore, the subtransaction will never be eligible for execution, and it should be looked upon as a failed subtransaction. In many cases, however, the dependent condition for the execution of a subtransaction is undetermined. In the example in section 2.3, while subtransaction C1 is executing, its eventual success or failure is unknown; therefore, the dependent expressions for subtransactions C2 and C3 are undetermined at that time. A state that is neither true nor false is needed to describe the current state of subtransactions C2 and C3. The conventional boolean type cannot handle cases of this kind, because the evaluation of a boolean expression returns either true or false.

To overcome this inadequacy on the part of the conventional boolean type, we here introduce the concept of the extended boolean type. The following is an intuitive definition of the extended boolean type:

**Definition 3.4 (The extended boolean type):** An expression is said to be an extended boolean expression if:

1. syntactically, it follows the format of conventional boolean type,

2. each variable in the expression, referred to as an extended boolean variable, can be in value either true, false, or undefined; and

3. its computational laws follows those of the conventional boolean type, with the following additional elementary laws:
   - true or undefined = true ;
• false or undefined = undefined;
• undefined or undefined = undefined;
• true and undefined = undefined;
• false and undefined = false;
• undefined and undefined = undefined;
• not undefined = undefined.

The extended boolean expression is evaluated true or false if its value can be determined; otherwise, its value is undefined. An undefined extended boolean expression in IPL will be evaluated later until its value becomes true or false. From this definition, it can be seen that the extended boolean type is a semantic extension of the boolean type. The extended boolean type can be readily understood and implemented and meets requirements of the Flex Transaction Model.

3.3 Implicit Dependencies

Definition 3.5 (Implicit dependency among subtransactions): If a subtransaction $S_i$ takes the result of another subtransaction $S_j$ as its input argument, then we say there is an implicit dependency between $S_i$ and $S_j$, or $S_i$ depends implicitly on $S_j$. A subtransaction can depend implicitly on several other subtransactions. Subtransactions can also depend implicitly on each other.

This definition of the implicit dependency is in contrast to the explicit dependency set forth in the dependency description. While subtransactions with explicit dependencies must be executed in a serial mode, one after another, subtransactions with implicit dependencies may be executed in pipe mode, in parallel. That two subtransactions run in pipe mode indicates that they may both begin their execution, even if their input parameters are not yet available. If absent input parameter(s) are required during the subsequent execution, processing will be suspended until they become available. It follows from these fundamentals that the implicit dependency permits a higher degree of concurrency than the explicit dependency.

In addition to the commitment protocol outlined in Section 3.1, the implicit dependencies between subtransactions may also be used to control the execution of subtransactions. If two subtransactions $S_i$ and $S_j$ are executed in pipe mode and the output of $S_i$ becomes an input parameter of $S_j$, then by generating different data as its output, $S_i$ can control the execution of $S_j$. Furthermore, $S_i$ can use its output to reveal its execution status to $S_j$. For a subtransaction ($S_i$) of a global transaction ($T_j$), two specially designed subtransactions ($S_s$, $S_t$) can be put in place to control the execution and indicate the status of $S_i$. The output of $S_s$ is used as an input parameter of $S_t$, while the output of $S_t$ becomes the input parameter of $S_i$. Since $S_s$ and $S_t$ are specially designed, $T_j$ can control their executions and directly access their data. When these three subtransactions are executed in pipe mode, $S_s$ allows $T_j$ to control the execution of $S_i$, while $S_t$ reveals the status of $S_i$ to $T_j$. More specifically, when $S_i$ reaches its prepare-to-commit status, it sends the message ready-to-commit as its output to $S_t$ and then waits for input data from $S_s$. $S_t$
is specially designed to allow \( T_j \) to access this message. After collecting such messages from all its subtransactions or running out of time, \( T_j \) then asks \( S_i \) to send a commit or abort message to \( S_i \), triggering \( S_i \) to commit or abort. A variety of commitment protocols can be implemented in this manner.

The concept of implicit dependencies also allows subtransactions that depend on each other to be components of the same IPL program, a situation not realizable in the Flex Transaction Model. For example, in an ordering airline-ticket application, a user may wish to order a ticket on United from Chicago to New York and to pay with his VISA card. Such an application involves two tasks, one on the United Airlines reservation system and the other on the VISA payment system. However, since the payment decision is based on ticket availability and price, while the United task involves an evaluation of the user’s credit line, the two tasks depend on each other. While the Flex Transaction Model does not support such applications, the capability of IPL to define the implicit dependency between two subtransactions does make their execution feasible. In Section 4, we will provide an example of such an application.

4 An Application Example

In this section, we will provide an example that illustrates the capability of IPL to support applications that conform to the Flex Transaction Model, as well as those that do not.

Consider a professor from Purdue University who wishes to attend a conference to be held at the Sheraton Hotel in San Francisco from Feb. 4 to Feb. 6, 1992. A global transaction for his trip may consist of the following objectives:

- He should provide information such as his name, the origin and destination of the trip, the departure and return times, and the type and number of the credit cards with which he wishes to pay for his airline ticket;
- Airlines companies should be contacted to book a flight;
- The Sheraton Hotel should be contacted for a room reservation; and
- The credit card companies should be contacted for the payment of the airline ticket.

Let us assume that, for the purpose of this trip, three airline companies (USAir, United, and American) and one credit card company (VISA) can be involved.

Suppose that the professor has the following preferences:

1. Order a ticket from United Airlines or American Airlines only if no ticket is available from USAir, because if he stays at the Sheraton when traveling by USAir, he can triple his frequent-flyer mileage.
2. Reserve a room at the Sheraton only if an airline ticket is available. If no airline ticket is available, he will not attend the conference.
3. Reserve a room at the Sheraton before 3:00 p.m. on Feb. 3, 1992. If a room is not reserved for him before his departure, he will also decide not to make the trip.
4. The cost of the airline ticket should be $350, this being his maximum budget.

The four objectives can be decomposed as four sets of subtransactions \{user\}, \{usair, united, american\}, \{sheraton\}, and \{visa\}, where they are defined as follows:

\begin{itemize}
  \item **user**: Obtain the information from the customer;
  \item **usair**: Order a ticket from USAir;
  \item **united**: Order a ticket from United Airlines;
  \item **american**: Order a ticket from American Airlines;
  \item **sheraton**: Reserve a room at the Sheraton;
  \item **visa**: Pay for the airline ticket with VISA;
\end{itemize}

As indicated by the client’s preferences, there are explicit dependencies among subtransactions usair, united, american, sheraton, and visa, which can be defined as follows: subtransaction united and american will be executed only if the execution of the subtransaction usair fails, and subtransaction sheraton will be executed only if one of the subtransactions usair, united, and american succeeds and the subtransaction visa succeeds.

There are also implicit dependencies between the set \{usair, united, american\} and the set \{visa\}, because if no airline ticket is available, it is unnecessary to pay for it, and if there is insufficient credit remaining on a credit card, no ticket will be sold. Therefore, the two sets of subtransactions depend upon each other, and implicit dependencies may be used to define this relationship. Because all other subtransactions need the information acquired by subtransaction user, there are also implicit dependencies between user and the other subtransactions.

It is clear that the global transaction will succeed if one of usair, united, and american succeeds, sheraton succeeds, and visa succeeds.

An IPL program for the global transaction could read as follows:

```
program
  class creditCard of
    cardholder : charString;
    num : charString;
    creditRemains : real;
    reserved.credit : real;
  endclass;
  class user-info of
    name : charString;
    origin : charString;
    destination : charString;
    departure_time : charString;
    return_time : charString;
    visa : creditCard;
  endclass;
  class ticket of
    flightNo : charString;
```
customer : user-info;
cost : real;
paid-by : creditCard;
endclass;
class room of
customer : charString;
roomNo : charString;
cost : real;
time : charString;
endclass;
subtrans user : user-info use user_interface at Customer_Service
beginexec
obtain the information from the customer.
endexec
endsubtrans
subtrans usair (user, visa) : ticket use ticket_order at USAir
beginexec
reserve a ticket for user.name.
endexec
beginconfirm
order the reserved ticket, pay with visa.
endconfirm
beginundo
cancel the reserved ticket for user.name.
endundo
endsubtrans
subtrans united (user, visa) : ticket use ticket_order at United_Air
beginexec
reserve a ticket for user.name.
endexec
beginconfirm
order the reserved ticket, pay with visa.
endconfirm
beginundo
cancel the reserved ticket for user.name.
endundo
endsubtrans
subtrans american (user, visa) : ticket use ticket_order at American_Air
beginexec
reserve a ticket for user.name.
endexec
beginconfirm
order the reserved ticket, pay with visa.
endconfirm
beginundo
cancel the reserved ticket for user.name.
endundo
endsubtrans
subtrans sheraton (user) : room use room.reserve at Sheraton_Hotel before Feb 3 15:00 EST 1992
beginexec
  reserve a room for user.name.
endexec
beginundo
  cancel the reserved room for user.name.
endundo
endsubtrans
subtrans visa (user, usair, united, american) : creditCard use credit.process at VISA_Card
  guard
    user.visa.num is valid and
    ((usair.cost ≤ $350 and user.visa.creditRemains > usair.cost) or
    (united.cost ≤ $350 and user.visa.creditRemains > united.cost) or
    (american.cost ≤ $350 and user.visa.creditRemains > american.cost)) ;
  beginexec
    reserve the credit for the ticket for user.name
  endexec
beginconfirm
  pay for usair if usair is chosen for commit.
  otherwise, pay for united if united is chosen for commit.
  otherwise, pay for american
endconfirm
beginundo
  cancel the reserved credit for the ticket for user.name
endundo
dependency
  not usair : united ;
  not usair : american ;
  (usair or united or american) and visa : sheraton ;
  (usair or united or american) and sheraton and visa : accept ;
enddep
acceptable_sets
  (usair, sheraton, visa), (united, sheraton, visa), (american, sheraton, visa)
endacess
endprogram

We will now use the foregoing example to elucidate some of the principles set forth earlier. The example illustrates how the preferences of a user are represented in an IPL program, how the time constraint and guard functions operate, and how explicit and implicit dependency relationships among subtransactions of a global transaction are presented.

Explicit dependencies are relationships among subtransactions of a global transaction which determine their correct execution order. These dependencies must therefore be taken into consideration by the Flex Transaction Model. In IPL, the explicit dependency relationships among
subtransactions are defined in the dependency description construct. A subtransaction is eligible to be scheduled only if its dependency becomes true. In our example, preference 1 can be seen as an explicit dependency for united and american and preference 2 that for sheraton.

Mixed transactions involving a variety of subtransaction classes can be implemented through the careful definition of an executing partial subtransaction, confirming partial subtransaction, and undoing partial subtransaction for each subtransaction, as illustrated in Section 3.1. In our example, subtransaction sheraton is a compensatable subtransaction, while usair, united, american, and visa are non-compensatable subtransactions.

The time constraint for each subtransaction is defined in a < time_expr > construct. In our example, preference 3 is a time-constraint for subtransaction sheraton.

A guard imposes another condition for the execution of a subtransaction. In our example, the guard for visa ensures that the ticket can be paid for by a credit card provided by the customer. Just as the failure of usair, united, or american will abort the global transaction, so will the failure of visa; guards therefore also guarantee the correct execution of the global transaction.

The usefulness of the concept of implicit dependencies among subtransactions of a global transaction is also illustrated here. In this example, there are implicit dependencies between the two groups of subtransactions {usair, united, american} and {visa}. There are also implicit dependencies between the subtransaction user and all other subtransactions.

There are three acceptable sets of subtransactions in this example, from which the user must choose one as the final result. All the subtransactions in the chosen set can and must be committed, and all other subtransactions must be aborted. Again, the commitment in this example is semantic-based, since different commitment protocols are applied to subtransaction sets {user}, {usair, united, american, visa} and {sheraton}. The appropriate protocol can be deduced from the semantics of these subtransactions, as illustrated in Section 3.1.

As the foregoing example was intended to illustrate the control structures of the IPL language, rather than the details of Local Software Systems (LSSs) and Remote System Interfaces (RSIs), which will be discussed in the next Section, we have used only pseudo codes for each action. This example particularly highlights the effectiveness of confirming and undoing partial subtransactions, since most businesses allow customers to make, confirm, and cancel reservations without an extra charge within a period of time which is sufficient for the execution of a global transaction.

5 The Implementation of IPL in the InterBase System

IPL is currently implemented as the transaction language of the InterBase System at Purdue University. In this section, we will first briefly describe the InterBase System and then discuss the IPL interpreter, the key component of the Distributed Flexible Transaction Manager.
5.1 An Overview of the InterBase System

Prior to detailing the implementation of IPL, it is appropriate to ground the discussion upon a description of the architecture of the InterBase System. We will describe the various components and modules of the system and briefly elucidate their mutual interactions. The InterBase System is designed to allow users to write global applications over a distributed, autonomous, and heterogeneous computing environment (in particular, a multidatabase environment), while retaining the autonomy of Local Software Systems (LSSs).

The major components and modules of the InterBase System and the relationships among them are presented in Figure 1. Arrowed lines indicate the flow of commands and data among the modules of different systems, while unarrowed lines represent such flow between Remote System Interfaces (RSIs) and LSSs. The unarrowed double line indicates that the subtransaction schedulers are a component of the RSIs. At present, the InterBase System runs on an interconnected network with a variety of hosts that include Sun, HP and NeXT workstations, Sequent machines, IBM mainframes, and IBM/PCs.

The architecture of the InterBase System is designed in such a way that all its major modules, except the LSSs, interface with the Distributed Flexible Transaction Manager (DFTM). The DFTM is the central component of the InterBase System, interpreting and coordinating the execution of global transactions over the entire system. RSIs ensure a uniform interface to the DFTM and deal with the heterogeneity of the LSSs. A user can invoke User Interfaces such as a graphical interface, to make a query to the InterBase System; the User Interface will translate the query into an IPL text, and the text will then be sent to the DFTM for execution. A user with a good grasp of LSSs and a fluency in IPL can also write and send IPL texts directly to the DFTM for execution. A consistent and reliable execution of an IPL text via the DFTM is seen as a global transaction over the InterBase System. Assisting in this process is the Distributed Concurrency Controller (DCC), consisting of a Group Manager and Subtransaction Schedulers, each of the latter of which is an important component of an RSI. The DCC is so named because it is based on the distributed algorithms discussed in [ECD+92]. The DCC is used to manage the parallel access of global transactions over the InterBase System. Because the DFTM allows several global transactions to be executed simultaneously, the InterBase System can be run in a multi-user environment.

5.2 The IPL Language Interpreter

An important part of the DFTM, the IPL language interpreter interprets the execution of IPL programs. For each execution of an IPL program, the DFTM generates an image of itself which is responsible for the consistent and reliable execution of the program. The existence of a DFTM image is therefore coincident with the execution of a global transaction in the InterBase System; after the execution of the global transaction, the DFTM image disappears. By providing a DFTM image for each global transaction, the DFTM allows several global transactions to run concurrently. The DFTM images must communicate with each other either directly or indirectly to coordinate their executions.
Figure 1: The Architecture of the InterBase System
During the execution of a global transaction of IPL format, the IPL interpreter is responsible for (1) checking the syntax and semantics of the transaction; (2) managing the flow of control specified by the transaction; (3) activating and opening connections to RSIs; (4) monitoring the status of the individual RSIs; (5) obtaining the DCC permission to execute subtransactions; (6) executing subtransactions (via corresponding RSIs), whenever possible; (7) determining the final status (accepted or unaccepted) of the transaction; and (8) committing or aborting its subtransactions according to their accepted or unaccepted status.

The DFTM image \((DFTM_i)\), therefore, governs the entire life cycle of a global transaction, from inception to completion. After syntax and semantic checks of a global transaction \((T_i)\), a simple execution graph, which reflects the dependency relations among all its subtransactions, is built for \(T_i\). The \(DFTM_i\) for \(T_i\) then obtains a group id for \(T_i\) and grouping triples for its subtransactions from the Group Manager of the DCC. The group id and the grouping triples [ECD+92] are used by the Subtransaction Schedulers of the DCC in the individual RSIs to guarantee that subtransactions of \(T_i\) are executed in quasi serialization order on each LSS. The \(DFTM_i\) then asks the relevant RSIs to approve the executions of subtransactions whose dependency conditions, time constraint, and guard are all satisfied. Upon receiving an approval, the \(DFTM_i\) immediately executes the approved subtransaction on the corresponding LSS, via its RSI, until the best reserving state specified in the IPL program is reached or the execution has failed. The \(DFTM_i\) then modifies the execution graph. This process continues until \(T_i\) reaches its final status. At that point, the \(DFTM_i\) commits those subtransactions selected by the user and aborts those which the user does not want. Throughout, the \(DFTM_i\) consults the RSI Directory to determine the interface and data transfer characteristics of the individual RSIs. Completed executions are reported by the \(DFTM_i\)s to the Group Manager of the DCC. The DCC thus tracks executing and completed global transactions.

6 Conclusions and Ongoing Research

The rapid growth of advanced applications involving multidatabase systems has resulted in the development of various non-traditional transaction models and languages. One of these, the InterBase Parallel Language (IPL), is presented in this paper. IPL, a distributed transaction-oriented language based on the Flex Transaction Model, provides an appropriate environment for the execution of flexible transactions. IPL supports flexible, mixed, and time-constrained transactions and offers additional features that are not supported by the Flex Transaction Model.

IPL includes and extends all the functions of the Distributed Operation Language (DOL) [ROEL90]. IPL also provides an environment in which global transaction management and query processing are closely integrated in order to meet the needs of various autonomous applications. IPL can be considered a general purpose language, because the text of subtransaction operations is not passed through the IPL interpreter. The proposed language requires no knowledge of the local software systems other than their operation languages; therefore, it does not violate local autonomy. In general, IPL offers great power and a particular suitability to the execution of global applications in a heterogeneous environment.
A more user-friendly interface for the IPL language, such as a graphical user interface, has been investigated and is currently being implemented. An object-oriented version of the IPL language is also being investigated. The investigation of the use of IPL for other advanced transaction models is also being undertaken. The results of these projects will be reported in upcoming articles.

Acknowledgements

The authors are very grateful to Weimin Du, a member of the technical staff at the Hewlett-Packard Laboratories, for many helpful comments and proofreading.

References


Appendix A: The Syntax of the IPL Language

The Backus-Naur Form (BNF) syntax of the IPL is as follows:

```
<program> ::= program <type_defs> <subtrans_decls>
               <dependency_decls> <final_status> endprogram

<type_defs> ::= <type_defs> <type_def> | <type_def>

<type_def> ::= class <user_type> of <type_list> endclass;

<user_type> ::= <id>

<type_list> ::= <type_list> <a_type> | <a_type>

<a_type> ::= <var_list> : <basic_type> ;

<type> ::= <basic_type> | <user_type> | <compound_type>

<compound_type> ::= array of <basic_type> | array of <user_type>

<basic_type> ::= int | real | boolean | charString | bitString

<subtrans_decls> ::= <subtrans_decls> <subtrans_decl> | <subtrans_decl>

<subtrans_decl> ::= subtrans <id> [ ( <arg_list> ) ] : <type>
                 use <rsi> at <site> <subtrans_body> endsubtrans

<rsi> ::= <id>

<site> ::= <id>

<subtrans_body> ::= [ <time_constraint> ] [ <guard> ]
                 [ <executing> ] [ <confirm> ] [ <undo> ]

<executing> ::= beginexec <exec.body> endexec

<confirm> ::= beginconfirm <confirm.body> endconfirm

<undo> ::= beginundo <undo.body> endundo

<time_constraint> ::= before <time>
                   | between <time> to <time>
                   | after <time>

<time> ::= "timeofday"

<guard> ::= guard <ext.bool.expr> ;

<dependency_decls> ::= dependency <dependency_list> enddep

<dependency_list> ::= <dependency_list> <dependency_pair>
                   | <dependency_pair>

<dependency_pair> ::= <ext.bool.expr> : <id> ;
                   | <ext.bool.expr> : accept ;
```
< ext_bool_expr > ::= < ext_bool_expr > or < ext_bool_term >
| < ext_bool_term >

< ext_bool_term > ::= < ext_bool_term > and < ext_bool_factor >
| < ext_bool_factor >

< ext_bool_factor > ::= < operand > < compare_op > < operand >
| ( < ext_bool_expr > )
| < partial_succ_ext_bool_expr >
| < id >
| not < id >

< operand > ::= < value > | [ < id > [ ( < index > ) ] ] . ] < id >
< compare_op > ::= > | >= | < | <= | <> | ==

< partial_succ_ext_bool_expr > ::= ( < number > : < subtrans_var_list > )

< subtrans_var_list > ::= < var_list >

< arg_list > ::= < var_list >

< var_list > ::= < var_list > , < id > | < id >

< final_status > ::= acceptable_sets < acceptable_sets > endaccs
< acceptable_sets > ::= < acceptable_sets > , ( < subtrans_list > )
| ( < subtrans_list > )

< subtrans_list > ::= < var_list >

Keywords are in boldface, as program, subtrans. < number > and < index > can be any positive decimal number. < value > can be any real or integer number.

The lower bound of an array is 1; its upper bound will be determined automatically by the IPL program interpreter.

timeofday is in the format mon day hh:mm:ss time-zone year; as
Feb 27 11:31:46 GMT 1991. If an item is absent, then it takes the default value;
11:31:46 will be interpreted as
Feb 27 11:31:46 EST 1991 for a program executed in the eastern time zone.
Appendix B: The Semantics of the IPL Language

We will explicate only those IPL syntactic constructs which are non-intuitive. We assume that the reader already has an understanding of context-free grammars.

- \(<\text{subtrans}\_\text{decl}>::=\text{subtrans}<\text{id}>[<\text{arg}\_\text{list}>]:<\text{type}>\)
  \(\text{use}<\text{rsi}>\text{at}<\text{site}><\text{subtrans}\_\text{body}>\text{endsubtrans}\)

This construct is used to define a subtransaction (e.g., \(S_i\)); its name, its type, its RSI server, and the RSI location are given by the \(<\text{id}>\), \(<\text{type}>\), \(<\text{rsi}>\), and \(<\text{site}>\), respectively. When \(S_i\) is eligible to be scheduled, and both its time constraint and guard are evaluated true, DFTM initiates the process by obtaining permission from DCC. DFTM then opens a connection between the global transaction where \(S_i\) is defined and an RSI service performed by the RSI server. Finally, using the RSI service as an intermediary, DFTM executes \(S_i\) on the appropriate LSS.

\(<\text{arg}\_\text{list}>\), an option, defines the parameter list of \(S_i\). As these parameters are actually the outputs of other subtransactions, the parameter list will consist of the names of these subtransactions, each of which must be unique.

Each \(<\text{id}>\) has a double typed definition. Its explicit type is given by \(<\text{type}>\), while its implicit type is an extended boolean type. That is, its value can be \(\text{true}\), \(\text{false}\), or \(\text{undefined}\), respectively representing the success, failure, or undetermination of the \(<\text{executing}>\) term in the \(<\text{subtrans}\_\text{body}>\) construct. When an \(<\text{id}>\) acts as an \(<\text{ext}\_\text{bool}\_\text{factor}>\) or appears in a \(<\text{subtrans}\_\text{var}\_\text{list}>\) construct, its implicit type is used; otherwise, it is defined by its explicit type. The use of the explicit type allows DFTM to process the output of the subtransaction as structured data rather than an uninterpreted string. Other subtransactions can then incorporate this output as a parameter in their execution. The implicit type of subtransaction, on the other hand, permits dependencies among subtransactions to be implemented as extended boolean expressions.

- \(<\text{subtrans}\_\text{body}>::=[<\text{time}\_\text{constraint}>][<\text{guard}>][<\text{executing}>][<\text{confirm}>][<\text{undo}>]\)

\(<\text{executing}>::=\text{beginexec}<\text{exec}\_\text{body}>\text{endexec}\)
\(<\text{confirm}>::=\text{beginconfirm}<\text{confirm}\_\text{body}>\text{endconfirm}\)
\(<\text{undo}>::=\text{beginundo}<\text{undo}\_\text{body}>\text{endundo}\)

An \(<\text{exec}\_\text{body}>\) construct constitutes a command text of the executing partial-subtransaction for a subtransaction \(S_i\) in which the text is defined. The text will be executed when it is sent by the global transaction \((G_j)\) to its RSI service.
A `<confirm.body>` construct, an option, constitutes a command text of the confirming partial-subtransaction for $S_i$. The text will be executed after $G_j$ decides to commit $S_i$ and sends the text to its RSI service.

An `<undo.body>` construct, also an option, constitutes a command text of the undoing partial-subtransaction for $S_i$. The text will be executed after $G_j$ decides to abort $S_i$ and sends the text to its RSI service.

- `<ext.bool.expr>` ::= `<ext.bool.expr>` or `<ext.bool.term>`
  
  ...<br>

  `<ext.bool.factor>` ::= `<operand>` `<compare.op>` `<operand>`
  
  ...

This is an extended boolean expression definition. Each `<ext.bool.expr>`, `<ext.bool.term>`, or `<ext.bool.factor>` carries a value of `true`, `false`, or `undefined`. If an `<ext.bool.expr>` is evaluated as `undefined`, it will later be evaluated until its value is `true` or `false`.

Both `<operand>` constructs in an `<ext.bool.factor>` should be of the same or compatible types. For example, if one is an integer and the other a real, the integer will be transformed to a real prior to comparison. If the two `<operand>`s are incompatible, then the value of the `<boolean.factor>` is `false`.

- `<time.constraint>` ::= `before` `<time>`
  
  | `between` `<time>` to `<time>`
  
  | `after` `<time>`

  `<guard>` ::= `guard` `<ext.bool.expr>`

  ...

Both `<time.constraint>` and `<guard>` are options. When a subtransaction ($S_i$) is eligible to be scheduled, its time constraint and guard are evaluated. Any absent time constraint or guard is assigned the value `true`. If both options are `true`, then $S_i$ can be executed; if one of them is `false`, then a `false` value is bound to $S_i$, as if its execution had failed. If both are `undefined`, the execution of $S_i$ is delayed for later evaluation.

`<time>` indicates the local time of the site where $S_i$ is executed. DFTM associates a time zone with each entry in the RSI Directory, enabling easy translation between the local and remote times.

For the `before` construct, if the current time is before `<time>`, then `<time.constraint>` is `true`; otherwise, `<time.constraint>` is `false`.

For the `between` construct, if the current time is before the first `<time>`, then `<time.constraint>` is `undefined`; if the current time is after the second `<time>`, then `<time.constraint>` is `false`; otherwise, `<time.constraint>` is `true`.

24
For the after construct, if the current time is before <time>, then <time_constraint> is undefined; otherwise, <time_constraint> is true.

Throughout the execution of $S_i$, DFTM continuously evaluates the <time_constraint>. If the evaluation returns false at some point, a timeout event occurs which signals the failure of the execution of $S_i$.

- `<dependency_pair> ::= <ext_bool_expr> : <id> ;
  |   <ext_bool_expr> : accept ;`

The construct defines an execution dependency. The subtransaction indicated by <id> is eligible to be scheduled if the <ext_bool_expr> on which it depends is true. If its governing <ext_bool_expr> is false, a false value is bound for the subtransaction, as if its execution has failed.

Subtransactions can be executed in parallel if they do not depend on any <ext_bool_expr> or if the <ext_bool_expr> on which they depend are true and their time constraint and guard are both evaluated true.

`accept`, a reserved word, indicates the final status of a global transaction ($G_j$). If its value is true, then $G_j$ succeeds; if false, then $G_j$ fails; otherwise, $G_j$ continues to run until accept becomes true or false. The accept is true or false if the <ext_bool_expr> on which it depends on is either true or false; otherwise, its value is undefined.

- `<partial_suc_ext_bool_expr> ::= ( <number> : <subtrans_var_list> )`

The partially successful extended boolean expression is true only if at least <number> subtransactions in the <subtrans_var_list> construct have the value true. Its value is false if DFTM finds that there will not be <number> subtransactions in the <subtrans_var_list> construct that are true. Otherwise, its value is undefined.

- `<acceptable_sets> ::= <acceptable_sets> , ( <subtrans_list> )
  |   ( <subtrans_list> )`

When the accept is undefined, DFTM continues to execute subtransactions whenever possible until the accept becomes true or false. The false value indicates that the execution of the global transaction has failed, and thus all its subtransactions must be aborted. The true value indicates that the execution of the global transaction has succeeded; in this case, different acceptable sets, each consisting of a set of subtransactions, will be listed. The user is asked to determine the preferred set. The subtransactions in the preferred set will be committed; other subtransactions, of course, will be aborted.