1988

Distributed Operation Language for Specification and Processing of Multidatabase Applications

Marek Rusinkiewicz

Kanchei Loa

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

Report Number:
88-834

DISTRIBUTED OPERATION LANGUAGE FOR SPECIFICATION AND PROCESSING OF MULTIDATABASE APPLICATIONS

Marek Rusinkiewicz
Kanchei Loa
Ahmed Elmagarmid

CSD-TR-834
December 1988

DISTRIBUTED OPERATION LANGUAGE FOR SPECIFICATION AND PROCESSING OF MULTIDATABASE APPLICATIONS*

Marek Rusinkiewicz†
Kanchei Loa†
Ahmed Elmagarmid‡

Computer Sciences Department
Purdue University
CSD-TR-834
December 1988

Abstract

In this paper we address the problem of interconnecting multiple hardware and software systems in a heterogeneous environment. In many large organizations, multiple software systems (databases, application programs with their dedicated files, etc.) exit and we not only must provide access to all the information stored in these systems, but also we must allow them to cooperate by exchanging the data and synchronizing their execution. The proposed solution is based on a Distributed Operation Language. We define a common communication and data exchange protocol that uses STUB processes to protect the autonomy of participating software systems. Our solution is modular and can be implemented in a heterogeneous hardware and software environment, using different operating systems and different network protocols. The paper describes the design of the language and its implementation, in the context of a multidatabase system.

---

* This research is supported, in part, by the PYI award from the NSF, the AT&T Foundation, Tektronix, and NASA JSC.

† Department of Computer Science, University of Houston, Houston, TX 77004.

‡ Department of Computer Science, Purdue University, West Lafayette, IN 47907

DISTRIBUTED OPERATION LANGUAGE FOR SPECIFICATION AND PROCESSING OF MULTIDATABASE APPLICATIONS*

Marek Rusinkiewicz
Kanchei Loa

Department of Computer Science
University of Houston
Houston, TX 77204-3475

Ahmed Elmagarmid

Department of Computer Science
Purdue University
West Lafayette, IN 47907

1. INTRODUCTION

In many large organizations, multiple software systems exist (databases, application programs with their dedicated files, etc.) that are frequently not compatible with each other. To protect the investments made in such systems, we must not only provide a uniform access to all data stored in these systems, but we must also allow them to cooperate by exchanging data and synchronizing their execution. Hence, the emerging need to provide organization-wide access to data and software resources is creating a demand to interconnect previously isolated application systems. An end-user in a heterogeneous computing environment should not only be able to invoke multiple existing application systems (or hardware devices) but also coordinate their interactions. These application systems may run autonomously on different computers, may be supported by different software groups, may be designed for different purposes, and may use different data formats.

The problem of integrating programs and data from various application systems has been addressed by Gates [Gate87], who proposed a software architecture allowing a single program to access multiple applications on multi-tasking micro computer systems. He also identified office-automation programs as an important area which would benefit from the proposed architecture. Barbacci and Wing [Barb86, Barb87] proposed *a task description language,* Durra, which can be used to describe applications requiring execution of multiple tasks. Durra can be used to describe tasks to be executed and to define data paths between the source and the destination processes.

* This research is supported, in part, by a PYI award from the NSF, the AT&T Foundation, Tektronix, and NASA JSC

Heytens and Nikhil [Heyt87] developed an information support system GESTALT, for CAD/CAM applications. GESTALT, provides a common data model allowing global access to various local systems through a *local schema interface.*

As an example of a global application that needs to access and coordinate interactions of several software and hardware systems, let us consider the problem of preparing materials for a meeting in a company [Gate87]. Let us suppose that in order to prepare the materials for the meeting, data must be extracted from a DB-2 database residing on a mainframe computer, combined with additional data stored in a LOTUS file on a personal computer and spooled to a printer to prepare transparencies for the meeting. Furthermore, the mail utility should be invoked to deliver the memo announcing the meeting, together with the materials, to each attendee's workstation.

Software systems capable of executing such applications automatically do not exist and many issues introduced by interconnection of existing heterogeneous application systems have been virtually unexplored. In this paper, we will address these issues in the context of providing transparent access to *multiple heterogeneous databases.*

The paper is organized as follows. In the next section we discuss the proposed software architecture which can be used to interconnect multiple applications in a heterogeneous computing environment. Then we introduce the Distributed Operation Language (DOL) used to specify invocation, synchronization and data exchange between various software and hardware components of a distributed system. Execution of DOL programs and the implementation of a DOL interpreter is discussed in Sections 4 and 5. In Section 6 we discuss a DOL-based solution to the problem of providing access to multiple heterogeneous database systems.

2. SOFTWARE ARCHITECTURE

A major limitation of the architecture proposed by Gates is the assumption that all application packages participating in a global application must support (or be modified to support) a common data exchange protocol and a common macro language processing. In the solution described here, we extend the approach proposed by Gates by eliminating the unrealistic assumption that all application systems in a heterogeneous environment can be retrofitted to support common protocols. We also assume a more general computing environment consisting of multiple and possibly heterogeneous computers connected by a network. Our proposed solution to the problem of interconnecting partially incompatible application systems consists of several components.

The common protocol is an agreement for all components in the system on how to exchange commands and data. It represents another level of abstraction of underlying network communication facilities.

The Script Language is used to specify global transactions as well as local transaction for each (member) application system. It also defines all events associated with a transaction, the sequence of events, logical dependencies, and the maximum degree of concurrency. In some cases, the end-users can specify their requests by directly providing a program in the script language and then executing it. On the other hand, a script program could be generated automatically by a software system. For example, in a multidatabase system users may specify their queries in a multidatabase language, e.g. in extended SQL [Litw87]. Such a query may be parsed, verified and decomposed by a multidatabe system and a script program corresponding to the query evaluation plan can be produced automatically.

The Program Interpreter of the script language is the "execution engine" for programs written in the script language. It initiates and terminates the execution of various application systems required by a global program and acts as a supervisor for all transactions in the system.

STUB processes are designed to preserve the autonomy of each participating application system [Elma88]. Resources within each application can be accessed through the STUB processes without modifying existing application systems. The common protocol is enforced only between the program interpreter and the STUB processes. Hence, a STUB process provides another level of abstraction for each existing application system.

Under the proposed structure, all details of network communication, data conversions, and access schemes are hidden inside the program interpreter and STUB processes. The program interpreter makes access to all resources in various application systems transparent to the users, as if they were locally available and under a centralized control, thus providing *location and distribution transparency.*

The execution of global applications in a heterogeneous environment consist of actions (e.g. send information, receive information, wait for information, perform query) and primitives for parallel processing (e.g. fork and join). A script language suitable for specifying the distributed execution of transactions should not only support the above primitives but also should allow definition of global and local computations of a transaction, specification of the creation of local or remote processes at target sites, data and control flow between processes and exception handling.

The processing of multidatabase queries in OMNIBASE will be discussed briefly in Section 6 to show a possible implementation of common protocol, script language, program generator, program interpreter and STUB processes.

3. SYNTAX AND SEMANTICS OF THE DO LANGUAGE

In this section, we describe a script language, called Distributed Operation Language (DOL). The language can be used to specify a distributed execution of global applications in a heterogeneous computing environment. Below, we will describe informally the syntax and actions for each statement.

```
OPEN [DYNAMIC] [STATIC] [PIPE] (task-name)
    AT site-name
    AS #X
```

The OPEN command is used to establish a reliable communication link with the task **task-name** at site **site-name**; this communication link will be referred to as #X. The task is a STUB process of an application system known to the program interpreter, such as a graphics system, DBMS or a statistical package. After establishing connection with a STUB process, the interpreter asks the STUB process to get access rights (login) to the local application system on behalf of the interpreter, check the status of the local system and be ready for further access. Three modes can be chosen to OPEN a task, namely default, DYNAMIC and STATIC modes.

Default mode: The interpreter checks the existence of a STUB process at the specific site for the specific task. If the STUB process exists, a connection to it is established, otherwise the OPEN command creates a remote (or local if the interpreter and the application are on the same site) STUB process at the specified site before connection is established.

DYNAMIC mode: Whenever the interpreter OPENs a task, a new STUB process is created and a communication link is built with this newly created process.

STATIC mode: The interpreter attempts to establish a connection with an existing STUB process identified by **task-name**. No new STUB process is created and if the STUB process does not exist, the OPEN command fails.

Two data transfer modes between processes are a batch mode and a pipe mode. The basic transfer unit for the batch mode is a file. The sender sends the name of the file along with the required access codes to the receiver. It is the responsibility of the receiver process to pick up data from this file. It is assumed that a file transfer utility is provided by the underlying network environment. In the pipe mode, the sender sends out data, record-by-record, to the receiver. A producer-consumer relationship exists between the sender and the receiver. Both parties have to agree on a predefined data transfer protocol before the actual data transfer starts. The batch mode is the default for the data transfer. PIPE modifier in the OPEN clause is used to check whether a STUB process is capable of handling both data transfer modes.

CLOSE #X

The interpreter ask the STUB process to stop all the ongoing activities, clear all relevant data structures and connections, then exit.

SCOPE #X

{ text }

ENDSCOPE TO #Y [PIPE] [THROUGH #Z [PIPE] [#Z1 [PIPE]]]
[, #Ya [PIPE] [THROUGH #Za [PIPE] [#Z1a [PIPE]]]

The interpreter sends the text between SCOPE and ENDSCOPE to the STUB process of #X without any interpretation. It is the responsibility of the STUB process to interpret the text and to invoke appropriate local actions. This command also specifies where and how the results should be sent to. #Y, #Ya, ... specify the sink destinations of the result, i.e. the results could be sent to multiple destinations. PIPE specifies that data should be transferred to the destination process record-by-record.

THROUGH specifies the path through which the result is sent to its destination. The THROUGH option in the ENDSCOPE means the process #Z should act as a filter that takes its

input from the process #X, performs predefined operations with the incoming data, then sends it to filter process #Z1, or to the final destination process #Y. Multiple filter processes can be defined for each final destination and are executed in the specified order. The THROUGH clause is optional.

BEGIN

END

The commands between BEGIN and END should be executed sequentially, i.e. the execution of a command must not start until the previous one successfully completes. If a command inside a BEGIN-END block fails, the interpreter invokes an exception handler.

COBEGIN

COEND

The commands between COBEGIN and COEND could be executed concurrently, i.e. a command could be started before the previous one completes. The COEND acts as a synchronization point. The interpreter waits at COEND until all the ongoing commands complete or abort. If one of the pending commands fails, then the interpreter aborts all unfinished commands and invokes the exception handler. Only after all of them complete successfully, the interpreter goes to next statement.

BEGIN-TRANSACTION

END-TRANSACTION

The statement BEGIN-TRANSACTION in the program is the *begin-transaction* primitive. END-TRANSACTION is the *commit* primitive for the transaction. If any of the statements between BEGIN-TRANSACTION and END-TRANSACTION fail, the *abort* primitive is invoked

automatically. These transaction control primitives affect all the tasks of the transaction.

In the next section, a sample DOL program is presented to show how the DOL statements can be combined to specify a distributed transaction in a heterogeneous computing environment.

4. USING DOL TO DESCRIBE DISTRIBUTED ACTIONS

As a first example, let us consider the meeting-preparation problem introduced earlier, assuming the following system configuration:

- The name of the site with the main database and the company-wide mail utility is IBM4330-1
- The site name of the minicomputer running the printer spoolers is VAX1
- PC1 is the site name of the PC containing the LOTUS data.

The application can be specified by the following program.

```
BEGIN-TRANSACTION

 OPEN  DYNAMIC PIPE ( DB2 )
              AT IBM4330-1
              AS #1

 OPEN  PIPE ( LOTUS-123 )
              AT PC1
              AS #2

 OPEN  PIPE ( LOTUS-FILTER )
              AT IBM4330-1
              AS #3

 OPEN  STATIC PIPE ( LASER-SPOOLER )
              AT VAX1
              AS #4

 OPEN  ( MAIL )
              AT IBM4330-1
```

```
        AS #5

COBEGIN
   SCOPE #1
    { local commands for DB2 }
   ENDSCOPE TO #2 PIPE THROUGH #3 PIPE

   SCOPE #2
    { local commands for LOTUS-123 }
   ENDSCOPE TO #4 PIPE, #5
COEND

SCOPE #5
  { local commands for Mail utility }
ENDSCOPE

SCOPE #4
  { local commands for the spooler of a laser printer }
ENDSCOPE

COBEGIN
   CLOSE #1
   CLOSE #2
   CLOSE #3
   CLOSE #4
   CLOSE #5
COEND

END-TRANSACTION
```

In the above example, the first SCOPE-ENDSCOPE statement specifies the local commands to be used by DB2 to retrieve data from a database (e.g. SQL SELECT statements). The data extracted from DB2 will be sent to LOTUS-123, record by record, through a filter task LOTUS-

FILTER. There are producer-consumer relationships between DB2, LOTUS-FILTER and LOTUS-123 tasks. LOTUS-123 is a a filter task needed to convert the data from DB2 format to LOTUS format. The PIPE attribute has been used in the OPEN statements for the above three tasks to improve the efficiency of the data conversion. The STATIC attribute in an OPEN statement can be used to limit the number of tasks at the target site. This consideration may be important for micro-computer sites.

Nested transactions can be expressed naturally using DOL. As a second example, let us consider an implementation of a data processing system for a travel agency. A transaction in such a system may consists of making flight, hotel and car rental reservations and may need to access multiple databases belonging to various companies. Obviously, it would be extremely inefficient to implement a DOL interpreter capable of executing such transactions on a workstation. It may take quite a long time to run such a transaction if the workstation uses telephone lines to dial databases one at a time.

To continue our example, let us assume further that we have access to two powerful communication sites, one designed to efficiently access all airline databases, and the other one designed to efficiently access all hotel databases. The following program shows how a DOL interpreter, with very limited capabilities, can handle a complex transaction by nesting and distributing the computations to other more powerful DOL interpreters.

```
BEGIN-TRANSACTION

 OPEN DYNAMIC (Supervisor1) AT IBM3090-1 AS #1
 OPEN DYNAMIC (Supervisor2) AT VAX785-1  AS #2
 OPEN (Text_Processor)      AT PC0       AS #3

 COBEGIN
  SCOPE #1
   BEGIN-TRANSACTION
   { Program1 written in DO Language
    for retrieving flight information }
   END-TRANSACTION
  ENDSCOPE TO #3
```

```
  SCOPE #2
    BEGIN-TRANSACTION
    { Program2 written in DO Language for
      retrieving hotel information }
    END-TRANSACTION
  ENDSCOPE TO #3
COEND

SCOPE #3
  { local commands for Text-Processor }
ENDSCOPE

CLOSE #1
CLOSE #2
CLOSE #3

END-TRANSACTION
```

In the above example, DOL is used to specify both the global transaction and the sub-transactions which will be executed by other DOL interpreters. The Supervisor1 at site IBM3090-1 is the DOL interpreter capable of efficiently accessing airline databases. The Supervisor2 at site VAX785-1 is another DOL interpreter capable of efficiently accessing hotel databases. The Text-Processor is an application software used to combine the retrieved data into a pre-defined format.

In a large heterogeneous environment, it is not practical to implement a "super" interpreter capable of accessing hundreds of physically distributed heterogeneous application systems, on one site. However, we can implement several "specialized" interpreters, each designed to efficiently control a class of application systems, on different sites and divide a large transaction into a hierarchical collection of subtransactions. All these subtransactions can be expressed in the DO language and can be controlled by the root transaction, also written in DO language. A subtransaction is dispatched to the interpreter that can execute it in a most efficient manner. As we have seen in the above example, DOL can be used to easily express nested transactions in such environments.

5. EXECUTION OF DOL PROGRAMS

A program written in DOL can be either submitted to a DOL interpreter for execution or compiled into an execution image and executed later. With the compilation approach, access schemes, communication parameters, and subtransactions are determined in advance. The compiled image of a DOL program can be executed efficiently but lacks run-time flexibility. Hence, this approach is suitable for applications executed routinely in a static environment.

For "ad hoc" transactions, such as the one discussed in the introduction, the interpreter approach is much more suitable. The interpreter may delay binding of access schemes and communication parameters of a transaction until the DOL program is executed, so the execution plan can be adjusted dynamically. A DOL program is parsed into primitive actions that are invoked in accordance with the precedence and concurrency limits specified in the program. To accomplish this task efficiently, the interpreter should be multi-threaded (i.e. capable of dealing with more than one action at a time).

An interpreter of a DOL program should be context-free, i.e. it should not be required to remember past programs. All information needed by an interpreter to execute a single program is embedded in that program. For this reason, the interpreter and the DOL program generator should be separate. In particular they can reside at different sites of the network. The interpreter is a "generic execution engine", that can perform its functions on behalf of different DOL program generators. The interpreter performs all transaction control functions (i.e. begin-transaction, commit and abort).

The interpreter is responsible for providing access to all participating application systems. A possible way to provide such access is to modify all member systems, so that they obey the common communication, process synchronization and data exchange protocol. However, usually it is not possible to enforce a common protocol between pre-existing application systems which come from different vendors and were designated to run in different computing environments. Another possibility is to incorporate into the interpreter the information about access schemes used by all application systems to access their resources. However, this would result in the interpreter being too complex to deal with and very inflexible. Every time when we need to accommodate a new application system, the interpreter would have to be modified.

To provide a common protocol in a heterogeneous environment consisting of autonomous application systems we propose to use STUB processes [Elma88]. STUB processes are designed to preserve the autonomy of each application system. They are local agents of the interpreter, through

which it can access the resources in each application. The interpreter initiates and terminates STUB processes as needed. There exists a master-slave relationship between the interpreter and each STUB process. The interpreter and STUB processes communicate through messages and can reside at different sites.

Whenever the interpreter wants to invoke a local application system, it communicates with an appropriate STUB, sending to it local commands, that should be executed by the application system. STUB acts as a proxy user: submits local commands to the application system, gets the results and sends them to the destination in a predefined format. Both the destination(s) and sending protocol(s) are specified by the interpreter.

A STUB process is single-threaded: it performs a single function completely, before initiating a new one. STUB processes are context-free and are not required to remember past requests. All information needed by a STUB process to execute a single request from the interpreter is contained in the request. A STUB process and its target application system can reside at different sites. A STUB process may perform its function on behalf of different interpreters or it may be dedicated to a particular interpreter. In most applications, it is convenient to allocate the interpreter at the program generator site and the STUB processes at application system sites, because the interprocess communication is usually minimal in this case.

6. IMPLEMENTATION OF A DOL INTERPRETER

In this section, we will discuss the implementation of a DOL interpreter using as an example a multidatabase environment. A Multi-Database System (MDS) is a system for the management of several databases, without a global schema [Litw86]. Each local database may be under the control of a centralized or distributed DBMS. In the latter case, the DDBMS is treated just like a centralized DBMS. OMNIBASE is an experimental multi-database systems that provides access to multiple pre-existing databases, supporting their own applications and end-users [Rusi88]. It is a heterogeneous DDBMS with federated control structure, that creates loose federation(s) of database systems which can cooperate in fulfilling user requests.

The multi-database system keeps track of data descriptions of the portions of the local databases, often called *export schemas* that have been made available to the global users. Figure 1 shows a diagram of the multi-database system modules. We briefly describe the functions of each module, and typical flow of information and control among the various modules.

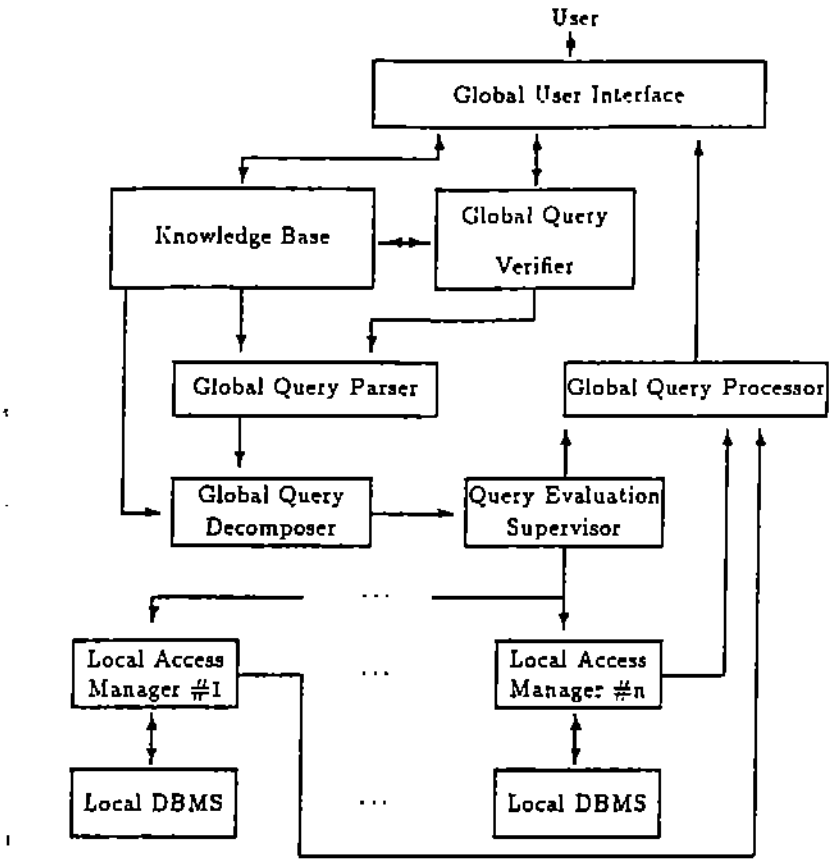

Figure 1. Global query processing in OMNIBASE

The *global data dictionary* contains information about export schemas, DBMSs, and locations of member databases. This information is accessed by other system modules whenever a global user issues a request. Such requests are sent to the global query parser and *decomposition module*, which decomposes the query into a coherent collection of subqueries.

The subqueries and global queries are passed to the DOL *program generation module* which produces a query execution plan formulated in the distributed operations language (DOL). This plan is then executed under the control of the *query evaluation supervisor* (QES), which directs the subqueries to their respective local systems.

At each local site, a STUB process called LAM, (local access manager) which is part of the multi-database system, translates the query to the local database language, submits it to the local DBMS, and produces intermediate results. These results are then directed by the query evaluation supervisor to other sites where further processing may be performed. Finally, the global query is executed to form the final query result.

The DOL OPEN statement creates a LAM process at the specified local or remote site and establishes a reliable communication channel with it. After the channel has been established, the supervisor sends an internal LOGIN command to the LAM in order to obtain access right to the local DBMS system. The DOL CLOSE statement results in an internal command EXIT being sent to the LAM process. Another internal command that a supervisor may send to the LAM is PROCESS-SCOPE which requests the LAM to start processing a subquery. The name of the file containing the subquery and the name of the file where the result is to be placed, along with the site id where the result must be sent, are passed to the LAM as arguments of the PROCESS_SCOPE command.

In the context of multi-database operations, the DOL can be used to express various semi-join based query processing algorithms, taking into account different cost functions, node and network characteristics, etc. Another advantage of using a script language is that the balancing of the system load can be taken into account while generating DOL programs. Thus, the OMNIBASE could optimize the query evaluation by optimizing the DOL program generated for each query. It progressively combines the temporary relations resulting from the execution of subqueries and eventually produces the final result of the global query.

The current prototype of OMNIBASE is implemented on a group of VAX/VMS machines using mailboxes for interprocess communication. Remote processes communicate via Virtual Circuits (VCs) under the control of DECNET. VCs are explicitly created by OMNIBASE system processes and are used only to exchange synchronization messages. For a given transaction, VCs are established upon demand, and remain active until the end of the transaction or until they are deactivated by the supervisor. When an OPEN statement is executed, the supervisor creates the target remote process and establishes a unique VC to communicate with it. When the supervisor executes a CLOSE statement, that VC is disconnected.

If a file is to be transferred in a batch mode, a message containing the name of the file to be transferred is placed on the VC of the target remote process. Then it becomes the responsibility of the target process to complete the transfer, which is actually done by calling the DECNET file transfer utility. After the completion of the transfer, the target process places an acknowledgement message on the VC where the file transfer request originated.

The communication model described above enables the concurrent execution of tasks assigned to subordinate processes. For example, tasks scheduled to evaluate subqueries of a global query are allowed to fetch subqueries, evaluate them and return the evaluation results is a clearly concurrent

manner.

7. CONCLUSIONS

Providing uniform and systematic access to multiple existing hardware and software systems is recognized as one of the major problems facing the information industry. Our approach will be applicable in many large organizations with existing heterogeneous computing environments, where the current network services such as electronic mail, remote login and file transfer are no longer sufficient. The proposed software architecture addresses this problem and promises a general and implementable solution. Similar work in the area of communication networks lead to the emergence of the "open network" concept. We believe that our work may contribute to the development of an analogous concept of "interoperable data processing systems". To achieve this goal, we have initiated the InterBase project, whose objectives are to complement the query formulation and processing solutions developed in OMNIBASE, with the support for atomic, recoverable and persistent multidatabase transactions.

8. ACKNOWLEDGMENTS

The authors gratefully acknowledge the participation of other members of the OMNIBASE project at the University of Houston and the InterBase project at Purdue University in the development of the prototype implementation of a DOL interpreter.

References

Barb86.
Barbacci, M.R. and J.M. Wing, "Durra: A task-level description language," *Technical Report*, no. CMU/SEI-86-TR-3, Software Engineering Institute, Carnegie Mellon University, 1986.

Barb87.
Barbacci, M.R., "A task-description language for real-time applications," *The SEI Annual Technical Review*, Software Engineering Institute, Carnegie Mellon University, 1987.

Elma88.
Elmagarmid, A. K. and A. A. Helal, "Supporting Updates in Heterogenous Distributed Database Systems," *Proceedings of the International Conference on Data Engineering*, 1988.

Gate87.
Gates, Bill, "Beyond Macro Processing," *BYTE Bonus Edition*, Summer 1987.

Heyt87.
Heytens, M.L. and R.S. Nikhil, "Gestalt: an epressive database programming system," *Private Communication*, MIT, December 1987.

Rusi88.
Rusinkiewicz, M. and et.al., "OMNIBASE: Design and Implementation of a Multidatabase System," *Distributed Processing TC Newsletter*, vol. 10, no. 2, pp. 20-28, IEEE Computer Society, November 1988.

Litw86.
Litwin, W. and A. Abdellatif, "Multidatabase interoperability," *Computer*, vol. 19, no. 12, December, 1986.

Litw87.
Litwin, W. and A. Abdellatif, "An Overview of the Multi-Database Manipulation Language MDSL," *Proceedings of the IEEE*, May, 1987.