Supporting Distributed Query Processing in a Heterogeneous Environment

George Semeczko, Stanley Y.W. Su, Tsae-Feng Yu, Fang Wang
Database Systems Research & Development Center
University of Florida
Gainesville, FL 32611
{georges,su,yu,fwang}@cis.ufl.edu
TR-96-032
August, 1996

Abstract
To make effective use of distributed systems acting together to form a virtual enterprise, global coordinated operations must be possible. Such operations may be mere queries or complex update operations. In this paper, we describe an existing system at the University of Florida which not only models a Virtual Enterprise using a common object-oriented semantic association model, but acts as a trader and repository for common information. The primary objective of this paper is to show how this existing system is being extended to handle complex global operations stated in the form of Object Query Language queries. These queries are then processed in a distributed manner, including any necessary object migrations. Problems addressed in this paper include: provision of support information in the meta-model, allocation of global object instance identifiers, issuing and coordination of subqueries to the legacy systems in the Virtual Enterprise, and the issuing and coordination of method executions.

1. Introduction
Virtual Enterprises (VE) consist of a group of legacy computer systems that cooperate in order to carry out some common task or goal. Not only must these legacy systems share data and services, but they must also coordinate their actions. These problems can be tackled using such services as traders or binders, Coulouris [3]. This allows for the various legacy systems to advertise and find appropriate services available within the VE. Given that the interactions within a VE can be very complex, these mechanisms may be insufficient to handle the cooperative computations desired.

An extension of this approach has been devised in the National Industrial Information Infrastructure Protocols (NIIIP)1. Here, an Active Object-oriented Knowledge Base Management System (AOOKBMS) is used to not only act as a trader as mentioned above, but it also provides a uniform model to describe the entire VE. As this model is Object-oriented, it not only models the functional attributes of the system by way of methods of objects, but also the data attributes and the relationships or associations, between objects. Object-oriented solutions to similar problems have also been described in Kottmann [5], Papazoglou [6] and Tirri [15] with a survey of several systems in Bertino [2]. One of the major differences between these works and that in the NIIIP project, is the emphasis on aiding VE-wide operations and not restricting them. Hence, the approach taken is to have an AOOKBMS that is easily extended, thus allowing for many types of legacy systems to participate in the VE. It also allows for the storage of shared information in the knowledge base that can also be used to aid in the VE computations.

All of the data stored in the knowledge base may be queried by an Object Query Language (OQL) which allows the formulations of complex queries. Being an active system, rules may also be specified to aid in the activation of methods under given conditions.

As explained in Su [14], research into this project was motivated by the work of two groups: the Object Management Group (OMG), described in Sessions [9], and the International Standard Organization’s Committee on the Standard for the Exchange of Product model data (ISO/STEP). The first took a method-based approach to interoperability whereas the second concentrated on product modelling and data exchange. Integrating these two approaches was achieved in the NIIIP project.

Even with such a rich trading and modeling system, complex coordinated operations are currently still limited. This is due to the fact that programs must still be written to use the interfaces described by this trader to carry out the coordinated operations. In this paper, we describe how this system is currently being extended to eliminate this limitation. This is being done by extending the existing query processing mechanism into a distributed query processing system. Queries will not be limited to the meta-model of the VE, but cover the data and operations within the VE and its

---

1 NIIIP is an on-going project being carried out by a consortium of industrial companies, universities and government organizations. It is funded by DARPA (see acknowledgment).
legacy systems. Hence, activation of methods can be imbedded in and coordinated by VE-wide queries on the VE data. This simplifies the development of the coordination software. Such software is often specified in very complex procedural languages. Here though, the coordination is automatically created using the OQL query and meta-data about the VE and its data.

In designing this system some of the problems that were tackled included:
• allocating global instance identifiers
• issuing and coordinating subqueries consisting of data accesses, constraint evaluation, method calls and association pattern evaluation
• data structures to assist in the operations above and their optimization
• data conversion between legacy systems

The rest of this paper is structured as follows. In Section 2, the existing system is described. This includes the overall architecture, its meta-model, object migration abilities and its current query processing capabilities and structure. Section 3 then provides the extensions to this system. This covers extensions to the meta-model, allocation of global instance identifiers and the new distributed query processing architecture. The appropriate data models are also described and are accompanied by an example. Section 4 concludes with a brief statement on the current implementation status.

2 Current System

In the NIIIP project, heterogeneous legacy systems involved in a VE are coordinated via an active Object-oriented Knowledge Base Management System (OOKBMS). Each of these legacy systems interacts with this OOKBMS via a wrapper. The wrappers perform any conversions necessary from and into the common global object-oriented model used in the OOKBMS. In this way, any legacy system could communicate with another via this common object representation. This is represented in Figure 1.

It is possible for homogeneous legacy systems to communicate directly as shown between systems 2 and 3 in Figure 1.

In this section, the existing structure of the OOKBMS being used in this project is described. This OOKBMS is called the OSAM*.KBMS as described by Su in [13] and is highly extensible. It has been developed and is under further extension at the University of Florida. The areas of this system described in this section include its architecture, its meta-model and its query processing capabilities.

2.1 OSAM*.KBMS Structure

The OSAM*.KBMS is based on a semantic association model, OSAM*, as described in Su [10] [12] and is an active OOKBMS, Su[11]. Not only does it model existing interfaces and hence act as a binder or trader of services, but it can also be used to generate interface stubs. These can be used to either create further services at the legacy system or to present a cleaner interface via the wrappers. Currently, the OSAM*.KBMS system is made up of several components as shown in Figure 2.
Details of these components can be found in Su [13]. The Query Processor is discussed later in this paper. Before discussing any further architectural issues though, the underlying model implemented by this architecture needs to be covered. The object model used is defined by a meta-model. A part of the current version\(^2\) of the meta-model, defining OSAM*, is shown in the Figure 3.

The primary components of the model are classes and the associations between them. Classes are defined by attributes (associations with domain classes), methods and rules. As with most object-oriented models the concepts of generalization (labelled by a G) and aggregation (labelled by an A) are catered for. Here though they are included with the other possible associations between classes. They provide a much more rich foundation to describe relationships between objects. These include: Generalization (inheritance), Aggregation and Interaction. Other work at the University of Florida have added to these three using a model extensibility technique, but for the purposes of this paper, this basic set is sufficient. Examples of Aggregation and Generalization can be seen in the meta-model above. For example, the class Class is a generalization of the classes Entity and Domain. Class has two aggregation associations with domain classes (i.e. two attributes) and four aggregation associations with four entity classes.

The two string attributes are schema and name. The other four aggregation associations are with the classes Site, Method, Rule and Assoc. The last three being of the set type. This allows for each class definition to contain a name, the schema it belongs to, a set of methods, a set of rules, a set of associations with other classes, and where a class is defined. Some of these aspects are now discussed a little further.

In the meta-model above, the class Assoc also has a set association with the class AssocLink. This shows that an association can be defined by many links. Assoc is the generalization of the association classes Generalization, Aggregation and Interaction. For a detailed explanation of the association types see Su [10].

To aid in the migration of objects throughout the VE, a protocol was developed in Semeczko [8] using the Knowledge Query and Manipulation Language (KQML) by Finin [4] and is supported by this meta-model through the classes MethodAlloc, MethodSource and MethodExec.

Not shown in this meta-model is the fact that every object instance stored in the OOKBMS is identified by an Instance Identifier (IID). This is made up of two components: Object Identifier (OID) and a class identifier. In this way, given an object instance, the KBMS not only can uniquely identify the object, but also the class to which the object belongs.

### 2.2 Query Capability

The current query processor only works upon the OSAM*.KBMS and has no distributed capabilities at all. It is a fully functioning query processor evaluating the queries specified in OQL described in Alashqur [1] and Potharaju [7]. An OQL query takes the following form:

```
CONTEXT association pattern expression
[WHERE conditions ]
[SELECT object classes ]
DO object method(s)
```

![Figure 3-Current Meta-model](image-url)
Space limitations restrict the discussion on OQL queries, so only a brief description is possible here of the features available. In the CONTEXT clause, an association pattern is expressed by detailing classes and the association operators between them. Such operations include: association (*), non-association (!), compliment (!) as well as branching operations of “and” and “or”.

The instances that match this pattern and form the result subdatabase can then be restricted further by the optional WHERE clause. This works in a similar manner to the WHERE clause in an SQL query. It is allowable though, to have WHERE restrictions based on the results of method calls.

Another optional component of an OQL query is the SELECT clause. This clause is also similar to that in an SQL statement except that instead of projecting on attributes, the projection occurs on classes. That is, this clause eliminates classes from the result subdatabase.

Finally, the DO components of an OQL query specifies what operations are to be performed on the result subdatabase. These operations maybe either system-defined maintenance operations (e.g. InsertObject) or they may be user-defined operations defined as methods belonging to the objects.

An example of an OQL query from Alashqur [1] is presented:

```oql
CONTEXT Faculty * Section *
    and (Course * Department, RA)
    WHERE Department.name = 'EE'
SELECT Faculty [name], c#
DO display
```

This query can be used to answer the following request for a university database:

“Display the name of any faculty member who is teaching any section of a course that is offered by the ‘EE’ department, provided that the section is taken by at least one graduate student who is an RA. Also show the course number.”

---

The OQL language is very rich and powerful and includes facilities such as universal and existential quantification as well as aggregation operations. For more details on the OQL language readers are referred to either Alashqur [1] or Potharaju [7]. Given this overview of the language, its processing can now be discussed.

The query processor architecture is shown in Figure 4. In this existing design, the textual query is parsed first for accuracy and then converted into a Query Tree (QT) by the Query Tree Transformer. This QT has the structure shown in Figure 5.

In this structure the different components of the query are split into several subtrees. Each node in these subtrees holds the data necessary to determine what the operation is and what its inputs are. After a QT is created for the query, the CONTEXT component is passed to the OQL engine for evaluation of the associations, the WHERE clauses and the SELECT clause. This is handled by the Context Subtree Handler which, upon return of the results, will pass the entire QT over to the Operation Subtree Handler which performs any required operations via the method calls for the objects identified in the previous result. The results of these operations are then returned to the issuer of the query.

---

3 Rules in this model are Event-Condition-Action-AlternativeAction (ECAA) rules.
The current QP is limited to querying only the OOKBMS which the OQL Engine works upon. For the VE, this means that queries can be formulated to return information about the VE model, but not the content of the distributed data of the VE held by the legacy systems. As it is not designed to handle distributed queries, it must be redesigned to handle them.

3. Distributed Processing

In this section, the design of a Distributed Query Processor is presented. Some of the problems handled by this design include: meta-model information for query optimization, allocation of global instance identifiers, performing the distributed tasks of association pattern determination and operation execution, and optimizing some of the inter-site class constraint evaluation.

In order to put into context how the distribute processing will work, an example is needed. The example presented here shows what will happen to an OQL query that is to be processed in a distributed manner. The rest of this section will then be concerned with how this is achieved.

For an example, the following query is used:

```
CONTEXT A * B * C * D * E
WHERE A.a = 5 AND B.b = 10
    AND A.f > B.g AND C.c < E.e
    AND D.d < 10 AND D.d > 5
DO C.methodC(), E.methodE()
```

While bereft of real meaning, this example is sufficient to present the problems that will be encountered in a virtual enterprise. In this example, we will assume that the OSAM*.KBMS is stored at Site1, the classes A and B are located at Site2, C and D at Site3 and finally, that E is found at Site4.

What is required of the distributed query processor (DQP), is to generate subqueries to each of the 3 sites to perform those operations at that site. In this case the following is required:

1. generate the following subqueries:
   - CONTEXT A * B WHERE A.a = 5 AND B.b = 10
   - CONTEXT C * D WHERE D.d < 10 AND D.d > 5
   - CONTEXT E

2. the results of each subquery is then returned to the OSAM*.KBMS for the following actions (assuming a naming of result patterns as ResSite1, ResSite2, ResSite3):
   - CONTEXT ResSite1 * ResSite2 * ResSite3

3. The result of this can then be used to get the values for C.c and E.e and A.x and D.y for all qualifying OIDs by issuing retrieve commands for each result entry. These values can then be used to reduce the result pattern even further. Some optimization techniques will be employed here to gather this data at the time of association evaluation. This should also cater for methods within WHERE clauses.

4. Finally, for the result objects, methodC and methodE will be activated at Site2 and Site3 for the respective objects.

5. Any output from these methods would be returned to the issuer of the query

How this query is to be processed by query processor components is discussed in the following sections.

3.1 New Meta-Model

To support the data requirements of distributed query processing, the meta-model requires additional information for query optimization. This data refers to either the objects and their components or to the distributed system infrastructure.

Information regarding the objects and their components includes: the location of objects and their components and the location of association data. For some query optimization methods, information regarding the size of object components may also be needed. This data may be generated using statistical data.

For the system infrastructure, data regarding the system configuration as it pertains to the network links and processors. Factors for which statistical data maybe required for both of these areas may include: speed, reliability, availability, reliability and cost.

Another aspect that is very important in regard to the complexities of the OQL language and its ability to be used in a VE, is operation processing ability. That is, for each site, which OQL operations are able to be performed at that site should be identified. It need not be specified how these operations are to be performed, but only that they can be performed. This caters for situations where the wrappers are able to transform OQL query operations into a locally equivalent operation.

All of these aspects regarding the meta-data have been included in a new meta-model presented in Figure 6. Here, several new classes and associations have been added to the existing meta-model. A ClassInstance class allows for the maintenance of meta-data relating to the location of class instances. In this model horizontal

---

4 - It is assumed that under normal circumstances that associations between classes stored at the same site will be stored locally. For associations that span legacy systems and hence sites, will be stored in the OSAM*.KBMS. This is consistent with it being used to store common and shared data.
fragments of classes are identified by a WhereClause and their location. Associations between Site objects allow for the modeling of any network on a point-to-point basis. For each of these sites, processing capability is also modeled. In this model, multi-processors are catered for by the use of a set association between each site and the Processor class. Each Processor object has attributes describing its capabilities. Finally, several classes have been included for statistical data. This caters for information about the size of instance data, executables and their source files and the size of fragments identified by ClassInstance. For execution statistics, ExecStats has been included and is associated with each allocated executable. This is needed as each method may execute differently at different sites.

While not complete for all possible forms of query optimization, the additions proposed are sufficient for the purposes of an initial implementation. As the meta-model is easily extended, future requirements can be easily accommodated.

### Figure 7- Distributed Query Processing Architecture

#### 3.2 New Query Processing Architecture

As distributed query processing requires distributed coordination and more sophisticated query optimization, a new query processing architecture is needed. The new architecture is shown in Figure 7. The overall structure is very similar to that in the centralized version shown in Figure 4. Besides the obvious processing differences in the modules, the major change is in the handling of the Context and the Operations.

The Query Parser makes use of the existing query parsing software to verify the OQL query and create internal structures for use in Query Tree (QT) construction by the New QT Transformer.

Due to the needs of a DQP, the QT structure requires some modification. In the centralized version, the QT had two major subtrees: Context and Operation. In the new structure, this is still seen as a necessary division. However, in the centralized version, the context component was then subdivided into three other subtrees: association pattern expressions, WHERE clauses, and the SELECT clauses. In the distributed version, it is essential to construct subqueries that contain elements of each of these components and then assign their evaluation to remote sites. For this reason, this second subdivision of the QT has been dropped in the new QT structure. The design for this new QT is defined by the schema shown in Figure 8.
Figure 8- New Query Tree Definition

In this new QT, four types of nodes in the tree can be identified. Each node represents one of four types of operations to be performed: instance access, unary operations, binary operations, and do operations. First, the generalized node definition is discussed and then its specializations covered.

For each node a descriptor, **NodeType**, is used to specify the type of node and whether the operation that it pertains to has been **Completed** or not. If it has been completed then the node is then associated with a **PatternTable** which represents the subdatabase which is the result of all operations in the subtree that has this node as its root. The location where the operation represented by this node is to be performed is specified by its association to a **Site** instance. Also, as nearly all operations can be coupled with a **WhereClause**, one is included as part of the common node specification. The final attribute, **Retrieve**, is discussed in a later section, but is used for optimization purposes.

The first specialization considered is the **LeafNode**. This represents the initial access to a set of class instances for the one class. It may be restricted by a **WhereClause** that consists of only intra-class constraints.

In the **UnaryNode**, unary query operations may be applied to the results of child node. This too may have restrictions applied in a **WhereClause**. This **WhereClause** may refer to any of the classes in the **LeafNodes** that exist in this subtree.

The **BinaryNode** is similar to the **UnaryNode** except that now binary query operations can be applied to the results of two child nodes. Any interclass restrictions that span both child nodes would normally be shown in the **WhereClause** attached to this node.

All the specializations of the QT Node shown so far are primarily for use in the context component of the QT. The final specialization, the **DoNode**, is for use in the operation component of the QT. This node specifies which method or set of methods is to be executed for the **PatternTable** associated with its child node. Any results from these executions are stored in the **TextResults** attribute for later presentation to the caller of the query.

Unary and binary query operations may be applied in the operation subtree, if the **PatternTable** is to be restricted further. This is useful if multiple subtrees are to be created in the operation subtree so that different sites execute methods on different sites of the **PatternTable**.

An example of the context component of the QT is shown in Figure 9. It is discussed at the end of this section.

The role of the New QT Transformer is to create a QT with all the query components taken from the OQL query. Only the **LeafNodes** have site information. This is derived from the metadata pertaining to where classes are stored. For the rest of this paper, it is assumed that all instances of each class are stored in only one site. The meta-model does allow for relaxation of this assumption.

Once the QT has been created with this basic site information, the query optimizer takes the QT and optimizes it to create a new QT that can be used as a query execution plan. This involves not only manipulating the order in which the operations are performed in accordance with the association algebra, Su [12], but to then add site information for all nodes in the QT. In this paper, query optimization algorithms are not discussed, but are the subject of further research and future papers. It is assumed for simplicity in the initial implementations of this system though, that processing occurs only at the sites where instances are stored or at the site where the OSAM* KBMS database is stored. Relaxation of this assumption can easily be accommodated by replacing the Query Optimization module with a more sophisticated one.

Given a fully optimized query plan in the form of a QT with all site information supplied, the Distributed Context Subtree Handler (DCSH) can then be passed the QT for processing. The DCSH sends subqueries to participating legacy systems and coordinates these results with the results from the local OQL Engine which operates only on the local OOKBMS. This coordination may include further query evaluation based on the many subdatabase results being sent from the legacy systems.

If certain OQL operations stated in the query are not able to be processed at the legacy systems, then objects from the legacy systems may need to be migrated to the central site for further evaluation. This topic is covered in Semeczko [8].

Completion of the context component of the query by the DCSH results in a single subdatabase
result represented by a *PatternTable* at the root node of the context subtree. This node is referenced by the operations subtree in the Distributed Operations Subtree Handler (DOSH).

The DOSH issues remote execution requests to the legacy systems where the objects are stored for each of the objects identified in the result *PatternTable*. Results from these executions are stored in the *TextResult* attributes and collected by the DOSH for presentation to the user at the completion of all operations.

The model used here also allows for the possibility of these executions being performed at sites other than the storage site if object migration was performed. These considerations are part of the query optimization task and involve updating the meta-data.

To better understand some of this work, part of the QT for the example shown at the beginning of this section, has been generated. This is shown in Figure 9. Only the context subtree is shown here.

In this diagram, the bottom five blocks represent *LeafNodes*. Each block shows the class (top-middle), the site (top-right), WHERE clause (middle) and values to be retrieved (bottom). The last one is discussed in a later section. All the other blocks represent binary operations and include the operation (top-left), the site (top-right) and any WHERE clause (middle).

Several aspects of the DQP need further explanation and are covered in the next two sections.

### 3.3 Handling Global Instance Identifiers

One of the major problems when dealing with VE’s and heterogeneous systems in general is having a global object identification scheme. Using a common model for the entire VE provides the opportunity to designate some global identifier. In the OSAM*.KBMS this identifier is in the form of the Instance Identifiers (IIDs). Due to the fact that legacy systems are involved in the VE and local autonomy is still a major issue, maintaining global IIDs is costly and prohibitive. Even with this in mind the use of global IIDs is still necessary if VE-wide queries are to be evaluated. A compromise solution has been devised for the current project.

The solution being implemented here is a combination of delayed update and IID generation on demand. Justification for this approach is based on several points:

- legacy systems may not be able to maintain IIDs
- it is too costly for the OSAM*.KBMS to maintain all IIDs with respect to both space and computational expense
- maintenance of local autonomy prevents legacy systems from advising the OSAM*.KBMS of all updates
- global IIDs may not be needed for all objects in the VE

It is assumed that algorithms can be specified within the OSAM*.KBMS on how IIDs can be generated given any legacy system object. This may involve accessing attributes of the object or other objects within the OSAM*.KBMS or other legacy system. Given these assumptions and the points above, the following scheme is proposed for maintaining and generating IIDs:

1. All objects local to the OSAM*.KBMS will have IIDs maintained automatically.
2. Local wrappers will be responsible for generating all IIDs not able to be maintained by legacy systems. These are generated only when objects are accessed for a query from the OSAM*.KBMS. That is, they are generated only on demand from the OSAM*.KBMS.
3. As IIDs are generated for legacy system objects, the wrappers will maintain a local list of the mappings from IIDs to local identifiers.
4. Wrappers will maintain a replicated copy of these IID mappings at the OSAM*.KBMS in case wrappers have volatile memory or recovery is needed.
Using this scheme, IID s are only generated as needed but with some history to speed up generation. This scheme allows for complex object structures where objects may extend across legacy systems.

3.4 Optimizing the Query Processing
In designing the DQP architecture, flexibility with respect to query optimization was of paramount concern. With any system being implemented though, some specific optimization techniques must be employed. One such aspect of concern was the evaluation of inter-class constraints between objects stored at different sites. An objective here was to reduce the amount of intersite communication needed to evaluate such constraints. This was also coupled with the fact that OQL allows for method invocation within these same constraint specifications in WhereClauses.

The example presented earlier in this paper highlights this problem. In the root of the subtree given in Figure 9, an inter-class constraint was being evaluated: A.x = D.y. A simple approach to this problem is to issue another query to each of the sites storing A and D to return the values of A.x and D.y for each of the objects identified by their IIDs in the PatternTable. This could be very costly in terms of processing and data communications.

The approach taken to alleviate this problem involves the implementation details of the PatternTable used to represent the result subdatabase. The current centralized version of the query processor uses a table of IIDs to represent the patterns that match the association pattern expression. This same structure will be used in the DQP as the IIDs can be assumed to be global.

To this structure data, values will be added. That is, for each instance in the PatternTable, the values of these desired attributes may be stored. This allows the data to be accessed during the initial access of the object instances saving data communication costs as well as accessing this data on the first pass. These savings are at the expense of storing these attribute values with the PatternTable and transmitting them with the rest of the PatternTable.

It is assumed that the query optimizer will determine when it is appropriate to use this facility. Using this facility is achieved in the QT by specifying what values to Retrieve. This field is defined for the QueryNode object in Figure 8. Examples of its use to solve the example query used in this paper can be seen in Figure 9, where the retrieve value has been specified.

This same facility is also useful when method invocation has been specified as part of the inter-class constraints in the where clause. Assuming that such method calls return simple domain values that can be stored in the PatternTable, the Retrieve attribute can be used to execute methods at the time of first pass of classes instances. Here again, it is assumed that the query optimizer will determine when it is appropriate to use this facility. For method invocation, the time and cost of execution must be taken into account when evaluating the query optimization strategies.

4. Conclusion and System Status
In this paper, an architecture has been presented to allow distributed processing of queries specified for a VE consisting of cooperating legacy systems. The approach taken has been to extend an existing OOKBMS system, the OSAM*.KBMS. This system was used as a trader and common model for a VE. It is being extended to allow VE-wide queries to be evaluated in a distributed manner utilizing the local query capabilities of the legacy systems involved in the VE.

Within this paper the major components of the Distributed Query Processor architecture have been presented. This has included: extensions to the meta-model, structure for the Query Tree and its processing for query evaluation and operation execution, generation of global IIDs, and some simple mechanisms to aid query optimization with respect to inter-class constraints between sites and method invocations within where clauses.

The existing centralized query processor is currently being extended to cater for these modifications. This includes modifications to the meta-model and the construction of several wrappers for other existing database management systems which are not all object-oriented.

Acknowledgment
This work is supported by the Advanced Research Project Agency under ARPA Order #B761-00. It is part of the R&D effort of the NIIIP Consortium. The ideas and techniques presented here are those of the authors and do not necessarily represent the opinion of other NIIIP Consortium members.

During the time of this research George Semeczko was on Professional Development Leave from the Queensland University of Technology

References


