

Design and Implementation of a CORBA Query Service Accessing EXPRESS-based Data

Jürgen Sellentin^{1,2}, Bernhard Mitschang²

e-mail: Juergen.Sellentin@DaimlerChrysler.COM, mitsch@informatik.uni-stuttgart.de

*DaimlerChrysler AG¹
Research & Technology
Dept. CAE-Research (FT3/EK)
P.O. Box 2360, D-89013 Ulm*

*University of Stuttgart²
Department of Computer Science
IPVR, Breitwiesenstr. 20-22
D-70565 Stuttgart*

Abstract

In this paper we describe the design and implementation of a CORBA Query Service targeted to access data that is defined by the EXPRESS data modeling language. EXPRESS is used primarily in engineering domains (like CAD/CAM and GIS) to describe mostly product model data (like parts explosion or product geometry).

In order to bring query facilities for EXPRESS-based data to CORBA a number of design decisions have to be taken, although the CORBA Query Service is standardized by the OMG. Among the most important and performance-indicating decisions are the definition of an appropriate query language and the description of the query result data structures. In this paper we discuss solutions to these topics and report on the experiences gained in designing and implementing our first CORBA Query Service for EXPRESS-based Data.

1. Introduction

Many advanced applications require solutions to important problems like data and system heterogeneity, system distribution, extensibility, and performance. This is especially true for the engineering domains like CAD/CAM and GIS, where mostly product model data are managed by means of separate design tools and management components. In order to conquer the overwhelming complexity of this interoperability scenario certain standardizations have been devised and applied in the systems' overall architecture. One is the definition of the international *Standard for the Exchange of Product Data* (STEP, ISO 10303 [6]). It defines a data definition language called EXPRESS, standardized EXPRESS schemata for specific business domains, and an interface for accessing data (*STEP Data Access Interface* - SDAI [8]). The other

standardization is the OMG's (*Object Management Group*) CORBA (*Common Object Request Broker Architecture* [12]) approach to distributed, portable, and fault tolerant object computing in heterogeneous environments.

During the last years several efforts have been started to combine both standards in order to achieve interoperable product data management (PDM). Presumably, the most important one is the official ISO mapping of the SDAI to CORBA [9]. But its (native) usage turned out as being inefficient due to too many fine grained objects that lead to *Operation Shipping* instead of *Data Shipping* [17]. Some projects have designed additional protocols (see e.g. [1], [2], or [11]), but those are not 100% standard compliant any more. Next is CORBA's *PDM Enabler Facility* [14] which can be seen as an approach to realize PDM within CORBA environments. It is not explicitly related to STEP or EXPRESS and does therefore not provide access to data modeled by arbitrary EXPRESS schemata. Even the STEP schema for the automotive industry (STEP AP 214 - which is most important for us) is not supported in an appropriate way. In addition, the underlying data model is really fine-grained, too, leading once again to reduced performance.

Thus we can see that the great challenge to realize advanced *Data Shipping* capabilities using CORBA and STEP/EXPRESS still remains open. In this paper we have therefore developed a new approach that combines both standards by offering efficient *Data Shipping* and additional query processing capabilities (which are neither covered by [14], nor by [9], [1], [2] or [11]). It is based on the CORBA *Query Service*. The interface of this service is standardized by OMG, but we have to define data structures and queries that will be used by the service. This step has to be done carefully since CORBA is not primarily designed for efficient data shipping [17]. In addition, implementation effort had to start from scratch: Currently only IBM's Component Broker provides such a service [4], but it is restricted to Windows NT and ad-hoc queries, yet.

Copyright 1999 IEEE. Published in the Proceedings of DASFAA'99, April 1999 in Hsinchu, Taiwan. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

The remaining paper is structured as follows: Section 2 contains a brief introduction to CORBA and its data shipping capabilities, as well as a description of the CORBA *Query Service*. Section 3 contains an outline of STEP. The design of our *Query Service*, including data structures and queries, is presented in Section 4, whereas implementation issues and experiences are discussed in Section 5. Finally, some conclusions and an outlook to future work is given in Section 6.

2. CORBA

In order to cope with the growing need for distributed, portable, fault tolerant and reusable components in heterogeneous environments, the Object Management Group (OMG) has defined the Common Object Request Broker Architecture (CORBA, see [12]) for distributed object computing. The core components of the standard are the *Interface Definition Language* (IDL), used for the declaration of interfaces as well as data structures, and the basic architecture itself. Mappings to the most commonly used programming languages yield to a flexible design of widely available modules with well defined interfaces. In addition, the OMG has defined basic services and facilities. Detailed information is contained in [19], [12] and [13].

2.1 IDL

Any interface or data structure has to be declared using CORBA's object-oriented *Interface Definition Language* (IDL). The system's IDL compiler will then translate these clauses into fragments of the desired programming language, e.g. C, C++ or Java. IDL contains the well known base types like long, char, boolean and float, aggregates like arrays and sequences as well as the struct and interface clause for modeling objects. The latter one is - as already obvious by its name - intended to be used for modeling interfaces of so called CORBA *Services*. These *Services*, simply called *objects*, will be managed by the Object Request Broker (ORB, Section 2.2).

```
enum Category {Compartment, Open_plan};
struct ResData {
    short    day, month, year;
    short    from_station, to_station; // coded
    Category cat_wish;
    boolean  smoker;
};

struct Seat {
    short coach_no, seat_no;
};

interface Reservation_Service {
    Seat ReserveTrain (in long train_no,
                     in ResData data);
};
```

Figure 1: IDL definitions of a reservation service

Data objects may be modeled either by the interface or struct clause. Structs are restricted to a collection of attributes with no support for inheritance or methods, whereas interfaces may contain methods and support even multiple inheritance. Both clauses yield to different runtime processing which is discussed in detail in [17]. A simple example declaring data objects as structs is shown in Figure 1. It also contains one CORBA *Service* called *Reservation_Service* which is declared using the interface clause.

2.2 Architecture of CORBA

The basic component of any CORBA system is the *Object Request Broker Core* (ORB Core). It is responsible for any communication, conversion of data depending on the host architecture (e.g. Little Endian -> Big Endian), registration of CORBA objects declared as interfaces, and the location of these requested CORBA *Services*.

The ORB Core (or just ORB) is often compared to a generic system bus where additional modules may be plugged in. A more functional view is presented in Figure 2. Several components which support the client and/or server are located on top of the core layer.

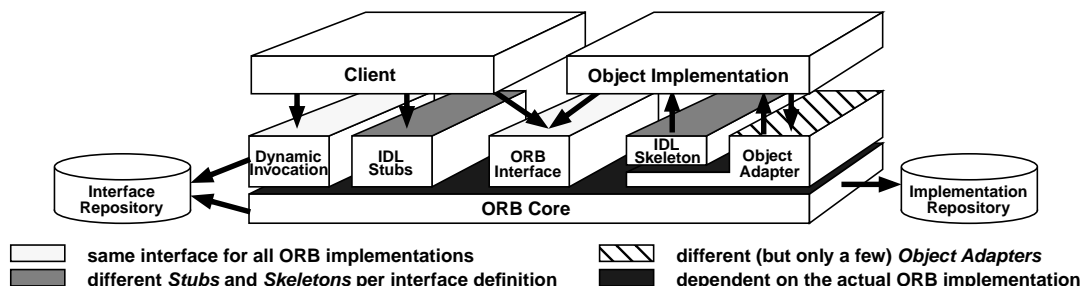


Figure 2: Architecture of CORBA

Clients may call a method of a CORBA *Service* either by using the *IDL Stubs* (built by the system's IDL compiler) or by creating a dynamic request using the *Dynamic Invocation Interface* (DII) that in turn queries the *Interface Repository* in order to forward the method call to the respective *Service*. Last but not least, the *Implementation Repository* is used by the ORB *Core* to store runtime information about available objects. Some of its data is accessible through the *ORB Interface*.

Any *server object* has to be declared by the IDL interface clause. It is embedded into the system through a general *Object Adapter* and a more specific *IDL Skeleton*. Both layers are responsible for object instantiation and initialization (called *object activation*) and the correct propagation of method calls.

Though CORBA is basically designed to support operation shipping and hence objects declared by the interface clause are often described as CORBA *Services*, the interface clause may (in principal) also be used to declare data objects. Nevertheless, [17] shows that real data shipping and caching at the client site can (currently) be realized with the `struct` clause only.

2.3 CORBA Common Object Services (COSS)

Beyond IDL and the basic architecture of CORBA, the OMG has specified some basic *Common Object Services* (COSS). These services should be available in any CORBA environment to ease the development of higher level functionality. E.g., the *Naming Service* can be used to obtain a reference to a component with a well known name, the *Trader Service* to search for a component offering specific operations (similar to yellow pages). The *Transaction* and *Synchronization Service* enable transactional processing; components can register to become part of a running or new transaction. All COSS are defined in [13].

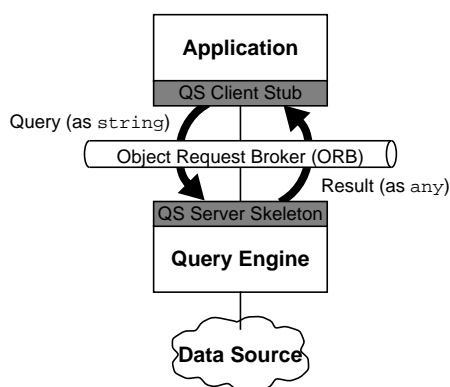


Figure 3: General architecture of a CORBA Query Service

CORBA Query Service. For this paper, the *CORBA Query Service* (QS) is most interesting. It specifies generic interfaces for querying all kind of data sources (not only those managed by database management systems) as well as some rudimentary collections. Its general usage scenario is illustrated in Figure 3: The client sends a query to the server which returns the requested result.

With regard to Relational Database Management Systems (RDBMS), the QS is a CORBA-suited interface similar to the *X/Open Call Level Interface* (CLI), ODBC, or JDBC. Thus RDBMS could be integrated by a simple wrapper (see Figure 4): The client transmits a string containing the SQL query to the QS server and the wrapper accesses e.g. DB/2 through its CLI interface. Afterwards it sends back the resulting (relational) tuples as a sequence of IDL base types (wrapped and shipped by an instance of the IDL any type). Multiple RDBMS could be integrated in a similar way by using DB middleware like IBM Data Joiner [3]. In principal, the same architecture might be deployed for OODBMS (substituting SQL by OQL), but one has to take care of result types (see below).

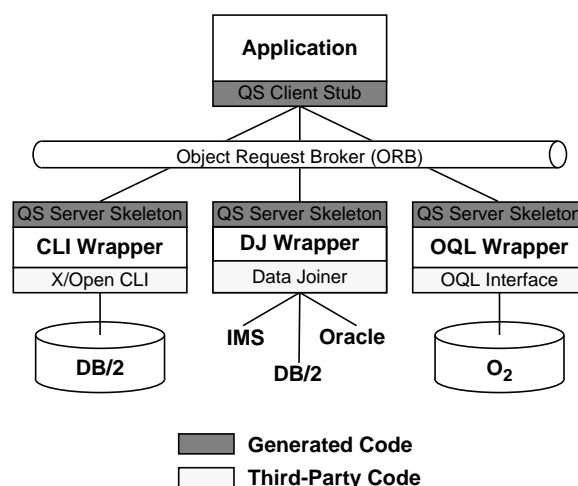


Figure 4: A CORBA Query Service accessing multiple databases

The hierarchy of interfaces of the *Query Service* is presented in Figure 5. The *QueryEvaluator* interface declares methods to execute ad-hoc queries and to obtain associated (that means supported and default) *QueryLanguageTypes* (see below). The *QueryManager* interface declares an additional method for obtaining a *Query* object for a particular query. The query has to be specified as a parameter and cannot be changed afterwards. The resulting query object is associated with its *QueryManager* in order to determine supported *QueryLanguageTypes*. The *Query* interface itself

declares methods to prepare, execute and retrieve the result of a query as well as a status flag. The *Collection* interface declares a simple collection type which is meanwhile substituted by the newer *CORBA Objects Collections Service*. The *Queryable-Collection* interface declares no additional methods.

The QS specification defines two levels of query processing. The first one consists of *QueryEvaluator* and *QueryableCollection*. It realizes query functionality for ad-hoc queries. Its *execute* operation implicitly combines compiling, optimization and execution of the specified query. The second level comprises *QueryEvaluator*, *QueryManager* and *Query*. It offers performance benefits by declaring explicit query objects with *prepare* and *execute* methods. It will be used in the remaining paper. In both levels, the result of a query is returned as an instance of the IDL *any* type. That means any kind of data structure or object can be returned.

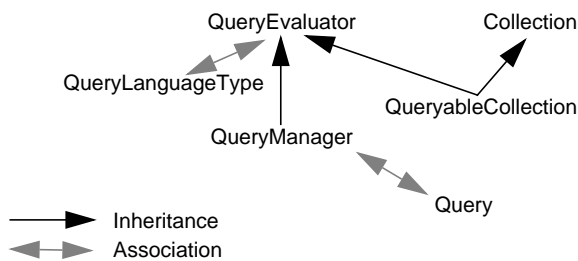


Figure 5: IDL interface hierarchy of the CORBA Query Service

The *QueryLanguageType* actually denotes another hierarchy of (empty) IDL interfaces that is shown in Figure 6. It is used to model the query language type that is supported by a particular QS. The query language type determines syntax and semantics of supported query strings. Currently, an implementation of the QS has to support either *SQL_92Query*, *OQL_93*, *OQL_93Basic*, or a combination of them. In order to declare support of a concrete query language, one has to define a new (empty) IDL interface that inherits from the predefined ones (see Figure 6 and chapter 11 of [13]). Once *OQL* and *SQL* have a suitable intersection, this is expected to become the only supported query language. A discussion of our prototype's query language type is contained in Section 4.2. Please note that access to metadata (except for the retrieval of the query language type) is also (and only) available through this query interface.

The *Query Service* can be used in conjunction with other COSS, e.g. the *Transaction* and *Synchronization Service* to ensure transactional behavior. Therefore the

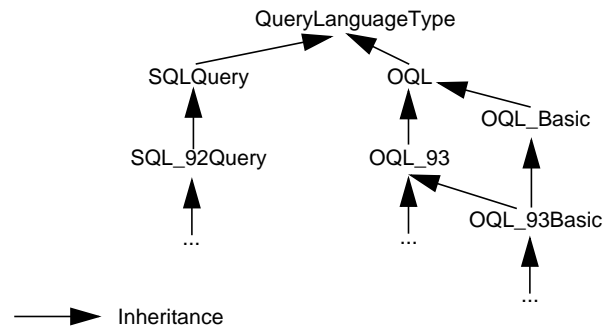


Figure 6: Query language type hierarchy of the CORBA Query Service

QueryManager might be registered by the *Transaction Service* as a transactional resource.

Though the QS specification is really technical, one should remember that the QS defines just a generic query facility (see Figure 3): An IDL string represents the query, an IDL *any* containing the result is returned. The actual query language type and the underlying result type are not fixed. Thus any implementation of a QS has to decide on this.

3. STEP

The *Standard for the Exchange of Product Data* (STEP, ISO 10303, see [6]) has been developed by the *International Organization for Standardization* (ISO) in order to define a common data model and procedures for the exchange of information. It consists of approximately 100 documents (so called parts) that can be partitioned into 4 major categories: description methods, implementation methods, conformance testing methodology and framework, and standardized application data models / schemata (so called *Integrated Resources* and *Application Protocols*). Though the latter topics represent basic concepts of the standard, they are out of scope for this paper and will therefore not be discussed in the remaining sections. The interested reader may refer to [1], [6], [7], [8], and [10].

3.1 Description methods

The description methods include the object-oriented data definition language of STEP, called *EXPRESS*. All information to be processed or to be exchanged has to be modeled using this language. *EXPRESS* provides a rich collection of base types (INTEGER, REAL, NUMBER, BOOLEAN, LOGICAL, STRING, BINARY), supports enumerations and different types of aggregates (ARRAY, BAG, LIST, SET), and enables the definition of objects that use single, multiple or even AND/OR inheritance (the latter

case is used to model objects that are instances of more than one subclass at a time). In addition, SELECT types can be defined for objects that change their base class during runtime, e.g. NUMBER = SELECT (REAL, INTEGER). EXPRESS can be used to define algorithms that are part of integrity constraints, but it does not support the definition of methods or behavior of objects.

Concluding all aspects, EXPRESS can be seen as a modeling language for data structures and corresponding constraints, but it is not a programming language for accessing or creating data.

3.2 Implementation methods

The implementation methods of STEP comprise an ASCII-based file format for data exchange (so called *STEP Physical Files*), and the SDAI for online data access. The SDAI is an object-oriented, navigational interface which is defined in part 22 of ISO 10303 (see [8]). Using the SDAI, data sets may be partitioned into several *Repositories* holding multiple *Models*. This structure is somehow comparable to databases and contained segments/partitions. Each object is part of exactly one *Model* in exactly one *Repository*. Relationships between objects in different *Repositories* or *Models* are supported. Any data processing has to be started by opening a *Session* and *Transaction* as well as the desired *Repositories* and *Models* before the objects can be navigated upon.

Beyond the formal specification in part 22, language mappings to C, C++, Java and the IDL of CORBA are under development for some years now. As we have already mentioned, the binding to CORBA showed to be inefficient. We conclude therefore that the SDAI can not be (directly) mapped to a CORBA component in an appropriate way, but we will show in the remaining paper that a CORBA *Query Service* can be built to access EXPRESS-based data in a similar way.

4. Design of the EXPRESS-based CORBA Query Service

In the previous two sections we have discussed the capabilities and features of the two international standards that form the basis for this paper: CORBA and STEP. Now we will discuss the combination of the two in order to realize an EXPRESS-based *Query Service*.

Considering the CORBA world, we have seen in Section 2.3 that the interface of the service is already fixed, but data structures that will be returned by a query and the query language type itself have to be specified. These issues are addressed in Section 4.1 and Section 4.2.

As far as STEP is concerned, we have to ensure that all EXPRESS-based data can be processed. This is also done by the definition of corresponding data structures in Section 4.1. Though the *Query Service* is not an access interface standardized by STEP, we want to enable interoperability with the SDAI as well as with components reading and writing *STEP Physical Files* (see Section 3.2). Section 4.1 therefore introduces object identifiers that take care of SDAI *Models* (SDAI *Repositories* need no specific mapping since they are treated as regular data sources processed by the *Query Service*). In addition, queries defined in Section 4.2 can also contain the ID of an SDAI *Model*.

4.1 Data structures for EXPRESS-based data

Designing our architecture, we decided to use generic data structures that do not rely on a particular EXPRESS schema. This is based on two requirements: First, the *Query Service* should be able to process all kind of data modeled by an arbitrary EXPRESS schema, without changing or adding any IDL definition (that would lead to code modification within the server). Second, the implementation should be as efficient as possible.

Considering the results of [17], generic data structures appear to be most suitable for shipping the result of a query within CORBA (so called *Data Shipping*). Therefore several issues have to be considered: First of all, we have to verify which kind of data can be referenced by other data (or directly by the user) and therefore needs an identifier. The EXPRESS ENTITY type corresponds more or less to an object. Instances can be retrieved (e.g. by using methods of the SDAI) and they can be referenced by other instances of an ENTITY type. Thus data structures for ENTITY types have to comprise an OID. Taking a closer look at EXPRESS, one can see that values of all other types are associated with a single instance of an ENTITY type. That means that their existence relies on the existence of the “owning” entity; so there is no need for an OID in principal. Nevertheless, considering huge aggregates, it might be more efficient to separate the transmission of aggregates from the transmission of their “owning” entity. Instead, the OID of the aggregate should be sent. If the client needs the aggregate, too, it can use this OID to obtain it afterwards. Therefore aggregates will comprise an OID, too. Nested aggregates belong to the top-level aggregate and have no OID.

Next, we have to map this decision and all EXPRESS types to IDL definitions (see Figure 7). The enumeration `ExpresType` lists all possible EXPRESS types (NUMBER is mapped to REAL). Entities and aggregates (that means all types that have an associated OID) are represented by `EObjOrAggr`. `EUnset` and `EUnknown` are generic tags to indicate uninitialized or unknown

(currently not accessible/computable) values. The union `ExpValue` can be used to represent any kind of EXPRESS data.

An OID is represented by a structure that contains a sequence number (`msl/lsl`), a `flag` indicating the EXPRESS type (“O” for an entity, “A”, “B”, “L” or “S” for arrays, bags, lists or sets), and the model ID. If SDAI-like processing of models is not supported, the latter one should be set to zero. The concatenation of `msl/lsl/modelID` has to be unique.

Entities and aggregates themselves are represented by the `ObjectOrAggr` struct. It contains the `OID`, the type `ID`, and the list of attributes (in case of an entity) or the aggregate’s components. In case of an aggregate, the type `ID` of the underlying base type is specified. Of course, the type `ID` is schema specific. Thus client and server have to make sure that they use the same set of schema information (see discussion of `MetaData` below). Each select query should return an instance of `seqObjectOrAggr` (wrapped by an instance of `any`).

In case of an entity, the order of attributes is specified as follows: Attributes appear in the same order than in the EXPRESS schema, attributes of supertypes are contained first, and supertypes are ordered in the same way than in the supertype clause. If an attribute is inherited through different supertypes, only its first occurrence is considered.

The EXPRESS `SELECT` type is mapped to `SelectType` and `FinalSelectValue`. Since `SELECT` types might be nested, the `SelectType` contains the hierarchy of `SELECT` types (represented as a list of type `IDs`) and the actual underlying (final) value. In principal, the attributes of the `ExpValue` type are already sufficient to represent final `SELECT` values, but we had to define the `FinalSelectValue` type in order to avoid cyclic dependencies within the IDL definitions of `ExpValue` and `SelectType`. Such dependencies are not allowed by the IDL specification (see Figure 7).

Values of an EXPRESS `TYPE` data type (which is more or less an alias) are tagged by the underlying type, e.g. if `MY_INTEGER` is defined as `INTEGER`, then values of `MY_INTEGER` are tagged like values of `INTEGER`.

The structure `MetaData` is used to exchange schema-specific and runtime-specific information. `typeIDs` and `typeName`s describe all known EXPRESS types (base types, user-defined `ENTITY` types, etc); `typeIDs[i]` corresponds to `typeName`s[i]. `msl` and `lsl` can be used by the client to create new `OID`s: The `msl` value is unique for each client, for each new `OID` the `lsl` should be incremented by one (the `lsl` value received denotes the currently highest used value). If SDAI-like processing is supported, `modIDs` and `modName`s contain `IDs` and names of known models.

```
module ExpressQuery {

    interface Express_SQL_Query : CosQuery::SQL92Query {};

    typedef short TypeID;
    typedef sequence<TypeID> seqTypeID;

    struct OID {
        long    lsl;
        short   msl, modID;
        char    flag;
    };

    enum ExpressType {
        EInteger, EReal, EBoolean, ELogical, EString, EBinary,
        EObjOrAggr, ENestedArray, ENestedBag, ENestedList,
        ENestedSet, ESelect, EUnset, EUnknown
    };

    union FinalSelectValue switch (ExpressType) {
        case EInteger:    long    intVal;
        case EReal:       double  realVal;
        case EBoolean:    char    boolVal;
        case ELogical:    char    logVal;
        case EString:     string  stringVal;
        case EObjOrAggr:  OID     oidVal;
    };

    struct SelectType {
        seqTypeID  type;
        FinalSelectValue value;
    };

    union ExpValue switch (ExpressType) {
        case EUnset:    char    unsetVal;
        case EUnknown: char    unknownVal;
        case EInteger:  long    intVal;
        case EReal:     double  realVal;
        case EBoolean:  char    boolVal;
        case ELogical:  char    logVal;
        case EString:   string  stringVal;
        case EBinary:   sequence<octet> binVal;
        case EObjOrAggr:  OID     oidVal;
        case ESelect:   SelectType selVal;
        case ENestedArray: sequence<ExpValue> nestArrayVal;
        case ENestedBag:  sequence<ExpValue> nestBagVal;
        case ENestedList: sequence<ExpValue> nestListVal;
        case ENestedSet:  sequence<ExpValue> nestSetVal;
    };

    struct ObjectOrAggr {
        OID      oid;
        TypeID    type;
        sequence<ExpValue> values;
    };

    typedef sequence<ObjectOrAggr> seqObjectOrAggr;

    struct MetaData {
        short    msl;
        long     lsl;
        sequence<short> modIDs;
        sequence<string> modNames;
        sequence<TypeID> typeIDs;
        sequence<string> typeName;
    };
}; // end of module ExpressQuery
```

Figure 7: IDL data structures for shipping of EXPRESS-based data

4.2 Supported query language type

Whereas the specification of IDL definitions for EXPRESS-based data could be done without any workaround, finding an appropriate query language type turns out as a hard problem. Though there exist several mappings from EXPRESS to relational or object-oriented database management systems (see e.g. [10]), all mappings lead to complex queries. In addition, queries rely on a specific mapping for a specific schema/system combination. Thus we assume that pure OQL or SQL92 is not sufficient.

A mapping of EXPRESS to SQL3 is currently investigated as part of another project. First results are quite promising. We therefore expect that SQL3 will be a suitable query language. In addition, the current *Query Service* specification states already the goal of having a single query language that reflects both OQL and SQL. Thus we assume that SQL3 will be the solution and next generation *Query Services* have to support SQL_3 instead of SQL_92Query, OQL_93, or OQL_93Basic.

In the meantime we have defined some rudimentary queries to achieve access capabilities similar to the SDAI (see Table 1). They do not offer value-specific query functionality, but they are sufficient for the prototype described in Section 5. Since all queries have an SQL-like syntax, we declared a new query language type `Express_SQL_Query` that inherits from `SQL_92Query` (see Section 2.3).

Category	Supported Queries
Data Retrieval	SELECT * FROM Repository SELECT * FROM Repository WHERE typeId=#1 SELECT * FROM Repository WHERE typeId=#1 OR SUBTYPE SELECT * FROM Repository WHERE oid=#1 SELECT * FROM Repository WHERE modID=#1 SELECT * FROM Repository WHERE modID=#1 AND typeID=#2 SELECT * FROM Repository WHERE modID=#1 AND (typeID=#2 OR SUBTYPE)
Metadata Retrieval	SELECT * FROM MetaData
Data Modification	INSERT #1 INTO Repository UPDATE #1 IN Repository DELETE FROM Repository WHERE oid=#1 DELETE Model FROM Repository WHERE modelID=#1
Metadata Modification	CREATE Model #1 RENAME Model TO #1 WHERE modelID=#2

Table 1: Supported queries

5. Prototype implementation

The implementation of our prototype has to fulfill several requirements. First, the client should be written in Java and serve as a new *Data Module* for our Java-SDAI architecture (see [18]). That means SDAI-like processing including *SDAI Models* has to be supported. This is ensured by the queries defined in Section 4.2. Second, the server should be written in C++ to guarantee sufficient performance. Third, all components should be independent of the used CORBA system. More precisely, they should run with (at least) IONA's Orbix 2.3 [5] and OOC's ORBacus 3.0 [15]. Fourth, we want to implement two *Query Services*, one reading *STEP Physical Files*, and another one that stores data in GDBM (*GNU Database Manager*) files. Processing at the client site should be the same in both cases due to the transparency provided by CORBA and the QS. Fifth, all components should run on different platforms/operating systems, at least Solaris/Sparc, HP-UX and Linux/X86 (the latter one is unfortunately not supported by Orbix). The resulting architecture is illustrated in Figure 8. Grey boxes symbolize code generated by an IDL compiler.

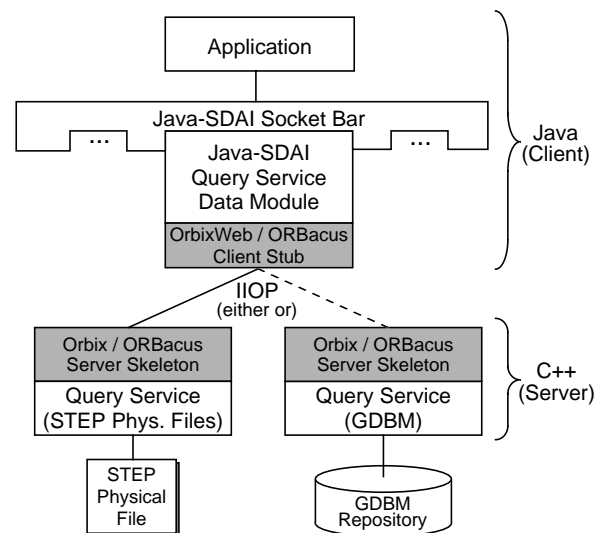


Figure 8: Architecture of the prototype

Since this paper focuses on the *Query Service*, development issues of the client-site *Socket Bar* and the new *Java SDAI Data Module* are out of scope for this paper (see e.g. [18]). The application is a simple test and proof-of-concept routine that accesses scalable binary trees. It is the same as already presented in [17]. Whereas the architecture presented there is based upon a proprietary interface, the architecture discussed in this paper utilizes the standardized interface of the QS that in addition enables the integration of database management systems and query capabilities.

5.1 Server development

Due to a missing query language and query engine for EXPRESS-based data (see Section 4.2), each server has either to contain its own proprietary query engine or specific query objects for each query listed in Table 1. We decided to use the latter solution in order to reduce unnecessary implementation effort. Thus both server contain registered CORBA objects implementing the IDL interface *Query-Manager* (one per GDBM Repository and one per STEP Physical File) and corresponding *Query* objects. Each of those *Query* objects provides a C++ method that realizes the corresponding query's functionality; it is similar to a given query execution plan that evaluates the query.

The GDBM-based server stores serialized versions of the data structures defined in Figure 7. It uses one GDBM file per *SDAI Model* and separate directories per *SDAI Repository*. In order to support extent queries (all objects of a particular type), it keeps additional lists per typeID that contain the OIDs of all stored objects of that type. The server contains no caching or prefetching mechanisms except for the internal GDBM cache.

The server of the *Query Service* for *STEP Physical File* processes the complete file at once and creates corresponding main memory objects. All queries are executed on these objects. Partial file processing is expected to be inefficient since the complete file has to be read for almost each query execution (a *STEP Physical File* holds a number of related data objects, as e.g. an *SDAI Repository* or *Model*).

5.2 Implementation experiences

Implementing both services, the development of code that will run with different CORBA systems turns out as the hardest problem. Some issues are system-specific, others are based on missing standardization:

- CORBA 2.0 and its *Basic Object Adapter* (BOA) do neither standardize names of skeleton classes, nor the complete signature of skeleton methods generated for IDL declarations. Some systems (like Orbix) use an additional context parameter, others (like ORBacus) not. All systems use different class names. We hope that this problem will vanish by the availability of CORBA 2.2 compliant systems which contain a *Portable Object Adapter* (POA), which will in addition standardize initialization of server processes.
- How to throw system exceptions is also not standardized.
- Some C++ compiler support namespaces/inner classes, others not. In the latter case, one has to use concatenated class names as a workaround, e.g. CORBA_Any instead of CORBA::Any.
- User manuals do often not illustrate how to implement CORBA compliant code. Instead, they put special emphasis

on proprietary features or use a proprietary syntax for generated code. E.g., Orbix uses the C++ type *any* for generated stubs and skeletons. The CORBA compliant syntax would be *CORBA::Any*. Compliant syntax will run with Orbix due to a typedef, but the application programmer never gets a hint that the generated code is not compliant. If he uses the Orbix syntax for his own code, the application is not portable among different CORBA systems.

A more general problem with CORBA's IDL occurred regarding to the definition of union *ExpValue* (see Figure 7). In case of nested aggregates, it contains a recursive definition: The underlying type is *sequence<ExpValue>*, which is a so-called *unnamed sequence type*. Unfortunately language mappings do not standardize names for these types, and, in addition, a named sequence can not be used here since *ExpValue* is not known before its own definition (forward declarations of union types are not supported by IDL). Beyond non-standard names, another problem occurs: *sequence<ExpValue>* is used once again in the definition of *ObjectOrAggr*, but, due to scoping rules (see clause 3.13 of [12]), both types are not equivalent. The first one resolves to *ExpressQuery::ExpValue::sequence<ExpValue>*, the second to *ExpressQuery::ObjectOrAggr::sequence<ExpValue>*. In principal, this could be solved by the definition of a named sequence that is used within *ObjectOrAggr*:

```
typedef ExpValue::sequence<ExpValue> seqExpValue.
```

In practice, none of our CORBA systems accept this statement: Orbix generates the correct namespaces (resp. inner classes) as part of *Stubs* and *Skeletons*, but doesn't allow the typedef within the IDL definitions, ORBacus considers no nested namespaces for unnamed sequences at all.

5.3 Sample application and performance issues

The prototype has been tested with the sample application already used in [17]. It comprises the generation and retrieval of scalable binary trees that are comparable to parts explosions in bill-of-material processing, thus offering a somehow realistic test scenario. Each node of a tree also references an array of integer values in order to test aggregates, too. To simplify measurements, the array contains a single element only.

All data is stored in exactly one *SDAI Repository* which contains one *SDAI Model* for each binary tree. Since the creation of data is not mission critical, we will only present measurements regarding to data retrieval. In order to test prefetching capabilities of the *Data Module*'s buffer, we have defined four different access strategies (see Table 2). The search operation touches approx. 5 to 10 nodes, depending

on the size of the tree. For each strategy separate test series have been performed that access a separate binary tree in its own *SDAI Model*. In a first phase (cold run) needed objects are requested from the server. If prefetching is used in this phase, all objects of the *SDAI Model* are transferred at once, but aggregates are still shipped separately on demand. After the cold run the same operation is performed once again, but all objects are already located in the client site buffer. This phase is therefore called hot run. It is repeated 5 times to avoid undesired runtime effects.

Access Strategy	Operation	Prefetching of the Entire SDAI Model
1	full scan	no
2	full scan	yes
3	search for a single node	no
4	search for a single node	yes

Table 2: Access strategies of the test application

All measurements have been performed using the GDBM-based server. Both, client and server, have been run on a PC with a Pentium 133 processor and 64 MBytes of main memory. The operating system is S.u.S.E. Linux 5.2 (kernel 2.0.33, gcc 2.7.2.1 and ORBacus 3.0). Additional analysis comprising multiple platforms, CORBA systems and different *Data Modules* will be published in [18].

A comparison of cold runs illustrating the different access strategies is contained in Figure 9. The upper two curves refer to a scan operation that requires all objects and aggregates at the client site. As expected, prefetching and bulk transfer (of objects) yield to some performance benefits since ORB to ORB communication is reduced a lot. The lower two curves (one is almost hidden by the x axis) refer to a search operation. Strategy 4 is worse since a lot of unnecessary objects have been transferred and cached, but only 1% of them are needed. A more interesting aspect is given by the huge runtime clash between strategy 2 and 4. Strategy 4 accesses only a few objects, but it initially

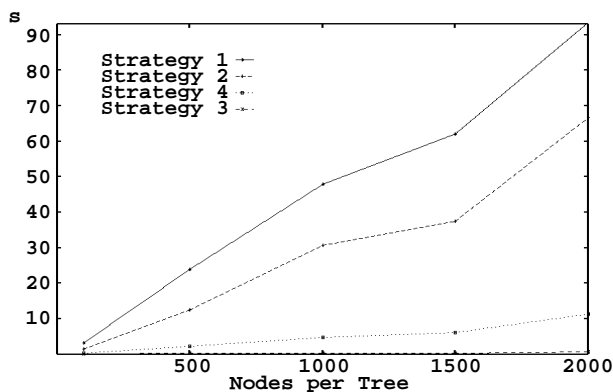


Figure 9: Cold runs using JDK 1.1.6 with JIT

requests bulk transfer of all objects. Strategy 2 will additionally load all aggregates on demand (using separate requests). That means the difference between both curves directly illustrates the time needed for aggregate transfer. Therefore we conclude that prefetching by bulk transfer should be configured (for objects and aggregates) as soon as access to a lot of objects can be expected. The concrete percentage indicating which strategy performs better will be determined by further measurements. Please note that our design enables changes of the prefetching policy on the fly!

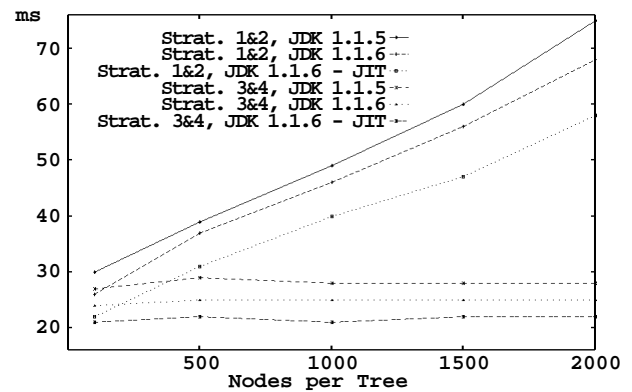


Figure 10: Hot runs

The results of several hot runs are illustrated in Figure 10. Since prefetching does not influence hot runs, we can only distinguish between scan operations (strategy 1 and 2) and search operations (strategy 3 and 4). In addition, the performance gain between different JDK versions and the usage of a *Just-In-Time Compiler* (JIT) can be determined. As one can see, time needed for the search operation is quite constant for all sizes of trees. Most of the time is needed for initial transaction processing and time actually needed for object access is too short to be measured. Time needed for scans scales linear and shows acceptable performance.

On the other hand, performance of cold runs is somehow difficult to evaluate. Strategy 3 is fast whereas strategy 1 is definitely too slow. That means one has to decide on an appropriate prefetching strategy. Nevertheless, we have also to consider the optimization of general design decisions. Taking a closer look at the implementation, we already figured out two aspects that have a negative impact: First, IDL enumeration types are mapped to the C++ `int` type. That means each union discriminator needs 4 bytes of memory (and communication bandwidth), though e.g. the enumeration `ExpressType` has only 14 enumerators. Second, using CORBA 2.0 as the basis, insertion into an instance of any leads to a deep copy of the data to be inserted. Thus all query results are unnecessarily copied.

The first issue could be resolved by using a `char` as the union discriminator. But this is somehow dirty since the

association of `char` values with EXPRESS types can not be specified in IDL (only as a comment). In principal, the second problem has already vanished by the introduction of CORBA 2.1. The specification defines two insertion operators for any types: One for deep copies, one for references. But there are no explicitly CORBA 2.1 or 2.2 compliant products available. Though some features are already realized by some CORBA system, one can not rely on this: ORBacus generates both operators, Orbix only the one for deep copies.

6. Conclusions and outlook

In this paper we have presented the design and implementation of a standardized component for accessing STEP-based data: A CORBA *Query Service* (QS). In general, The QS offers query facilities to CORBA clients, comparable to X/Open CLI, ODBC or JDBC. Considering the concrete design presented here, it realizes a new approach to integrate the CORBA and STEP standards that outperforms the current binding (see [9]). Once the work of our colleagues to harmonize data representations between EXPRESS and SQL3 is done, and the OMG has adopted SQL3 as the standard query language type for its *Query Service* specification, our architecture is capable to combine even the three standards CORBA, STEP, and SQL3 (see Section 4). Furthermore, a commercial (SQL3) database engine could be used to realize the query functionality of the QS (as indicated by Figure 4).

Implementing the *Query Service*, it turned out that the development of server code that is portable among different CORBA systems is still a hard problem. The reason for this is not only based on specific product features, but also on still missing standardization (see Section 5.1).

Data structures for EXPRESS-based data as defined here (see Section 4.1) can also be used in a broader setting, i.e., to guarantee interoperability between existing legacy systems, data sources and new applications. Our data structure approach will therefore be used to model a new integration architecture within DaimlerChrysler that comprises well-established PDM and CAD components as well as new Java applications written on top of an integrated API (see [16]).

Last but not least, we should stress that the *Query Service* presented here is not limited to access of data which is modeled by a schema standardized by STEP. It can process all kind of data described by an EXPRESS schema. Please note that EXPRESS is a generally available modeling language that is especially often used in engineering domains and technical standardization like e.g. *Geographic Information Systems* (GIS).

Acknowledgements

We would like to thank our colleagues Roland Nagel and Ulrich Schäfer for their help and the implementation of the *Query Service* for *STEP Physical Files*. We are also greatly in debt to Toni Maurer for performing the measurements and to Günter Sauter for fruitful discussions.

References

- [1] M. Hardwick, D. Spooner, T. Rando, K.C. Morris: *Sharing Manufacturing Information in Virtual Enterprises*, Communications of the ACM, February 1996.
- [2] M. Hardwick, D.L. Spooner, T. Rando, K.C. Morris: *Data Protocols for the Industrial Virtual Enterprise*, IEEE Journal for Internet Computing, Vol. 1, No. 1, <http://computer.org/internet/ic1997/w1toc.htm>, 1997.
- [3] IBM Co.: *Data Joiner: Administrator Guide and Application Programming*, IBM Co., San Jose, 1997.
- [4] IBM Co.: *IBM Component Broker — Advanced Programming Guide*, Release 1.3, Third Edition, July 1998.
- [5] IONA Technologies Ltd.: *Orbix Programming Guide / Reference Guide*, Release 2.3, Dublin, Ireland, 1998.
- [6] ISO IS 10303-1 TC184/SC4: *Product Data Representation and Exchange — Part 1: Overview and Fundamental Principles*, International Standard, 1994.
- [7] ISO IS 10303-11 TC184/SC4: *Product Data Representation and Exchange — Part 11: The EXPRESS Language Reference Manual*, International Standard, 1994.
- [8] ISO FDIS 10303-22 TC184/SC4: *Product Data Representation and Exchange — Part 22: Standard Data Access Interface*, Final Draft International Standard, July 1998.
- [9] ISO DIS 10303-26 TC184/SC4: *Product Data Representation and Exchange — Part 26: Interface Definition Language Binding to the Standard Data Access Interface*, Draft International Standard, July 1998.
- [10] D. Loffredo: *Efficient Database Implementation of EXPRESS Information Models*, Ph.D. Thesis, Rensselaer Polytechnic Institute, Troy, New York, May 1998.
- [11] NIIIP Consortium: *National Industrial Information Infrastructure Protocols*, <http://www.niiip.org/>, 1998.
- [12] Object Management Group: *The Common Object Request Broker Architecture: Architecture and Specification*, Revision 2.2, <http://www.omg.org/corba/corbiiop.htm>, Upd. February 1998.
- [13] Object Management Group: *The Common Object Request Broker Architecture: Common Object Services Specification*, <http://www.omg.org/corba/sectrans.htm>, Upd. July 1998.
- [14] Object Management Group: *PDM Enabler Specification*, OMG document mfg/98-01-01, http://www.omg.org/library/schedule/Technology_Adoptions.htm, Adopted July 1998.
- [15] Object-Oriented Concepts, Inc.: *ORBacus Documentation*, Release 3.0, MA, USA, <http://www.ooc.com/>, 1998.
- [16] St. Sarstedt, G. Sauter, J. Sellentin, B. Mitschang: *Integration Concepts for Heterogeneous Application Systems at Daimler-Chrysler based on International Standards* (in German), in: Proc. of 8th GI-Fachtagung BTW 99, Freiburg (Germany), March 1999.
- [17] J. Sellentin, B. Mitschang: *Data-Intensive Intra- & Internet Applications — Experiences Using Java and CORBA in the World Wide Web*, in: Proceedings of the 14th IEEE International Conference on Data Engineering (ICDE), Orlando, Florida, 1998.
- [18] J. Sellentin, B. Mitschang: *Using the SDAI Socket Bar for the Evaluation of Different Data Shipping Strategies*, Internal Report, Subject to Further Publication, 1999.
- [19] J. Siegel: *CORBA: Fundamentals and Programming*, Jon Wiley & Sons, 1996.