In: Proc. 25th Int.VLDB Conf., Edinburgh, 1999, pp. 575-586.

Generating Call-Level Interfaces for Advanced Database Application Programming[1]

U. Nink, T. Härder, N. Ritter

University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
e-mail: nink/haerder/ritter@informatik.uni-kl.de

Abstract

It has always been a hard problem to provide application programming interfaces (API) for database systems without sacrificing some advantages of either the database management system or of the programming languages. Various approaches have been proposed. We discuss APIs with respect to SQL3 and its object-relational extensions as well as to object-oriented programming languages. We argue that generated call-level interfaces (CLI) are better suited than classical CLIs and language embeddings to couple database languages to object-oriented programming languages. Profiting from code generation and early binding of type information, generated CLIs improve the pros of embeddings while obviating the cons of classical CLIs. We propose an architecture for generated CLIs consisting of a cache module, a generated run-time system, and a compiler that generates parts of the (generated) CLI. The partial generation is specified using a configuration language to describe application-specific early binding of type information corresponding to data models, schemas, and queries. With this approach, we can control the sharing of database type information for application programs as well as the deferrable adaptation of applications to different needs by extending interfaces and by replacing implementations.

1 Introduction

Database management systems (DBMS) store and manage large sets of shared data whereas application programs perform the data processing tasks, e. g., to run the business of a company. Often, these programs are written in various programming languages (PLs) embodying different type systems. Thus, DBMSs should be "multi-lingual" to serve the application requests. This is typically achieved by providing a DBMS and its database language (DBL), like SQL2, with an own type system. To access the database (DB), a DBL/PL-coupling called database API (DB-API or API for short) is required.

1.1 DBL/PL-Couplings

The main problem of coupling DBLs and PLs is the impedance mismatch between them, resulting from differences in the type systems or programming models. While, for example, relational DBMSs offer a quite simple and flat data model, programming languages provide many helpful type constructors for the design of complex information models. Furthermore, relational DBMSs support set-oriented, declarative queries (n-set-oriented queries), while PLs are typically navigational, that is, the programmer has to manually follow links between single objects or iterate collections of objects.[2]

Another aspect is that software technology is changing rapidly. Thus, for APIs it is important to exploit approved and stable base concepts, which, additionally, are flexible enough to be adapted to a changing system environment. The paradigm of object-orientation comprises a set of concepts fulfilling these requirements. It has not only considerably influenced application development, but it also forms "a new great wave in the database ocean" - the object-relational wave [23]. What can be observed here is, as the data models are coming closer, the differences seem to vanish more and more. The most important concepts on both sides are references and abstract data types. References allow, similar to pointers, to directly model (n:m)-relationships between object types. Thus, complex object support is also needed at the API. Abstract data types (ADT) allow to define interfaces and to hide their implementations; modeling of ADTs is done with user-defined types (UDT) in object-relational DBMSs (ORDBMS) and with classes and interfaces in OOPLs. ADTs allow for the extension of the type system without extending the underlying grammar. Thus, many needed extensions for a coupling can be implemented through ADTs and obviate grammar extensions.

1. This work has been supported by the German Science Foundation (DFG) as part of the Sonderforschungsbereich (SFB) 501 "Development of Large Systems with Generic Methods".

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999

2. Extensions in OOPLs support a restricted form of querying (evaluation of simple search arguments on collections: 1-set-oriented queries). Nevertheless, applications still tend to be navigational.

Different couplings have been proposed in the past [9, 13, 14, 15]. Programmers seem to prefer simple interfaces based on well-known concepts in the context of application programming; CLIs, e. g. JDBC [5, 7], have always been accepted best, since they can be combined with the use of standard compilers. API developers, on the other hand, seem to prefer solutions that lead to more elegant interfaces (embedded DBLs, e. g. eSQL or SQLJ [19]); most often precompilers or extended compilers were used to reach this goal.

In order to access the DB via the API, some preparation tasks are needed; each DB statement has to be compiled, optimized, and bound to the data structures (as described in the DB schema). The pure CLI approach essentially considers the DB statement as a string to be passed to the DBMS at run time. Therefore, (most of) the tasks preparing the DB access are to be deferred until run time. Hence, DB access involves additional overhead burdening the response time. Since binding of DB operations (to DB schema information) is "late", DB schema changes can be adjusted until the latest time possible. Hence, data independence is preserved to a maximum degree. In contrast, embedded DBLs shift all preparation tasks to compile time which improves run-time performance and enables more choices to react to errors. On the other hand, early binding makes compiled programs depending on DB schema changes. Hence, such an approach increases data dependence.

1.2 Our Goal

We want to show that by following a generative approach, we are able to 'equip' CLIs with the advantages of embeddings. Thus, we can provide APIs which combine the best of both worlds, i. e., are efficient, have strong typing and early error handling, as well as a high degree of data independence. These APIs that optimally couple DBLs like SQL3 [10, 11] to OOPLs like Java [1] or C++ [21] are *configurable* specializations of CLIs, named ***generated call-level interfaces***.

Such a generative approach allows to properly adapt the API to the application's needs. We will see that especially the selection of suitable binding times (early, late) for interfaces and implementations of API functions and the choice of an adequate pointer swizzling strategy for DB objects cached at the client side are crucial issues in that concern.

In the following, we will first introduce CLIs before we sketch a running example to explain the concepts. Afterwards, the mapping of SQL's data model into data models of OOPLs is discussed. We contrast late binding to early binding and introduce *virtual late binding* as an interesting compromise. Furthermore, the integration of the different programming models on both sides of the API is explored. Then, we introduce a configuration language that allows to specify the adaptation of the API to different needs. Finally, we step through a sample program again contrasting binding times.

2 Generated Call-Level Interfaces

In this section, we want to detail the notion of generated CLIs and introduce our approach. The first subsection identifies important concepts to improve CLIs and defines generated CLIs. Afterwards, we discuss our generative approach, especially by emphasizing the configurability of the API.

2.1 Preconditions and Definition

Many advantages of language extensions like embeddings can also be achieved for CLIs. But, the distance between the concepts of the database language and those of the host language is an important factor for a successful coupling. The following concepts of advanced database languages narrow the gap to programming languages.

- An extensible type system exists for data modeling (collections, user-defined types, and user-defined functions).
- A surrogate concept is the basis for identification and referencing of database objects.
- References between database objects (often as set-valued reference attributes) allow for the direct modeling of (n:m)-relationships.

Regarding OOPLs, we identify the following concepts that allow to improve CLIs.

- Extensible type systems support the definition of user-defined ADTs through interfaces and classes that can be used very similar to built-in data types. Thus, database objects can be treated like programming language objects.
- Encapsulation by using interfaces allows for the replacement of underlying classes and, in consequence, the choice between equivalent implementations with different characteristics.
- Polymorphism and subtyping provide a compromise between pure early and pure late binding and, therefore, open a spectrum of gradual early binding which we also call polymorphic binding.

Code generation is needed to introduce early binding of type information and especially, the early binding of application developer knowledge to certain type information. This leads us to the following definition [16].

Definition: *A generated or early-bound call-level interface (gCLI) is a CLI introducing early binding of type information usually by code generation at compilation time of schemas, queries or applications.*

We propose a specialized compiler to support the provision of early bindings and the necessary code generation. But, in contrast to embeddings, such a CLI compiler is decoupled from the OOPL compiler in order to reduce dependencies. The CLI compiler is based on the database language, that is, it extends the database language compiler (like storage structure languages as part of database languages) or analyzes the database language compiler's output (meta-data, dictionary data).

2.2 Architectural Overview

Figure 1 gives an architectural overview of our approach. At the left-hand side, we see the components needed during run time of an application program: the application program itself, a generated run-time system (gRTS), a cache module, as well as the DBS.

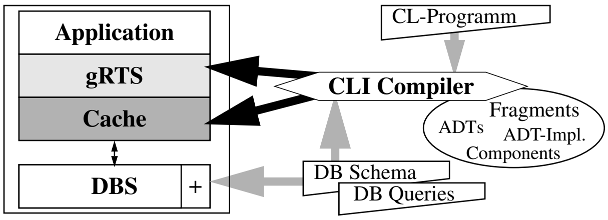

Figure 1: Architectural Overview

Location transparency for access to database objects through the client is provided by the cache module. It supports communication with the DBS, abstraction of object mapping, and (set-oriented) access to DB objects.

Operations of the API are implemented by the gRTS. It contains the operational semantics of the API and provides late-bound and, optionally, early-bound interfaces implemented on top of the cache module. By the gRTS, application programmers get interfaces for database access and session control (transactions). In addition, they can use cursors or iterators to work with the instances. A dictionary provides meta-data access and may be accessible by the programmer, too.

The CLI compiler (see figure 1, right-hand side) produces fragments of the gRTS and the cache module. It performs early binding of type information and allows for the replacement of equivalent (early-bound) implementations (e. g., classes that implement interfaces). In the following, we discuss the task of the CLI compiler in more detail.

2.3 The CLI Compiler

Specifications of the application programmer in a configuration language resulting in a configuration program are analyzed by the CLI compiler to produce early bindings for type information specific to schemas and queries. The generated code extends the core run-time system. As figure 1 shows, the CLI compiler may choose from a pool of code fragments, interfaces, implementations, and ready-to-use components. In addition, it refers to meta-data (DB schema, DB queries) in order to generate type- and query-specific functions. The '+' in figure 1 represents extensions that are needed to manage configuration data. In the following, we discuss the influences of the CLI compiler on cache module and gRTS. Let us start with the cache module.

Access and Object Mapping: These methods support object access based on database object identifiers (OID) and corresponding local identifiers (LID) in virtual memory (pointer, reference) [22] as well as the mapping between OIDs and LIDs. Access through OIDs is location transparent but not as efficient as access through LIDs which omits residency checks and/or reservation checks of referenced objects. In order to speed up access to objects tailored implementations of operations in the run-time system may be generated by the CLI compiler.

The following components (of the cache module) can optionally be configured. To support application-oriented prefetching and pointer swizzling [12], specific control mechanisms are implemented in the run-time system while the basic mechanisms belong to the cache module. Therefore, the following two components cross borders.

Prefetching: If the application knows in advance which objects are needed in the future when calling certain methods of the API, it is useful to early bind this knowledge. Thus, this knowledge can effectively be exploited, to prefetch needed objects in a single step. The CLI compiler can translate this knowledge into additional code for method bodies calling corresponding fetch operations.

Pointer Swizzling: If the application knows in advance, which references between objects are often dereferenced (say more than 2 times), it is useful to early bind this knowledge in order to trigger reference transformation (pointer swizzling [22]). Since dereferencing has to know if references are swizzled or not, a swizzling check is needed, which is also called *lazy-if*. Similar to location transparency we provide reference transparency, which allows for automatic dereferencing of OIDs and LIDs. Again, the CLI compiler may produce efficient implementations that omit the swizzling check in the run-time system.

Before listing the gRTS functions, which can be adapted via the CLI compiler, we want to mention that the gRTS is divided into an external layer and an internal layer. Application programmers use the external layer while API implementers use the internal layer to build the external layer. Each layer is divided into late-bound and early-bound interfaces. At each layer, the decision which kind of interface to use is important for the implementation of the next higher layer (external layer or application, respectively) because of the different syntax. The CLI compiler can either let interfaces reuse existing generic (late-bound) implementations or generate more efficient implementations that use generated (early-bound) interfaces. The following gRTS components can be configured.

Session Control: Local savepoints on the client including all cached database objects and current cursor states are based on the conversion of objects into byte streams and vice versa. This conversion may either occur in a generic method or using generated conversion methods for the various object types.

Database Access: In addition to the generic query class "SQL.gCLI.Query" that can handle query results of arbitrary select statements, each query may be represented by a generated specialized query class "SQL.gCLI.Generated.Query<name>"[3]. An easy thing to do is the early binding of column names of the result set. More work has to be done in order to early bind complex-object structures. We come back to this point later.

3. Queries must be named in this case. The name may be specified in the configuration program.

Result Construction: Also based on conversion of byte streams and virtual memory objects, result construction, too, either occurs in a generic way or has to take replaceable storage structures for objects, indices, and collections into account.

Instances: Besides replaceable storage structures for instance collections, attribute collections, and set-valued attributes the CLI compiler can optionally configure type specific indices for such collections. The CLI compiler produces corresponding code that, for instance, knows if an index is available or not for the current operation.

Dictionary: The dictionary itself is rather generic in nature. But since our object types provide (replicated) methods for meta-data access, these methods can be optimized. Instead of asking the dictionary in the body of the method, it is possible to hardwire known meta-data in advance (like the name of an object type).

Cursor/Iterator: In our architecture this is the most interesting module for the CLI compiler. The CLI compiler can generate statements containing meta-data as constants into method bodies. Attribute access, for instance, can be accelerated by such an optimization since it saves the dictionary lookup at run time. Other configurations may affect prefetching and pointer swizzling. Since method calls trigger prefetching actions or transformation actions, the CLI compiler can also generate code into the method bodies calling predefined or generated actions.

The outline of our approach given in this section is meant to prepare the following discussions. We will start these discussions by introducing a simple application scenario, which will serve as a running example.

3 Example

Developed for the evaluation of ODBMSs the OO7 benchmark [4] is also a candidate for discussing ORDBMSs. Since we do not deal with measurements here, but concentrate on concepts, we narrow our view to a small part of the benchmark which we name *MicroOO7*.

The OO7 benchmark defines a database schema with several entity types. The assembly hierarchy, which we skip here, manifests assembling of composite parts (`CompositePart`) given by a design library.

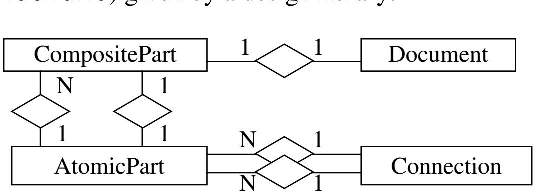

Figure 2: MicroOO7 Schema (ER diagram)

Each `CompositePart` is associated with a describing `Document` and contains a set of `AtomicParts` that are interrelated by explicit `Connection` instances (see figure 2). A CompositePart has exactly one such Document, and each Document belongs to exactly one CompositePart. A CompositePart has a number of AtomicParts (a benchmark parameter), one of which is marked as the root. Each AtomicPart is connected with a fixed number of AtomicParts (also a benchmark parameter). Each such link is represented by a Connection that carries additional attributes.

Figure 3 shows how a complex object of the OO7 benchmark might look like in the database. The entities are represented using row types whereas relationships are modeled by reference attributes. Thus, the instantiated vertices become references in an SQL3 database[4].

We use the MicroOO7 schema as a sample basis for the discussion of API functions; the database excerpt depicted in figure 3 will be used to explain query evaluations.

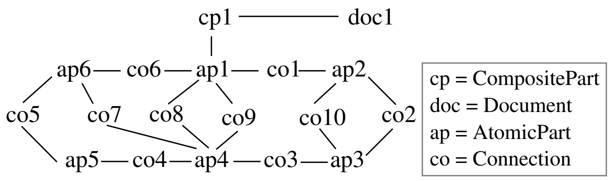

Figure 3: MicroOO7 Complex Object

4 Mapping SQL's Data Model

Now we discuss general aspects of the ADT concept that is found in most advanced data models. Afterwards, we discuss the mapping of SQL concepts.

4.1 General Aspects of ADTs

The concept of ADTs is very important for the DBL/PL-coupling because of its strong encapsulation. ADTs define interfaces which may have several (replaceable) implementations. In OOPLs ADTs are implemented using classes. Some OOPLs like Java allow to explicitly distinguish between interfaces and implementations (interfaces and classes). In order to exploit the ADT concept for mapping, we can to distinguish three cases. First, a seamless coupling can be reached if the concept of ADTs is similar in DBL and PL; thus, DB-ADTs can be mapped to PL-ADTs. Second (sets of) DB-operations are mapped to PL-ADTs. Third, DB-types are mapped to PL-ADTs; for example, for row types the mapping may specify column or attribute access methods with different degrees of early binding for method signatures and their implementations.

4.1.1 Interfaces of ADTs

In the following, we discuss different degrees of early binding for interfaces. Operations or methods constitute the interface of a class (including observer functions and mutator functions for attributes). Thus, the binding of an interface may be reduced to binding corresponding operations. To explain the main idea, we concentrate on attribute read access for row types. Starting discussions with generic solutions independent of row type and attribute we proceed with type-specific and attribute-specific solutions.

4. Due to simplicity, we omit corresponding SQL3 data definition statements. An intuitive understanding of the schema structures is sufficient to follow the discussions in the subsequent sections.

A generic mapping defines a signature for a method GetAttribute of the class Row that is capable of reading arbitrary attributes of rows of arbitrary row types. Below we give the definition of the signature and the usage of the method.

```
// as method of class Row
Attr GetAttribute(String rowtype,
String attribute, String attrtype)
{...}
// usage:
value = comp_part.GetAttribute(
"CompositePart", "buildDate", "Integer");
```

The result value is of type Attr, which is an attribute container for values of all possible attribute types. The current row comp_part is of type Row and allows to call the method GetAttribute by using the dot-notation. The first parameter rowtype of type String contains the name of the row type of the current row. The second parameter attribute of type String specifies the name of the wanted attribute. The third parameter attrtype of type String is to specify the name of the expected attribute type.

A disadvantage of this approach is that the correctness of the parameter values can only be determined at run time. Another disadvantage is that later calls of type-specific operations on the attribute value demand for explicit type casting.

With the following signature more type information is bound early. Two differences are obvious. First, instead of specifying the attribute type by a string, an output parameter value of the attribute's type (Integer) is used. Second, void indicates that no return value is delivered.

```
// as method of class Row
void GetAttribute(String rowtype,
String attribute, Integer value)
{...}
// usage:
comp_part.GetAttribute("CompositePart",
"buildDate", value);
```

A disadvantage is the need for implementing one method per attribute type. And still, the consistency of the attribute type and the names for row type and attribute must be checked at run time.

The parameter for the name of the row type may be omitted in an alternative signature, if the name can be implicitly determined. This is easily implementable with an OID containing a hint to the row type. The OID can be part of the row. Unfortunately, this does not avoid the run-time checks, but, at least, it simplifies the interface as is shown below.

```
// as method of class Row
void GetAttribute(String attrname,
Integer value)
{...}
// usage:
comp_part.GetAttribute("buildDate", value);
```

A further improvement is possible by defining attribute access methods for concrete row types. We map the row type CompositePart to a subclass of Row named CompositePart.

```
// as method of class CompositePart
void GetAttribute(String attribute,
Integer value)
{...}
// usage:
comp_part.GetAttribute("buildDate", value);
```

As a second possibility, the name of the corresponding generated class can be determined dynamically by reusing Java [1] concepts in two ways. First, since Java provides run-time type information (RTTI) for all objects in a dictionary, the name of the class of a given object may easily be found out at run time via the dictionary interface (Java Reflection API). Second, the GetAttribute method of CompositePart overwrites the inherited method of Row and, therefore, knows which row type is evaluated.

This scope reduction requires schema-specific code generation. The row type must be mapped to a separate class. On the other hand, method evaluation is more efficient, since less (generic) parameters have to be checked.

The overwritten method for attribute read access enables the application to switch from using the inherited method to using the new overwritten method without changing application code. If the parameter for the class name is part of the method's signature, as is with the first two signatures given in this subsection, it must also be part of the overwritten method's signature.

A maximum of performance can be reached by also avoiding the check of the attribute's name and type.

```
class CompositePart ...{ ...
Integer GetBuildDate() {...} }
// usage:
value = comp_part.GetBuildDate();
```

Here, the number of methods to be generated is proportional to the number of attributes in the class. But, the code calling this method is short. Moreover, the correctness of the call can completely be checked at compile time.

4.1.2 Implementations of ADTs

Principally, binding type information for implementations is independent of binding type information for corresponding interfaces. Possible combinations will be discussed in the next subsection.

We again start with the most generic solution for attribute read access. A global procedure expects the name of a class, the name of the attribute to be retrieved, and an object containing the specified attribute.

```
value = GetAttribute("CompositePart",
"buildDate", comp_part);
```

Since the compiler has no clue about the correctness of the call, all checks have to be performed at run time. Therefore, the body of the method implements the following actions.

- Ask the dictionary if the class name exists.
- Check if the given object comp_part is an instance of the given class.
- Ask the dictionary if the attribute name exists in the schema and if it belongs to the identified class.
- Determine or check (if given by another parameter) the type of the attribute and write the type information into

the attribute container `value`. Alternatively, leave the type open and delegate the determination of the type to the application programmer.

- Determine the storage location of the attribute[5]. Read and return the corresponding value.

Obviously, the above implementation is rather costly at run time. The optimization goal of attribute read access is to reduce overhead of the these actions or even to avoid costs completely. JDBC reduces overhead of attribute access through a binding mechanism at run time. The application programmer has to explicitly bind columns to variables in the programming language. This way, the main costs are shifted to the call of the function that does the binding. Thus, the performance of consecutive access is increased. We want to do something similar, but automatically at compile time and without preprocessing application code. Therefore, a CLI compiler generates a CLI. To enhance such an API, we have identified two main principles of optimization.

Introducing types and constants corresponding to schema-specific and query-specific type information *reduces the scope of evaluation* of methods at run time. This type-oriented scope reduction heavily depends on the features of the type systems of host programming languages. Especially row types and their columns are ideally suited for the generation of the corresponding types and constants of a host programming language. In contrast to simple strings, generated constants firmly restrict possible values at the type level (column names of row types, for instance). The corresponding generated types narrow the scope to certain row types and certain column types each as a whole. Constraints on row types and column types as well as accepted values at the instance level (when used as parameter types, for example) may be encapsulated. Thus, a generated body of a method exploiting generated types and constants already assumes some preconditions to be fulfilled when the evaluation of the method begins. In consequence, the body of such a method is easier to code and will perform better than generic methods.

Avoiding or improving search is based on replication. While indices are usually used to improve the performance of database queries, many operations of an API are navigational and, therefore, either do not use queries at all or only use simple index lookups. Especially for fine-grained operations, it is often beneficial to improve the index lookup itself. This is because lookups may cost more than the rest of an operation due to the evaluation of the hash function of the index[6]. Improving index lookup may be done by replicating index data into the class of, the object of, or the body of a given method. Instead of asking the index, the replicated value is read. On the other hand, replicated values are to be kept consistent with the index. This means that changing a value may require an index update as well as the update of the replicated value (double update).

These two principles may be applied at different degrees to optimize our example of attribute read access. Skipping the intermediate approaches that have defined methods for class `Row` we now discuss only the most advanced solution using a generated class `CompositePart`. In comparison with the actions to be executed by a generic method (see beginning of this subsection), the list of tasks to be performed by the generated version looks as follows:

- *There is no need to ask* the dictionary if the class name exists in the schema, because the existence has already been checked at compile time.
- *There is no need to check* if the given object `comp_part` is an instance of the given class, because this instantiation check is automatically done by the run-time system of the host programming language[7].
- *There is no need to ask* the dictionary if the attribute name exists in the schema and if it belongs to the identified class, because this has already been checked at compile time, too.
- *There is no need to determine or check* the type of the attribute, because this check, too, has already been done at compile time. *Do not write* any type information anywhere, because the result type exactly matches the attribute type.
- *There is no need to determine* the storage location of the attribute, because it has already been identified at compile time. Read and return the value of the attribute using the access information generated as part of the code of the body. Depending on the storage it may be necessary to type-cast the value to the given return type.

Obviously, this implementation is much more efficient than the generic version. In addition, it is type-safe. On the other hand, the implementation overhead is higher due to code generation. Application compilation takes longer, especially if the generated code has to be regenerated and compiled. Data independence is decreased, because schema-specific or query-specific type information gets part of the grammar of the API.

4.1.3 Interfaces and Implementations

Regarding binding of interfaces and binding of implementations two main aspects can be identified.

- Different degrees of early binding may coexist because of method overloading (as has already been shown in the previous subsections).
- The spectrum of gradual early bindings is orthogonally applicable to interfaces as well as to implementations as to be shown next. To concentrate on the major issue, we, again, only discuss the end points of the spectrum (early and late).

Some combinations of interfaces and implementations seem to be better than others. But this depends on applica-

5. The current object and additional access information that the dictionary has delivered for the checked names are used for the determination. The attribute may, for instance, be accessible through an index into an array of all attributes of the object or through the name of an instance variable defined for the class.

6. Remember that CPU costs are more important for locally executable operations than I/O costs.

7. Of course, a fully static solution avoiding all run-time checks is even more efficient, but it is not quite handy.

tion needs. The most common combinations are generic interfaces with generic implementations and early-bound interfaces with early-bound implementations (a shown previously). But, two other possibilities exist.

A generic interface can have an early-bound implementation that exploits early-bound type information. An example is the replacement of the generic dictionary lookup for a name of a row type by a generated switch statement (case statement) that embodies all possible names of the current schema as constants.

On the other hand, an early-bound interface may have a generic implementation ignoring the early-bound information given by its interface. This fourth combination may prototypically reuse an existing generic method feeding its generic parameters.

We identify the following four rules of thumb for the usefulness of combinations.

- Generic interfaces and implementations (combination L/L) are beneficial w. r. t. implementation overhead, application compilation, and data independence.
- The replacement of a generic implementation by a generated one (L/E) raises implementation overhead and application compilation times, but improves performance [16]. Data independence may stay at the same level when regeneration, compilation, and code replacement are automated.
- Early-bound interfaces and implementations (E/E) are most efficient. In addition, they improve error handling. On the other hand, implementation overhead and application compilation times are increased, and data independence is decreased.
- The combination of early-bound interfaces and generic implementations (E/L) considerably simplifies the prototypical API design or extension. Calls to such an interface are type-safe. Furthermore, an optimized implementation may substitute generic code at any time.

We think that the combination of generic interfaces and early-bound implementations embodies a high potential for optimization in SQL environments. We name this combination ***virtual late binding***, since early binding of implementations is hidden from the application programmer.

4.2 Mapping SQL Concepts

After having introduced the major principals, we proceed with discussing the mapping of SQL concepts.

4.2.1 Generic Mapping

Of course, we cannot discuss all concepts of SQL3 here, so we concentrate on a few that are most important following the top-down access sequence in applications:

- handling SQL statements;
- iterating result sets;
- accessing single rows of query results;
- accessing single fields of a row.

SQL Statements. A generic class `Query` allows to handle arbitrary SQL statements. An instance of `Query` represents a concrete statement. The constructor of the class accepts the statement as string. A call of the method `compile()` triggers compilation of the statement by the DML compiler. To evaluate the statement the method `execute()` is to be performed. The result set is accessible when the evaluation has finished (or earlier in case of asynchronous execution and early delivery of partial result sets).

Iterating Result Sets. A cursor or iterator mechanism is needed to identify a single row for subsequent access. For a detailed discussion of different cursor concepts see [16]. We think that it is a good approach to provide exactly one instantiated cursor as part of `Query`. Such a built-in cursor is most efficient since it assumes that there are no other cursors, and, therefore, does not have to care for keeping several cursor spheres consistent. If more than one cursor is needed or wanted, additional cursors can explicitly be instantiated from a special class `Cursor`. These cursors know about each other and about the existence of the built-in cursor, and, thus, cannot invalidate states of other cursors, respectively. They provide comfortable access sacrificing performance.

Accessing Single Rows. Usually, cursors are positioned to a specific row, in order to access that row. Rows are represented by instances of `Row`. Several methods allow for the manipulation of single rows: `toString()`, `set()`, `copy()`, `delete()` and so forth. Each of these methods may be parameterized to be "shallow" or "deep".

An explicit "fetch into host variables", as supported by JDBC, is not needed. The fetch is implicit and hidden, as supported by the new SQLJ. A call to one of the methods mentioned above automatically involves a dereferencing of the current row. The dereferencing is used as the event to trigger the fetch of SQL data[8].

Accessing Single Fields. A row type specified in a table or query definition consists of fields each representing a pair of field name and data type. The generic class `Row` for row types must be able to handle arbitrary query results with arbitrary numbers of fields (represented by `Field`).

According to the SQL standard, user-defined types implicitly define observer and mutator functions. In analogy, we define access methods to columns of row types for the binding to the host language. In contrast to the SQL binding mechanisms, our approach avoids the need for explicitly binding columns to host variables by the application programmer. A generic method `getField(int)` or `getField(String)` of such a class allows to access all fields by index (depending on the definition sequence) or by name (depending on their names in the table definition[9]). If the table structure is defined by a UDT, then

8. In C++ the dereferencing operator "->" can be overwritten in order to implement a smart pointer that fetches a row if it is not already cached. Since Java does not allow to overwrite the dot-operator, this action has to be implemented in the body of the methods mentioned above or by introducing an indirection using a descriptor (like the Java Reference Class).

9. The initialization of such names occurs at query execution time. In case of compiled queries, the association of names and access information may be stored in the database at compile time. In consequence, the association is only read and does not need to be constructed at query execution time.

getField behaves like getAttribute of the UDT (see below) avoiding the indirection through field.

Several data types are to be distinguished: predefined types, collection types, reference types, locator types, and user-defined data types. User-defined types are schema elements, that is, they are part of a DB schema.

Predefined types include numeric types, string types, the boolean type, datetime types, and interval types. There are always two possibilities to map these types to Java, our representative of OOPLs. Either database types are mapped to predefined Java types or to user-defined Java classes. In the first case it is necessary that the predefined types on both sides are equivalent (range of values, precision) or are at least compatible. Such a mapping is most efficient, but usually provides poor control of value modifications. The second case, mapping to classes, allows for the encapsulation of database types. This approach is not as efficient as the first one, but provides better control of changes, automatic maintenance of constraints (NOT NULL, restrictions like for date), and new operations, which are not available on the server. In addition, polymorphism and subtyping can be exploited, thus, enabling the generic Field to hold any predefined type.

Collection types are mapped to Collection. The only currently proposed collection type is "array". SQL arrays are either mapped to the predefined type constructor "[]" or to Array as subtype of Collection. The former case is, again, faster than the latter case, but provides poor control. C++ offers the concept of templates to realize the latter case. In Java we exploit polymorphism and restrict the element types by their common supertype Element representing all possible element types. The Java approach saves code in the API implementation, but is slower since the exact type of an element has to be determined at run time.

Reference types are mapped to Ref hiding physically stored values (database reference, virtual memory reference). The API implementation may choose between two interfaces implemented by the cache. The first interface already delivers location transparency simplifying the API code above. The second interface forces the API implementation itself to provide location transparency. This interface is harder to use but allows for application-specific optimization. In analogy to the location of objects, it is possible to reach swizzling transparency. In consequence, the API implementation can choose between existing swizzling strategies or can implement an application-specific strategy.

Locator types are used to hide transferring of very large data values or of parts of these values[10]. Locators provide location transparency of SQL data in the absence of explicit references (see reference types above). Moreover, locators are used to handle UDTs. This is the only SQL way to work with user-defined types in the APIs.

10. The standard differentiates between several kinds of locators and their features which we do not discuss here.

User-defined types are schema elements. Therefore, we need a generic UDT to handle arbitrary UDTs. A UDT can be distinct or structured. A *distinct* UDT is based on exactly one predefined type (its base type). Distinct types support strong typing, that is, argument and parameter types of routines must be the same. A method getValue() returns the value of such a field. A *structured* UDT defines a list of attributes and may be subject to inheritance. To generically access those attributes, a method getAttribute() is needed accepting at least a description of the wanted attribute's name (see section 4.1.1).

4.2.2 Generated Mapping

In order to introduce early binding of type information, we have to generate classes specific to schema and query type information.

SQL Statements. Given an SQL statement with name "Parts", a generated class QueryParts allows to handle exactly this SQL statement. An instance of QueryParts represents that statement. The constructor of the class does not need to receive the statement as a string, because it is known at compile time. Calling the method compile() is optional, since the query has already been compiled. Nevertheless, it is useful to support recompilation in the case that preconditions on which the current query execution plan is based on have changed. To evaluate the statement the method execute() is needed. The result set is accessible when the evaluation has finished (or earlier in case of asynchronous execution and early delivery of partial result sets).

Iterating Result Sets. A cursor or iterator mechanism is still needed to identify a single row for later access. The initially instantiated cursor as part of QueryParts knows about the usage of the result set specified by the application programmer. In consequence, cursor operations like "next" are viewed as events and trigger actions, say, prefetching or swizzling of SQL data.

Accessing Single Rows. After having positioned a cursor to a specific row, this row can be accessed. Rows are represented by instances of class RowParts. This class encapsulates the row type implicitly given by the compiled query. Query-specific methods allow for the type-safe manipulation (setting, copying) of corresponding rows. The implicit dereferencing on method invocation may be used to control fetching of fields.

Accessing Single Fields. The generated class RowParts for the given row type of the compiled query can only handle query results with the specified number and names of fields (class FieldX represents a field with name 'X'; the name scope is the package resulting by code generation for the given query).

For each field 'X' a generated method getX() allows access to the field named 'X'. The result type of the method maps the data type of the field.

Predefined types are not affected at all from early binding at the interface. Collection types, reference types, and locator types are independent of the DB schema, too. User-defined names that could be early bound do not exist in their context. But, we can generate specialized methods for the data type ‘Y’ “behind” a collection type, a reference type, or an locator type. The containing fields provide access methods that deliver “collection of Y”, “reference to Y”, or “locator for Y” respectively. In consequence, the application programmer can perform save type-casting, and coding profits from type safety.

User-defined types are schema elements and, therefore, have names that can be bound early. In addition to the generic `UDT` we generate a specialized `UDT AtomicPart` that can handle instances of AtomicPart only. For each attribute ‘x’ of AtomicPart we generate an access method `getx()`. If ‘x’ is of type ‘y’ then this method returns a value of the type that maps ‘y’.

5 Integration of Programming Models

Different programming models exist on both sides of the coupling of database languages to programming languages. The following aspects have to be discussed:

Kind: SQL3 is n-set-oriented and programming languages like Java, C++, and C are navigational. Adding collections and search over single collections with simple search predicates (SSP) to programming languages, they become 1-set-oriented. Nevertheless, the programming model of the coupling is neither purely n-set-oriented nor navigational or 1-set-oriented. Both styles are available. Most often, the result of n-set-oriented operations like SQL queries is examined with navigational or 1-set-oriented operations. We name this combination *semi-set-oriented*.

Location transparency: Knowing the location of data supports efficient manipulation. Hiding the location of data supports easy coding. Our solution is to provide location transparency to the application programmer. Underneath, optimization may orthogonally occur. Different interfaces with and without location transparency are only internally visible.

Flexible functionality: Different applications have different needs. Thus, it is useful to allow for the configuration of APIs. A browser application does not need early binding at the interface, for instance. Therefore, it should at least be possible to switch support for early binding of the interface. The same argument holds for optimizing applications that use late-bound interfaces. The possibility to switch between late-bound and early-bound implementations of interfaces allows for the optimization of the application without application source code changes.

Decoupled query processing: Queries may be instantly evaluated or evaluation may be deferred. Deferred evaluation demands for the decoupling of query compilation and query execution. This decoupling may be local, inside the same transaction, or global, perhaps spanning different processes. The last case is important for design applications. To support this kind of decoupling we need persistent queries, i. e., storing and maintaining query information in the database. Additionally, it must be possible to find compiled queries at run time, to optionally recompile them, and to execute them. Besides saving the compilation overhead at run time, the generated code for handling such queries and their results inside an application is checked at compile time.

Result set processing: Cursors or iterators are needed to process result sets. As mentioned before, we distinguish flat cursors and nested cursors which support iteration at different levels of abstraction and navigational access at each level (access from nested cursors to flat cursors, for instance). It is useful to have 1-set-oriented operations with cursors. However, the more query functionality the cursor is equipped with, the more server code replication is required at the client.

Dictionary access: Browser applications need dictionary access. Furthermore, in our opinion, any application may profit from dictionary access. Since many operations of the API use dictionary information, dictionary access is mandatory in our architecture.

6 The Configuration Language

The CLI compiler is based on the database language and, optionally, based on a configuration language, which decouples API generation from application development. With a configuration program it is possible to introduce different degrees of early binding and to optimize the application program.

To reduce the overhead of early binding especially for fine-grained operations like attribute access, we prefer a type-oriented instead of a value-oriented approach. In a type-oriented approach types are configured and, thus, all of their values or instances share the same strategy. In a value-oriented approach, the strategy implemented for values or instances depends on their values or states respectively. Value orientation demands for run-time checking and strategy migration in case of value or state changes.

As we have learned from several projects, a configuration language should only depend on DBL concepts and on the methods provided with the API. Thus, configuration becomes implementation independent and portable. In addition, code generation primarily extends the bodies of methods of the API, since these are viewed as events.

In the following, we want to introduce our configuration language. Terminal symbols are written in capitals. We use the following meta-symbols of an extended BNF:

```
#    end of grammar rule
[]   option
{}   choice
[]+  many times, at least once
[]*  many times, optional
```

A configuration program (cl_program) consists of configuration blocks. A configuration block may be a type control block or a query control block (type_cntrl_block or query_cntrl_block). We concentrate on query control blocks here.

```
cl_program ::=
    BEGIN [query_cntrl_block |
    at_cntrl_block]+ END #
query_cntrl_block ::=
    DEFINE_QUERY name AS sql_query
      [BINDING binding ;]
      [SWIZZLING swizzling ;]
      [query_rule ;]*
    END_QUERY #
```

If a query control block is specified, then the generic query class is specialized. The result, the generated query class, contains the name of the query. The AS clause contains the definition of an SQL statement.

The BINDING clause determines the degree of binding for the (nested) cursor of the query result type.

```
binding ::= LATE|VIRTUAL_LATE|EARLY #
```

LATE, the default, corresponds with a late-bound interface using a late-bound implementation. Access uses a dictionary mapping node names and attribute names to node cursors and attribute offsets.

VIRTUAL_LATE means virtually late binding, that is, a late-bound interface on top of an early-bound implementation. Access omits dictionary lookup by testing names against constants contained in the method's body. It is possible to switch from LATE to VIRTUAL_LATE without changing application code.

EARLY demands for the generation of an access method per node and attribute. Node names and attribute names are used to define the names of these methods. The above mentioned tests may be omitted.

Of course, more degrees than shown here can be defined in order to fine-tune applications.

As an example for the use of API method calls as events to trigger optimizing actions we concentrate on pointer swizzling. First, we have to globally decide if pointer swizzling is to be done and if so, which of the both extreme strategies is to be used.

```
swizzling ::= NONE|LAZY|EAGER #
```

NONE, the default, means that no pointer swizzling should take place for query result. Thus, chasing references costs as much as a dictionary lookup.

LAZY means that a reference is swizzled when it is first dereferenced. The advantage is that only those references get swizzled that are really needed[11]. On the other hand, dereferencing the same reference later has the additional cost of asking, whether the current reference is swizzled or not (lazy-if).

EAGER means that references are swizzled before they are dereferenced. Most often swizzling is done when instances are loaded and mapped into virtual memory. The advantage is that the lazy-if is saved, because it is always known in advance that references are swizzled. On the other hand, usually more references are swizzled than needed.

Obviously, there are situations where no strategy fits best. In these situations it may be reasonable to swizzle certain portions of data at certain events. In addition, performance can benefit from programmer knowledge about which references are dereferenced (more than once perhaps) or which event (call to a method) leads to dereferencing such references. In order to support such dynamic swizzling decisions, we propose simple event-action rules.

```
query_rule ::=
    ON query_event DO query_action #
```

An event is defined to be a call to the methods of the query class.

```
query_event ::= exec_event|co_event #
exec_event ::= EXECUTE|RESTORE #
co_event ::=
    FIRSTCO|NEXTCO|PREVIOUSCO|
    OPENNESTEDCURSOR #
```

By `query_granule` it is specified, which references are to be swizzled.

```
query_action ::= qswizzle_action #
qswizzle_action ::=
    SWIZZLE [query_granule] #
query_granule ::= {CURRENT|ALL} #
```

CURRENT means to swizzle the current object under control, which is a result set in case of an exec_event and a single complex object in case of an co_event. ALL can be used to trigger swizzling of all objects at the next higher level; in case of an co_event all complex objects in the query result get swizzled.

We extended the grammar to capture fine-grained objects like the nodes in a complex object and even single attributes, though the overhead increases substantially. Technically, the class for the complex-object cursor delivers the corresponding events, that is methods (not shown in the grammar above). In general, optimization cannot compensate the implied overhead if the targets to be optimized are too fine-grained. But, in some cases it may nevertheless be beneficial. For very large fields like BLOBs, for instance, selective allocation of main memory and piecewise fetching can significantly increase economic memory usage and processing speed.

7 Sample Program

To see how the discussed concepts apply to application programs we proceed with a brief example based on the query over our sample database (see figure 3). The resulting program compiles and executes a query and does some navigation through the query result. We use this sample scenario to further discuss each of the three main degrees of binding: late, early, and virtual late.

Before going into details, we want to mention that we have chosen an example dealing with complex objects. Currently, SQL3 gives only some basic support for complex objects by means of references and collection types. In order to, at one hand, give some idea of the potential of our approach, and, at the other hand, motivate our opinion that SQL3 needs better complex object support than currently offered (and that current object-relational DBMSs need better data management support at the client side) [3, 8, 6, 20], the query in our example goes beyond the capa-

11. An ideal solution would swizzle only those references that are at least used twice to amortize the swizzling overhead.

bilities of SQL3. We assume that in the FROM clause of a SELECT statement a complex structure can be specified which corresponds to a graph, where the nodes represent base tables and the vertices represent referencing columns. Furthermore, we assume that corresponding instances (complex objects) can be delivered by the DBMS as units and that nested cursor structures can be generated by the CLI compiler serving for traversing and manipulating complex objects (as well as the contained elementary objects, respectively). Due to space limitations, we cannot detail this aspect and have to refer to a long version of this paper [17].

7.1 The LATE Binding

The late-bound interfaces are always provided. Only the corresponding class definitions must be imported.

```
import COAPI.*;
```

The Java package COAPI contains the definitions of all interfaces and classes of the API to handle complex objects. The most important ones are `Query` and `NestedCursor` that control inter-complex-object operations and intra-complex-object operations, respectively.

First, an instance representing the query is instantiated.

```
SQLQuery myQuery = new SQLQuery("
      SELECT ALL
      FROM CompositePart - AtomicPart -
            Connection
      WHERE CompositePart.type='type002';");
```

A call of the method `compile` delivers the query string to the DML compiler of the DBMS.

```
myQuery.compile();
```

The name of the query or the name of the application is used to decouple compilation from execution.

A call of the method `execute` asks the DBMS to evaluate the query. A corresponding transaction is opened.

```
myQuery.execute();
```

Internally, the name of the query is used to parameterize the query execution. The result set (or only the set of representatives depending on the configuration) is shipped to the client and stored in an object buffer in the application's main memory. The result-set cursor is initially opened.

Assume, we want to position the cursor on the first element of the result set.

```
myQuery.firstCO();
```

To step into the complex object at the current position, we open a nested cursor and position it (at the first level) to the first "CompositePart" node and (at the second level) to the first corresponding "AtomicPart" node, and there set the value of attribute "x".

```
SQLNestedCursor co =
    myQuery.openNestedCursor();
co.first("CompositePart").first("AtomicPart");
co.setAttribute("AtomicPart", "x", someInt);
```

Of course, a design step will do a lot more than setting an attribute. But this simple example suffices to illustrate the basic principals, and we proceed to the end of the program by performing the `checkin` of the internally recorded changes.

```
myQuery.checkin();
```

Besides propagating changes, the checkin operation also closes the transaction.

7.2 The EARLY Binding

The early-bound interfaces and implementations are only generated on demand. Some steps have to be taken to prepare their usage in an application.

First, we give a little configuration program that defines lazy pointer swizzling (SWIZZLING LAZY) for all applications bound to the API. It defines a query named "demo" (DEFINE_QUERY) and causes the system to bind early (BINDING EARLY) and to generate an optimizing rule for pointer swizzling of results of the given query (ON ...).

```
BEGIN
SWIZZLING LAZY;
DEFINE_QUERY demo AS
      SELECT ALL
      FROM CompositePart - AtomicPart -
        Connection
      WHERE CompositePart.type = 'type002';
      BINDING EARLY;
      ON EXECUTE DO SWIZZLE ALL;
    END_QUERY;
END
```

This configuration program is input to the CLI compiler which extracts the query, passes it to the DML compiler for compilation, stores resulting meta-data, and generates additional code for the API.

Now we come back to the application using the generated interfaces and classes. First, we have to import the Java interfaces and classes again.

```
import COAPI.Generated.*;
```

Then we instantiate the representative of the query. The name of the query becomes part of the interface name.

```
SQLQuery_demo myQuery = new SQLQuery_demo();
```

Compilation (if needed) and execution of the query stay the same as in the case of late binding (see section 7.1).

```
myQuery.compile();
myQuery.execute();
```

Next, we set the result-set cursor to the first element.

```
myQuery.firstCO();
Stepping into the complex object looks
    like the following.
SQLNestedCursor_demo co =
    myQuery.openNestedCursor();
co.first_CompositePart().first_AtomicPart();
co.AtomicPart().set_x(someInt);
```

Obviously, all strings that have to be used with the late binding are integrated into type names and method names and, therefore, can be checked at compile time.

Checkin is performed in the same way as in the case of late binding.

```
myQuery.checkin();
```

7.3 The VIRTUAL_LATE Binding

The virtual late binding is generated on demand. For that purpose, a configuration program is needed.

```
BEGIN
SWIZZLING LAZY;
DEFINE_QUERY demo AS
      SELECT ALL
      FROM CompositePart - AtomicPart -
        Connection
      WHERE CompositePart.type = 'type002';
      BINDING VIRTUAL_LATE;
      ON EXECUTE DO SWIZZLE ALL;
    END_QUERY;
END
```

Note that the only clause that has changed w.r.t. to the configuration program given in the previous subsection is the BINDING clause. The application program remains similar to the solution with the late binding. As the only difference it does not have to contain the query definition, because it is already part of the configuration program.

A disadvantage of this approach is that only one query is supported. If several queries are to be supported, it is possible to extend the API by context switches using the class loader of Java. A switch to another query implies loading of the corresponding class that implements the stable interface. Then, only the instantiation of the query representative changes. But, different queries can still be used only sequentially. If the results of several queries are to be processed simultaneously, the API must be able to mix query-specific type information within a single class or to provide specializations for each query. This, in turn, implies changes to all statements that use the types `Query` and `NestedCursor`, because the programmer must then use the specializations instead of the superclasses.

8 Conclusions

We have introduced generated CLIs as a configurable specialization of classical CLIs. The central idea to become better than classical CLIs like JDBC or language embeddings like SQLJ is the possibility to choose single or many different binding times out of a spectrum of binding times for database types and operations when mapping them to object-oriented programming languages like Java or C++.

Early binding improves application performance and early error handling. As an import advantage over SQLJ, our solution enables not only schema-specific type information but also query-specific type information to be bound early. By optionally using code generation only for mission-critical paths, the compilation overhead is controllable. An intermediate binding, the virtual late binding, avoids a higher degree of data dependence. This binding substantially enhances performance of generic interfaces like JDBC or ODBC [16].

We have shown that extensions to SQL3 like support for complex objects open a new dimension for early binding of type information. Extending the FROM clause by directed graphs of referencing types in the database allow for generating efficient and type-safe inter- and intra-complex-object operations. Furthermore, we have proposed an architecture for the implementation of generated CLIs that is based on three main components: a cache, a generated run-time system representing the implementation of the operations of the interface, and a CLI compiler. The cache provides location transparency. The run-time system represents an adaptable implementation of the interface operations. The CLI compiler parses a configuration program that contains statements for the application-specific adaptation of the generated run-time system.

Acknowledgments. We would like to thank A. Lambert for his extensive implementation effort.

References

[1] Arnold, K., Gosling, J.: The Java Programming Language. Addison-Wesley. 1996

[2] Atkinson, M., Morrison, R.: Orthogonally Persistent Object Systems. VLDB Journal 4:3. 319-401. 1995

[3] Chamberlin, D.D., Cheng, J.M., DeMichiel, L.G., Mattos, N.M.: Extending Relational Database Technology for New Applications. IBM Syst. Journal 33:2. 264-279. 1994

[4] Carey, M.J., DeWitt, D.J., Naughton, J.F.: The OO7 Benchmark. SIGMOD Record 22:2. 12-21. 1993

[5] Geiger, K.: Inside ODBC. Microsoft Press.1995

[6] Gesmann, M.: Parallel Query Processing in Complex-Object Database Systems (in German). Shaker. 1997

[7] Hamilton, G., Cattell, R.G.G., Fisher, M.: JDBC[tm] Database Access with Java[tm] - A Tutorial and Annotated Reference. Comp. & Eng. Publ. Group. The Java Series. 1997

[8] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications. Proc. 13th Int. VLDB Conf. 433-442. 1987

[9] Härder, T., Rahm, E.: Database Systems - Concepts and Techniques of Implementation (in German). Springer. 1999

[10] ISO: Final Committee Draft (FCD), Database Language SQL - Part 1: SQL/Framework. 1998.

[11] ISO: Working Draft, Database Language SQL - Part 2: SQL/Foundation. 1998

[12] Kemper, A., Kossmann, D.: Adaptable Pointer Swizzling Strategies in Object Bases. Proc. 9th Int. Conf. on Data Engineering. 155-162. 1993

[13] Lacroix, M., Pirotte, A.: Comparison of Database Interfaces for Application Programming. Information Systems 8:3. 217-229. 1983

[14] Mitschang, B.: Query Processing in DBMS - Design and Implementation Concepts (in German). Vieweg. 1994

[15] Neumann, K.: Coupling of Programming Languages and Database Languages (in German). Informatik-Spektrum 15:4. 185-194. 1992

[16] Nink, U.: Coupling Design Databases with Object-Oriented Programming Languages (in German). Shaker. 1999

[17] Nink, U., Härder, T., Ritter, N.: Generating Call-Level Interfaces for Advanced Database Application Programming, Internal Report, SFB 501, Univ. of Kaiserslautern, 1999

[18] Nink, U., Ritter, N.: Database Application Programming with Versioned Complex Objects. Proc. Conf. on Advanced Database Systems (BTW'97). Informatik aktuell. Springer. 172-191. 1997

[19] Oracle White Paper: SQLJ: Embedded SQL in Java. 1997

[20] Schöning, H.: Query Processing in Complex-Object Database Systems (in German). Dt. Universitäts-Verlag. 1993

[21] Stroustrup, B.: The C++ Programming Language. Addison-Wesley. 1992

[22] Suzuki, S., Kitsuregawa, M., Takagi, M.: An Efficient Pointer Swizzling Method for Navigation Intensive Applications. Int. Workshop on Persistent Object Systems (POS). Springer. 79-95. 1995

[23] Stonebraker, M., Brown, P., Moore, D.: Object-Relational DBMSs - The Next Great Wave. Morgan Kaufmann. 1998