# Database Application Programming with Versioned Complex Objects

**Udo Nink, Norbert Ritter**
Department of Computer Science
University of Kaiserslautern
P O Box 3049, 67653 Kaiserslautern, Germany
e-mail: {nink|ritter}@informatik.uni-kl.de

**Abstract**

Database systems as basis for CAD frameworks have to provide data management as well as transaction management facilities meeting the requirements of design applications. One of the most important features is an application programming interface (API) supporting design tool implementation as well as integration of design tools into the CAD framework by integrating a database language into a host programming language. Different integration techniques have been proposed in the past. We argue that call interfaces are the choice for object-oriented environments. Especially, code generation can effectively be used to tackle the impedance mismatch problem and to achieve a seamless integration, an easy-to-use interface as well as an efficient run-time environment. We exemplify this by introducing a database management system tailored to adequate management of explicit complex-object versions. Its API integrates a set-oriented, descriptive database language into an object-oriented, high-level programming language (C++) by following a call interface approach. We also report on our prototype system and corresponding experience.

**Keywords:** Application Programming Interfaces, CAD Frameworks, Call Interfaces, Host Language Integration, Complex Objects, Versioning, Design Databases.

## 1. Introduction

Nowadays, design applications are supported by CAD frameworks [HNSB90, RS92, Wo94]. A CAD framework can be seen as a software infrastructure, providing an application environment for CAD tools. Among the vital components of a CAD framework are data management services [KS92] as well as activity control services [RMH+94]. The former have to deal with typically complex-structured and versioned design data. The latter have to control the various types of activities occurring in design processes, including transactions manipulating design data as well as cooperative actions among designers involved in a common design task. The integration of tools into a framework requires techniques for design-data manipulation within (tool-constituting) application programs written in high-level programming languages. This, in turn, demands for an adequate *application programming interface* (API) integrating concepts of database (manipulation) languages (DML) and programming languages. Integration has to consider the following aspects:

- *Functionality:* The powerful manipulation facilities of the data model, which is responsible for design-data representation and manipulation, must also be provided through the API. Additionally, activity control features, e g transactional functions, are mandatory.

Published in “Informatik aktuell: Datenbanksysteme in Büro, Technik und Wissenschaft”, GI-Fachtagung Ulm, March 1997

- *Seamless integration*: If host and database language harmonize with each other and integration results in a computationally complete language, then, code readability and usability by the application programmer are improved and contribute to acceptance.
- *Architectural and performance issues*: Adequate realization of the API functionality satisfying performance, perhaps the main criteria for acceptance, have to take into account a usually workstation/server-based processing concept (data vs function-request shipping).

Concerning the choice of the host language for integration, object-oriented languages, e g C++ [St92], seem to be best suited for two reasons. First, C++ can be seen as the current-generation programming language. And second, the provided abstract data type (ADT) concepts allow for user-defined type extensions as well as elegant integration of API functions into C++ programs.

In this paper, we will show the benefits of integrating the database language OQL (Object Query Language) into the programming language C++ and how the resulting API meets the requirements behind the above mentioned aspects. We will do so by classifying known API approaches and identifying major criteria for assessing APIs in sect 2. Then, in sect 3, we motivate why call interfaces are the best choice. The following three sections introduce OVM (Object and Version Data Model), OQL, and the software architecture of our VStore DBMS respectively. The VStore implementation follows a transformation approach mapping OVM structures to C++ structures managed by the ODBMS ObjectStore. That means that OQL statements are directly transformed into C++ code. In sect 7, we will detail that the code generation approach used in the VStore DBMS is effectively exploited to provide an easy-to-use and efficient call interface for application programming. Sect 8 finishes the paper with a conclusion.

## 2. API Classification

Before giving a classification, we first identify some important criteria for assessing API approaches. These contribute to the quite informal measure "*relative cost of human programming*" which is the accumulated amount of users' work spent for data management, correctness and integrity checking, etc, as well as respective coding. The reduction of a user's or application programmer's work-to-do becomes possible with more powerful data modelling and manipulation. Maier [Ma88] claims that the power of relational algebra as an abstraction of disk storage comes from its encapsulation of iteration. Object orientation, on one hand, delivers powerful data modelling capabilities, but, on the other hand, evolving object-oriented database programming languages likely do not encapsulate iteration. Thus, the gap between descriptive and procedural processing remains. Consequently, we still have to deal with and aim at minimizing the impedance mismatch occuring with integration of set-oriented data manipulation languages and procedural programming languages. We will do this by leveraging from encapsulation which leads to exploiting abstract data types. Beside this general problem, we identify the following criteria reflecting crucial API aspects that may serve to classify and assess known approaches:

- *Compilation Speed:* Additional compiler passes of an API may extend overall compilation time, which is quite high already for a language like C++, to an unacceptable amount.

- *Performance:* A database application at best should run as fast as a corresponding main memory program. Useful information may be collected and used during compilation to reach this goal, but, obviously, compilation speed will decrease.
- *Error Detection:* This may happen either *early* at compile-time exploiting a-priori knowledge or *late* at run-time. Early error detection prevents run-time errors and, therefore, is more suitable for application coding, but often demands for extra parsing of database operations.
- *Access Control:* The corresponding mechanisms of the database language must also span application programs. Thus, access to (buffered) database data must be controlled by the API.
- *External Coding Costs:* As Atkinson and Morrison [AM95] state, 30% of the total code for a typical database application is concerned with maintenance of mappings between database model, programming language model, and real world model. This unnecessary code performs explicit movement and representational changes (flattening, graph reconstruction) of data between main and backing store. Support through automatic mappings may significantly reduce the application code, thereby decreasing the relative cost of human programming.
- *Internal Coding Costs:* API complexity is proportional to its implementation costs. Furtheron, its usage in the software development cycle demands for its high availability and easy maintainability. Therefore, it should support its implementors to react to changes in the host language, to bug reports, as well as to extensions as fast as possible.
- *Learning Overhead:* APIs usable through well known (programming language) concepts integrate better than proprietary solutions. Again, the relation to complexity is proportional.
- *Substitutability:* Product availability is heavily influenced by possible standardization which focuses aiming of competing vendors on a common goal. As a consequence, users remain more independent and may choose between replacable products.

Now, we introduce two known classification schemes for database APIs. The first approach [LP83, Ne92] concentrates on syntactic integration of database operations and the second [Hä87, Mi95] discusses binding times for database-operation integration and database-type integration. Both are suitable in the context of classic DBMSs and programming languages but need to be reviewed in the context of object-oriented technology. We will outline pros and cons found in the literature and, when appropriate, state our own opinion.

Lacroix and Pirotte [LP83] define the notion of *call interfaces*, *simple host language extensions*, *embeddings of database languages*, and *integrated languages*.

*Call interfaces* (or call level interfaces) like the ones from IMS, ADABAS, or SQL CLI [MS93] are orthogonal to DML extensions. In contrast, dynamic SQL ([MS93], p 254) is rather a DML extension than a call interface and may be integrated by different techniques, one of which could result in a call interface. Coding with call interfaces follows the syntax of the programming language, and applications use database functionality through generic routines of given libraries. Thus, no preprocessing concept or host language compiler extension is needed, and standard compilers suffice. Therefore, compilation speed is optimal. On the other hand, performance is poor because of indirections. Moreover, access control and error detection are poor, too. Special areas for communication between programming language and DBMS are common use, but programmers are responsible for correct data allocation and deletion, which most likely is the cause of errors. Though internal coding is easy due to the decoupled approach, external coding is quite low-level and often cryptic. In general, the learning overhead is quite low and depends more on the complexity of the database data model. Finally, the authors stated that call interfaces would unlikely be subject to standardization. This has been disproved lately by ODBC [Ge95], ODMG [BF95, Ca96], SDAI (STEP Data Access Interface [ISO22, ISO23]), and - last but not least - the future call level interface of SQL [SQL3].

*Simple host language extensions* like CODASYL DML enrich programming language syntax by a small amount requiring additional compilation either through a host language compiler extension or an often favoured preprocessor solution (to be more independent). Thus, compilation speed is reduced. *Embeddings of database languages* (short embeddings) that are most often found with relational systems (eSQL, eQUEL [MS93]) or extended-relational systems [Hü92, Kä92, LKD+88] require complex preprocessors or extensive host language compiler extensions especially for query handling. Because of the massiveness, compilation speed is even more reduced. Since the major difference between embeddings of database languages and simple host language extensions is not syntactic integration but expressiveness of the chosen database language, we view them as one approach and call it *embedding*. Performance may be better because many indirections can be eliminated in the code generation phase. Static type and operation checking, automatic and therefore reliable and easy-to-use buffer allocation, as well as access restrictions on query results improve error detection and access control. But, problems arise through the generated intermediate code, which may produce error messages that can only be interpreted with knowledge about the transformation. The same holds for debugging. Code is more readable in comparison with call interfaces, but functionality is equivalent, and, therefore, the external coding costs are only slightly lower. Another side-effect is the minimal interaction between host language expressions and database statements, i e, parameters of such statements must be known symbols to the DBMS and need to be adequately declarated. In the case of set-orientation, the strong mismatch between data structures and operations results in an unnatural interface with a mixed syntax of host language and database sublanguage. This approach even restricts the use of the host language (try to use recursion, for instance). An acceptable solution is provided through the cursor concept, which is used for a one-tuple-at-a-time processing of the result set of a select statement and allows for data exchange with host language variables (eQUEL even averts explicit cursor definition, which is a first step to the abstraction of iteration). Obviously, internal coding costs and learning overhead are increased.

*Integrated languages* like Pascal/R, persistent ALGOL [AM95] and others [ABGS91, ADG93, BFS88, GNB93, E92] most often require significant host language compiler extensions (or very complex preprocessors), so, compilation speed is even more reduced. On the other hand, this results in a close match of data structures, e g by host language type extensions (relation constructor), and improves performance. Moreover, error detection and access control are extended, since, e g, even normal assignments are under API control. Furthermore, expressions may be used in any statement and their validity spans database and host language. Though programming is rather high-level, still two programming styles are mixed and the gap between procedural and descriptive processing remains. Some abstraction of iteration is achieved by the “foreach” statement. Another plus is orthogonality of persistence and type, which allows for uniform treatment of persistent and transient data. Therefore, the external coding costs are lowest here. But whereas embedded SQL always uses the same database operations with almost the same syntax regardless of host programming language (a typical exception to this is specification of operations' ends), integrated languages produce much more overhead for internal coding, which makes it harder to adapt to new technologies like new programming languages. As integrated languages concentrate on one programming language, the support of multi-language environments likely leads to the problems that actually should be avoided. Concluding, no integrated language has ever reached high acceptance because of missing substitutability.

Härder [Hä87] differentiates between *operation integration* and *type integration* in his classification. *Early* and *late bindings* are possible for both so that four basic cases can be identified. *Early operation binding* provides database operations as tokens for the host programming language which results in one uniform syntax. On the other hand, *late operation binding* results in generic procedures of the DBMS run-time system, which may be called from within a program. A concrete database operation is chosen by parameter. Whereas the first approach is based on an embedding (see discussion above), the latter embodies an interpretative behaviour and allows for naming a database operation at run-time. *Early type binding* results in symbolic addressable database types, which are also interpretable in normal host language constructs. *Late type binding,* on the other hand, demands for explicit extraction of database data, for instance, out of communication areas into host language variables and vice versa.

The following consideration relates both classification schemes with each other: CODASYL-DML and database programming languages use early operation and type binding; in contrast, SQL module language and embedded SQL offer late type binding and the CALL-DML of CODASYL provides late operation binding; late binding in both cases is found with ADABAS and SESAM. Call interfaces principally fall into the category 'late operation binding' and the other approaches (embeddings and integrated languages) into the category 'early operation binding'.

Concluding, early binding likely reduces compilation speed and increases internal coding costs and learning overhead. On the other hand, it improves performance, access control, and error detection, and reduces external coding costs. Substitutability is not affected.

## 3. Improving Call Interfaces

The classification of Lacroix and Pirotte [LP83] reflects the historical evolution. In the last few years, the trend is reversed back to call interfaces again. These classically feature late type and operation binding. But the ODMG binding to C++ [Ca96] deviates from classical call interfaces providing early type binding of schema structures and a mix of early (attribute access and instance navigation) and late operation binding (queries). The STEP data access interface (SDAI) for C++ [ISO22, ISO23] is very similar, but additionally supports both early and late binding for some operations (Get Attribute). Though for object-oriented technology operation and type binding belong together, binding times depend on the point of observation: early binding of an ADT implies early binding of its methods signatures; but in their bodies, late binding may still occur. In a concrete implementation of SDAI measured in [Ni96], a call to "Get Attribute" results in the evaluation of schema-dependent but very fast switch (case) statements instead of metadata look-up. Thus, for SDAI as a whole, binding times are even mixed. Moreover, it is a good idea to offer different language integrations for different application characteristics as will be done in the future SQL standard through embedded SQL, the module language, and the call level interface.

The recent improvements to call interfaces are closely connected to features of current-generation programming languages like C++, the most important of which is the class mechanism, which allows for, to a certain degree, user-defined type system extensions. Thus, user-defined data types are used through the same syntax as the built-in ones. Now, integrating data type and operations together as ADT combined with early binding improves integration as shown above. Moreover, ADTs allow for easily adaptable implementation changes (user-defined integration adaptation). Additionally, inheritance and polymorphism allow for easy management of OIDs or (typed) pointers. These concepts enable better control of query results which consist of sets of complex objects (CO) in our case. A typed pointer may reference a CO obeying to the interface of a generated ADT achieving a compromise between performance and access control.

Unfortunately, internal coding becomes more complex because early binding is reached through generated (and in most cases schema-dependent) code. It turned out in our prototypes that code generation itself is quite manageable, but its analysis and documentation is more difficult. Clearly, the generator part needs most attention in the case of reengineering our system.

As we have shown, call interfaces do not necessarily imply late binding. In fact, operation binding and procedural coupling of software components are orthogonal. Therefore, classification of integration techniques should consider both aspects. Then, the two call interfaces ODBC [Ge95] (late operation and type binding) and SDAI (see above) belong to different classes as it should be; they are quite different indeed.

Before discussing our API approach, we introduce the corresponding DBMS, called VStore (short for VersionStore). The following two sections will outline data model and language.

## 4. Object and Version Data Model - OVM

VStore implements the object and version data model (OVM) that [KS92] developed for the management of design data. The major concepts of this model are illustrated in fig 1.

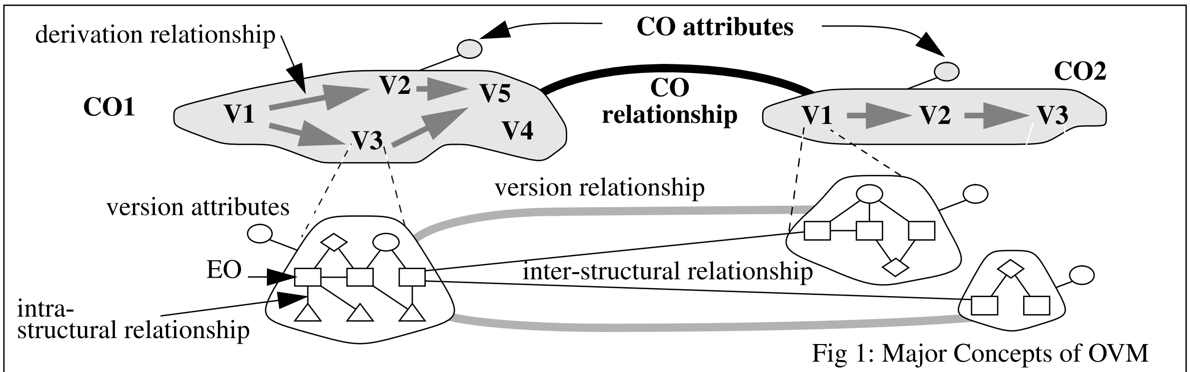


Fig 1: Major Concepts of OVM

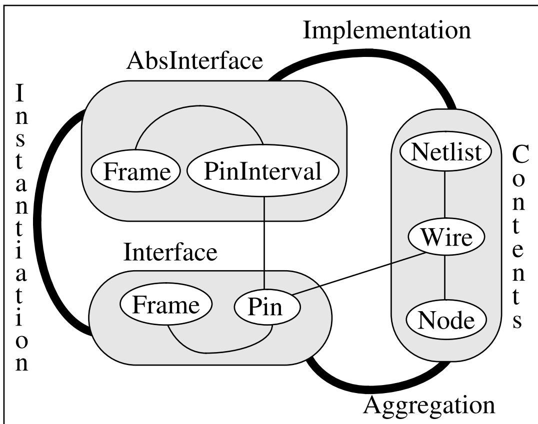
**Complex Objects and Versions**

*Complex Objects* (CO) are identifiable occurrences of CO types and are structured sets of elementary data. In its totality, a CO is described by CO attributes. It combines *elementary objects*

(EO), which can be compared with the tuples of the relational model. In addition, EOs can be connected via (typed) *structural relationships*. Obviously, it is one of the major tasks of a design process to create “contents” of COs, i e nets of EOs. Design is an iterative process typically leading to several (similar) nets of EOs which we call *versions* (complex-object version, COV). Thus, versions are different states of the net of EOs constituting a CO. In this way, versions are capturing the various CO states derived during the design process with the intention of a step-by-step improvement of preliminary data in order to reach the (partial) design goal. The relationships between the versions of a single CO, representing the derivation of new versions out of existing ones, are managed in form of a derivation graph, which, in turn, may be organized as list, tree or acyclic graph.

**Relationships between Objects / Versions**

COs may be connected by CO relationships. Obviously, these relationships must also be captured at the version level. Therefore, version relationships are refinements of CO relationships, i e, the former depend on the existence of the latter. In addition to the explicitly represented version relationships, implicit relationships can be modeled resulting from data overlapping. Overlapping can occur directly by shared EOs or by inter-structural relationships, which should only be interpreted in the scope of configurations for consistency reasons.




```
DEFINE OBJECT_TYPE AbsInterface
   COMPONENTS      ( Frame, PinInterval        )
   OBJECT_ATTRS    ( id:                IDENTIFIER,
                     name:              STRING,
                     function:          STRING    )
   VERSION_ATTRS   ( id:                IDENTIFIER,
                     scale_factor:      REAL,
                     no_feedthroughs:   INTEGER   )
   VERSION DERIVATION GRAPH IS TREE;

DEFINE LINK_TYPE Aggregation
   FROM Contents TO Interface: (1,1000)
   FOR VERSIONS: (1,1000),
   FROM Interface TO Contents: (1,1)
   FOR VERSIONS: (1,*);
```

EO type — CO type

structural relationship — CO / version relationship type

Fig 2: Applying OVM Concepts

**Configurations**

It is the goal of configuring to establish consistent units by selecting certain versions out of the set of versions stored in the database [Ka90]. Configurations (not shown in fig 1) are occurrences of specified configuration types, which are usually associated with special integrity constraints expressing requirements the corresponding unit of versions has to fulfil. Configuring usually is an explicit process of selecting appropriate versions, and configurations mainly serve as a kind of handle for a consistent set of versions. Therefore, the versions contained in the configuration continue to be the major units of processing (not the configuration). For this reason, we will not detail the configuration concept any further in this paper and refer to the corresponding literature [KRS96].

**Example**

Since we cannot explain all OVM concepts in a detailed manner, we want to illustrate the major concepts with an example from the area of VLSI design. Chip planning [Zi88] is a very important phase within the VLSI design process. Here, a hierarchy of VLSI cells is considered, reflecting a decomposition of the resulting chip's functionality. In the following, we will refer to the set of tools applied during chip planning as the *chip-planner*. A single chip-planner run considers a single cell of the mentioned hierarchy together with its direct subcells. As a result, the chip planner delivers an arrangement of the subcells within the area given for the supercell (topography). Modeling the design data by means of OVM concepts may lead to structures illustrated in fig 2.

On the left-hand side, the instance structures are depicted; the right-hand side shows sample data definition statements (due to space restrictions, we just give a single example for the definition of a versioned CO type and an CO/version link type, respectively). The graphical illustration of the schema shows three CO types. *Interface* and *AbsInterface* describe the interface of a VLSI cell. Whereas *Interface* represents a concrete interface and, therefore, contains the EO types *Frame* and *Pin*, *AbsInterface* is an abstraction of the interface description (EO type *PinInterval*), so that several *Interface* objects can be associated with the same *AbsInterface*. The CO type *Contents* consisting of the EO types *Netlist*, *Wire* and *Node* describes the internal construction of a cell. Considering the semantics of the relationship types *Instantiation*, *Implementation*, and *Aggregation*, it becomes obvious that the path *Interface-AbsInterface-Contents* connects a supercell with its direct subcells.

## 5. Object Query Language - OQL

The example given in the previous subsection contains a sample data definition statement for both a CO type and a CO/version link type. The EO type definitions were omitted, because they are very similar to tuples in the relational data model. In this subsection, we will briefly discuss the object query language (short OQL) being the language of OVM. We only consider those features needed to understand the following API discussion.

```
CREATE VERSION
   ( scale_factor := 1.0,
     no_feedthroughs := 0)
   COMPONENTS
   Frame          (  id = 1234                          )
   PinInterval    (  id = 5678                          )
                  (  name := "xxx",
                     x := 1.5, y := 1.0, len := 0.5     )
FROM AbsInterface
WHERE id = 4711;
```

```
SELECT   OBJECT
FROM     Contents-<Aggregation>Interface
WHERE    EXISTS Interface(name = "1-Bit-Adder");

SELECT   VERSION cell
FROM     Interface-<Instantiation>AbsInterface-
         <Implementation>Contents
WHERE    EXISTS AbsInterface(no_feedthroughs < 3);
```

Fig 3: Sample OQL Statements

**Data Definition**

The definition of a CO type (see fig 2) contains clauses for aggregating EO types (COMPONENTS clause), specifying CO attributes and version attributes as well as defining the structure of the derivation graph. The sample definition of link type Aggregation shows that different cardinality restrictions can be specified at CO and version level.

**Data Manipulation**
OQL contains statements for the creation and deletion of COs, versions, links, configurations, as well as for update-in-place modification of COs and versions. Due to space restrictions we just give an example for the creation of an *AbsInterface* version (see fig 3, left-hand side).

**Data Selection**
The select statement provides means for retrieving complex structures (CS). CSs consist of CO/ version nodes and relationships. On the right-hand side of fig 3 some examples are given. The first statement retrieves CO level information (CO attributes, CO relationships), i e, the nodes of the resulting complex structures are CO nodes. The FROM clause expresses that simple hierarchies are considered consisting of a *Contents* object and all *Interface* objects used within. In the same way, CSs can be defined in the 'select version' statement (see second statement) retrieving version level information (version attributes, version relationships, EOs, structural relationships). In general, the FROM clause allows the specification of nets, but with the restriction that a unique root is specified. Thus, a select statement retrieves for each root CO/version a CS as defined in the corresponding FROM clause. OQL statements may contain very powerful predicates (WHERE clause) expressing value-based as well as structural requirements.

## 6. Software Architecture of VStore

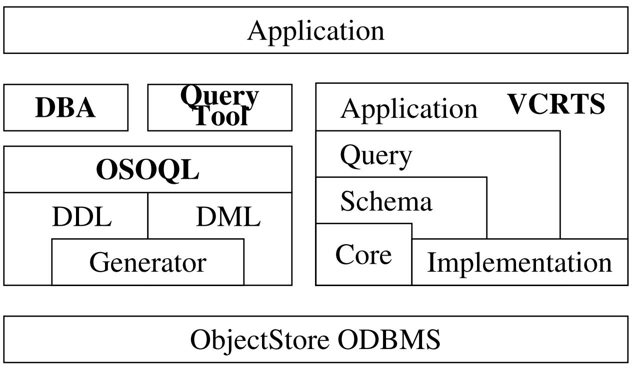


Fig 4: VStore Software Architecture

As already mentioned, VStore is implemented on top of *ObjectStore*. In the following we explain the software architecture of our prototype system (see fig 4). A database administrator may use a tool (*DBA*) to install a schema, or to influence *OSOQL* which is the DDL and DML compiler of VStore. A *Query Tool* allows for compiling queries with OSOQL and saving the corresponding code. These queries may later be called from within a program. We implemented a graphical user interface to VStore upon these components to test their functionality and to have ad-hoc access.

### 6.1 Mapping of OVM Structures and Operations to C++

In this subsection, we consider OSOQL in order to learn how schema information is mapped to C++ and how database operations are implemented.

#### 6.1.1 DDL

In C++, a versioned CO is represented through an CO representative containing the CO attributes (as instance variables) and one version representative for each version (set of references). Furtheron, each version representative aggregates the corresponding version attributes and EOs. In addition, information for internal use (metadata, relationships inherent to OVM, support for cooperation, versioning, and integrity checking) is included where appropriate.

Translation of DDL statements is done in three phases, which we explain for DEFINE OBJECT_TYPE. First, the parser checks syntax and semantics and writes metadata (into the ObjectStore database) reflecting the schema change. Second, the code *Generator* defines a C++ class and, thereby, writes a file named o_<object type name>.hh containing class definitions for object and version representatives. Third, for each schema change implying significant modification of existing instances schema evolution takes place in order to integrate the new classes into the ObjectStore schema. Significant modifications occur with EXPAND, SHRINK, and DEFINE LINK_TYPE, because these operations change the set of attributes associated with a type implying a change of its memory layout.

### 6.1.2 DML

DML statement translation, which we explain for the SELECT statement also needs three phases. First, the parser generates internal representations for the complex structure type given through the FROM clause and the predicate defined in the WHERE clause. Second, a nested C++ class named Q_<name of the query> is generated for the complex structure type. Each of its classes contains a method *expand*, which is used at query execution time to build up the substructures below the current node. Query execution is initiated through a call to the static method *evaluate* of the root class taking a list of parameters as input. The generated code for *evaluate* implements the predicate in the WHERE clause and coordinates the query result construction. Third, an executable program linked with the generated classes must call *evaluate* to trigger checkout. The query result is then transferred to the client (see 7.1 for details).

## 6.2 Using the API through the VStore Client Run-Time System (VCRTS)

The VStore client run-time system (VCRTS, see fig 4) consists of several components. Basic mechanisms and generic data types dependent only on our data model can be found in the *Core* part. Here, handling of identifiers, user-defined links, COs, versions, components, internal links, and address tables is determined. *Implementation* contains a pool of different implementations for the abstract data types the object buffer is built on. In generation phases, user input may be used to select implementations for the construction of higher level VCRTSs, which are:

- *Schema VCRTS*: Decisions that may be associated with schema information like implementation of set-valued attributes are input to DDL statement translation. The resulting code represents the basic interface to user-defined types.
- *Query VCRTS*: DML-statement translation, especially translation of the FROM clause, produces the object buffer's infrastructure and the (hierarchical) cursor interface that is directly visible to the database application programmer. At this stage, decisions like "use some pointer swizzling strategy with hierarchical cursors as control instances" may occur.
- *Application VCRTS*: Different data access sequences specific to different applications may be considered by adequate buffer and cursor tuning (list sizes; backward cursor moving) at application compile-time. The interface produced through the Query VCRTS remains unchanged.

We force an application to be linked with the *Query* or *Application VCRTS* and access database data through the generated nested cursors only which lets us control data access. The matter of database operation specification (queries) is separated from the rest of the application code and may be shifted to a database administrator. The application programmer needs far less knowledge about the database system, he only has to learn ADTs and their interfaces. Now, this is just the same as with normal third-party libraries: standard C++ header files tell the programmer what may be done and how. The usage of database objects is much like the usage of those on the heap. This approach, in our opinion, leverages more from the host language capabilities than integrated approaches do.

### 6.3 Implementation Experience

ObjectStore from Object Design Inc [LLOW91], the infrastructure we have chosen, is a page-server ODBMS with a very close binding to C++. The system in fact is rather a DBPL than a DBMS in the traditional sense and provides two different embeddings to C++. The first interface, called *library interface,* allows access to all database functionality through the use of class libraries. As database schema construction is based on C++ type definitions and more precisely the database data model is mainly based on this language's type system (persistence orthogonal to type), we have early binding for types as well as operations here. For browser-like applications not knowing a database schema in advance, the metaobject protocol also offers late binding for data access. In either case, the programmer or schema developer has to deliver type descriptions for each persistent type, which is mainly a task of providing unique names. The second interface called *DML* adds some keywords to the host language (persistent, foreach), simplifies the use of database features (type information is automatically generated), and is offered through a specialized *cfront* compiler called *OSCC*[1] delivering C code, which may then be compiled with a standard C compiler.

We have chosen ObjectStore because we didn't want to build the system from scratch. Neither its versioning facilities nor its meta object protocol seemed to be adequate for us but we have made heavy use of many other features like persistence, transactions, queries, references through object pointers, navigation, and schema evolution. Some restrictions on these had quite some influence on the prototype design:

- We have built a multi-user system and could have saved a lot of work, if ObjectStore did not lock pages; we implemented complex-object-level locking ($C^3$-locking protocol, [Ri96]), which allows for higher concurrency, especially in combination with our versioning mechanism, and is, however, much cheaper than object-level locking. The costs are rather comparable to those of page-level locking since components outnumber COs.
- The query facilities are quite flat in that the input is a single collection of object pointers; this also holds for the output. We implemented a query transformation approach splitting OQL queries into a hierarchy of ObjectStore queries glued together with some code.

1. Remarkably, the vendor will substitute this integrated solution with a preprocessor approach because of poor acceptance from industrial users disliking to rely on a non-standard compiler and maintainability. Meanwhile, schema parsing has been extracted out of OSCC into a new schema generator.

- A set of objects may be versioned together independent of type. Workspaces qualifying at most one version per object provide access to them. Though manipulation of several versions of the same object at the same time, which is a key concept in our system, is possible with several workspaces, access gets more complex and data exchange between workspaces more indirect. In addition, the mechanism is very space consuming and prevents clustering.
- The metaobject protocol has been designed for the host language and is not extendable, e g to support version links; we built a metadata manager from scratch. Though this alleviates porting, we would, today, choose a hybrid approach to save the effort.

## 7. Application Programming Interface

In nowadays systems, code generation takes place for schema data and non-associative operations (say navigation). We transfer these mechanisms to set-oriented database languages, i e our prototype language coping with versioned COs. The goal is to enable application programmers to leverage from the given data model and query language as much as possible through pure C++, and at the same time averting the mentioned disadvantages of call interfaces.

As CAD applications show high referential locality as well as a-priori knowledge about the processing context, we have chosen and implemented an object-buffer-based processing model. With the object buffer at its heart that is loaded through checkout queries, an interface is provided to locally manipulate the buffered data. Any changes of buffered data are automatically propagated back to the server at commit. This processing model called **checkout-work-check-in** minimizes communication between server and workstation, allows for catching server failures, and maximizes application performance.

A VStore application checks data out into a complex-object buffer placed in the same virtual address space using precompiled queries. Several queries may be executed per application and their results (the CSs) are stored in multiple partitions (one for each query), which overlap in shared database COs/versions. Here, we will not discuss merging modified buffer contents and new query results or propagation of changes in one buffer to another; so, for the rest of the paper, we stick to the simple model.

We realize that applications do not always have a closed processing context, i e, some are not able to describe all needed data in advance. To support open processing contexts, successive single-COV loading may be used. Starting from a single COV, an application may browse the corresponding version derivation graph and load neighbours when needed.

Because of automatic change propagation at commit, object flags for update operations are used at run-time. The resulting overhead can significantly reduce performance of operations especially of those working on EOs, since these mainly fill the buffer, i e, their number is quite high compared to that of COs, and they are quite small in size (again compared to COs). Using preclaiming for checkout (tell the system what you intend to do with the query result) to adapt the object buffer to application needs we avoid this overhead for read-only data.

### 7.1 Checking out the Processing Context

For the following discussion, imagine that the Schema VCRTS has just been built and the schema is now stable. A typical VStore application knows this schema and uses precompiled queries describing the processing context. Sometime between schema translation and application compile time these queries have to be compiled using the Query Tool. The resulting code (C++ class Q_cell for the chosen query) must be linked to the application. Part of the code representing the example query translated to ObjectStore DML looks like this:

**(1) cs [: !nAbsInterface [: no_feethroughs < 3 :].empty() :]**

'cs' represents the extent of the root in the CS defined in the FROM clause. Everything between '[:' and ':]' is an integer expression to be evaluated on each element of 'cs'. Regard that the path expression on 'AbsInterface' ('nAbsInterface' is the internal name) results in a nested query. The EXISTS predicate is translated to 'collection not empty'. If we extend the WHERE clause to 'status = $p AND EXISTS ...' the result will be:

**(2) cs [: status == V_p && !nAbsInterface [: no_feethroughs < 3 :].empty() :]**

To compare an attribute like 'status' of 'Interface' to a query parameter 'p' a global (local to the query class) variable 'V_p' is defined, which may be used anywhere in the expression. The same mechanism is used to allow comparisons between objects of different nodes in the CS.

To trigger checkout a call to the static member function 'evaluate' is sufficient which results in two execution phases. First, a set is allocated and, topdown, each of the CS nodes is linked to a COV that may qualify. Second, this set is filtered through the predicate in the WHERE clause. The result is then locked on CO level. Internally, short ObjectStore transactions are used for read and write phases.

Afterwards, the result is transferred to the client. Behind the scenes, it is copied into a newly allocated database segment called application segment. There, address tables are created that manage old and new addresses of COs, COVs, and EOs and other useful information like access right (read, derive, update) to or update states (unchanged, updated, inserted, deleted) of an entry. In addition, pointer swizzling takes place which transforms some (again depending on the query) of the inter object references in the application segment. While developing on top of ObjectStore we decided to execute checkout, work, and checkin in subsequent transactions so that page-level locking of ObjectStore does not interfere with our CO-level locking protocol. But then, references to and from database objects must be kept consistent over transaction boundaries. This is possible with special ObjectStore classes for pointers from the heap into the database but not vice versa. In addition, we saw a big problem in dynamically linking the generated code into the running C++ application. Therefore, we decided to execute checkout, work, and checkin in separated processes. This lead us to a quite elegant solution named **persistent object buffer** (see fig 5). Checkout and checkin are generic ObjectStore clients (an exception is the usage of generated query classes in checkout) copying data from one database segment into another and freeing locks when finished. The application itself is also an ObjectStore client but works on the segment holding the query result only. Cache management at the client and mapping of pages into virtual main memory is subject to ObjectStore.

We identified the following advantages with this approach.

- Checkout and checkin now logically belong to the VStore server (the connection to 'Application' in fig 5 reflects inter-process communication) which is an ObjectStore client and may physically be on the server or client machine.
- The application has a smaller footprint (needs less main memory).
- The single-user object buffer is now directly reflected in our architecture. A lightweight persistent object service like PSE from Object Design Inc would further improve application performance and further decrease its footprint.
- Because of stronger decoupling from the server and reuse of persistence mechanisms the implementation of application-controlled interruption of program execution, i e interruption of the transaction on the object buffer, can significantly be alleviated.

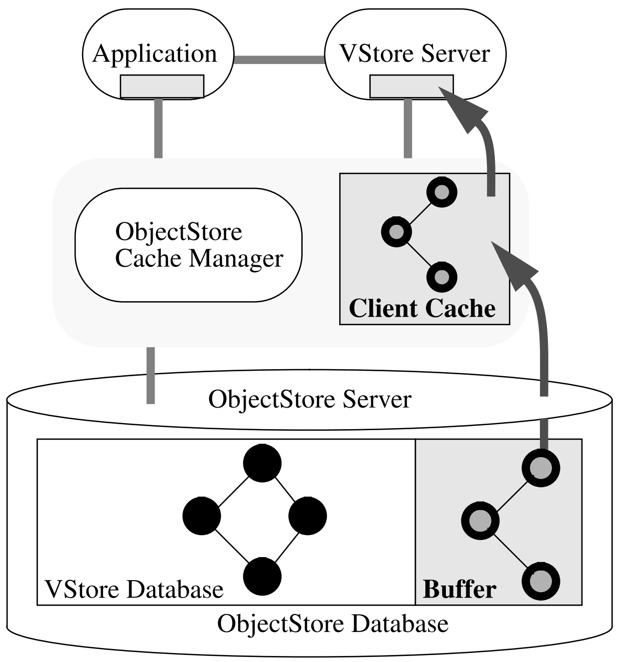


Fig 5: The Persistent Object Buffer

Unfortunately, the problem of keeping references consistent over transaction boundaries returns with the last point or when single-COV-loading shall be used. While this, clearly, is no severe performance penalty for the former it would be for the latter, since such interrupts may be frequent.

For the former a solution is to provide entry points, or roots, into the application segment (you also have to save such things as the segment's name or address). Restoring the application simply means starting from these roots (this is more difficult for stopping than for pausing the application).

There are at least two solutions in the case of the latter. Either locking on the object buffer is switched off, if possible, or demanded COVs are copied into yet not locked pages. For the former, API implementors have to synchronize the loading process against the application. The latter implies shifting some VStore server functionality into the application to update address tables and object references and, if specified, swizzling pointers. Because switching off locking is not supported we allocate new pages.

### 7.2 Object Buffer Manipulation

With hierarchical or nested cursors reflecting the complex-structure-oriented view on query results we support local work on buffered data. The query's FROM clause is used to generate a nested cursor class named CURSOR_<name of the query> in a very similar way to generating the query class. Each node in the directed acyclic graph wraps, i e controls access to, an CO type in the FROM clause. The top node may only reference roots of CSs, and each son of a node may only reference buffered or newly inserted partners of its father for the given link type between them. The cursor class has only one instance. And only with this, updating may take place because we experienced a massive overhead for simple operations with an earlier prototype supporting many cursor instances. But as it may be useful to remember visited objects, we allow instantiation of read-only nested cursors that conserve cursor states.

```
class cell_CURSOR {
  public:
    cellInterfaceCursor*        Interface();
    cellAbsInterfaceCursor*     AbsInterface();
    cellContentsCursor*         Contents();
    cell_CURSOR&                setTo(cell_Readcursor *);
...
```

Fig 6: Nested Cursor

Nested cursor methods exist to control iteration on all levels (first, next, previous, last), navigation within a single CS (using the dereferencing operator), and navigation on the version derivation graph (successor, nextSuccessor, predecessor, nextPredecessor). Manipulation of buffered data (EOs, COs, and COVs) includes reading/writing attribute values, creating/deleting objects, and setting/unsetting links. Again, nested cursors provide the interface and, therefore, adequate access control. In addition, they are used to specify operands for binary operations like “connect” which sets a link between two objects. Fig 6 shows part of the generated class definition of a nested cursor. The method ‘setTo’ allows for conserving the state for binary operations or backtracking. One of the CO cursors that build up the nested cursor is shown in fig 7. ‘getSurrogate’ delivers an identifier for an CO/version the programmer may use as substitute in subsequent operations. ‘object’ leads to the CO representative and ‘frame’ to a component. Attribute access methods and cursor iteration methods follow. Cursors for EOs are quite similar and, therefore, omitted.

```
class  cellAbsInterfaceCursor : public cellVOperations {
  public:
                                              cellAbsInterfaceCursor();
    virtual void                              connect(cellOperations*, cellLinkNames);
    virtual Surrogate*                        getSurrogate();
    virtual void                              createSuccessor();
    cellAbsInterfaceObject*                   object();
    cellFrame_AbsInterface_eobject*           frame();
    char* status(); void status(char*);
    void setFirst();  void setNext(); void setPrevious();
    void succ();  void nextSucc();
...
```

Fig 7: Subcursor

The usage of these methods is illustrated in fig 8, which shows an excerpt of a sample application program changing a *PinInterval*’s position and updating corresponding *Pin*s. The first two lines contain pointer declarations for CO cursors, which may be set to some version nodes (navigation in CS in lines 6 and 7) after transaction begin (line 4) and query execution (line 5) to abbreviate following code. The *AbsInterface* is noted (reading an attribute in line 9) and changed in its *x* coordinate (writing an attribute in line 10). All *Interface*s (lines 11 to 17) that are incorrect now (cursor as parameter in line 12) get new versions (line 13). We then navigate through the implicitly updated version derivation graph (line 14), reposition a *Pin* (line 15), and set a link between the new *Interface* and the *AbsInterface* (binary operation in line 16)[2]. Line 17 switches iteration to the next CS.

```
01    cellInterfaceCursor *ifc;
02    cellAbsInterfaceCursor *aifc;
03
04    TX adop = TX::begin();
05    q_cell->evaluate();
06    ifc = q_cell->interface()->moveFirst();
07    aifc = q_cell->interfaceA()->moveFirst();
08
09    cout << “Changing:” << aifc->id() << endl;
10    aifc->pinInterval()->moveFirst()->x(newx);
11    do {
12        if (!correct(ifc)) {
13            ifc->createSuccessor();
14            ifc->moveSuccessor();
15            ifc->pin()->x(calcX(aifc));
16            ifc->connect(aifc); }
17    } while(ifc->moveNext());
18    adop->end();
```

Fig 8: Sample Application Code

‘createSuccessor’ is a very good example for enriching an API to decrease external coding costs; its algorithm shall be explained now. Let ‘ifc’ be a cursor on ‘Interface’ already positioned as in the sample code. The call to ‘createSuccessor’ will check the access right to the ‘Interface’ and return with an error if it is ‘read’ only. Else a new COV is created which implies construction of a new version representative with a temporal identifier, setting its access right to ‘update’, updating internal helpers like reference counts and status information, connecting the new version with its CO representative, and updating the version derivation graph. If successful, the new version is appended to an additional successor list in the cursor which is used to differentiate between navigation in CSs and navigation in version derivation graphs. The temporal identifier becomes persistent at checkin or on execution of the ‘getSurrogate’ method because a surrogate shall survive process boundaries.

### 7.3 Checking in the Changes

Propagating updates is one of the steps performed within the method finishing the application transaction (line 18 in fig 8). These steps are:

(1) Opening an ObjectStore transaction;

(2) (Preliminary) propagation of updates;

(3) Checking consistency;

(4) Repairing inconsistencies through application-specific exception handling;

(5) a) Aborting the ObjectStore transaction in case of (non-repairable) consistency violations;
b) Committing the ObjectStore transaction otherwise;

(6) Releasing (VStore-) locks and deleting the application segment.

Obviously, the abort operation (for VStore transactions) is pretty simple. It just consists of performing the last step of the outlined process.

---

2. As the two used CO cursors are part of the same nested cursor, connecting in fact works with only one nested cursor here. If objects shall be connected across CSs, the above mentioned read-only cursor is needed.

Incorporating the checkin step into this process has the advantage that the application programmer is not unnecessarily burdened by having to deal with explicit update propagation (external coding costs are reduced). The overall process is performed within a separate operating system process, which, in turn, is an ObjectStore client being usually executed on the ObjectStore server machine in order to minimize communication overhead. This process depends on the generated Schema VCRTS because of the contained consistency checks and of application-specific exception handling for fixing consistency violations. The latter, if used, implies that an (application-) program-specific process must be provided. The corresponding overhead, at first glance, seems to be a disadvantage but, actually, it is not, because it can be generated as soon as the exception handling is coded.

The (preliminary) propagation (step 2 in the list given above) consists of the following steps:

(2.1) Collecting ‘update information’.
The global address table contains flags indicating the update states of all COs, COVs, and EOs contained in the object buffer. During local work corresponding flagging is performed by generated update operators (see sect 7.2). In order to only propagate the net effect, address table information is exploited to determine the smallest set of updates which are to be performed on the VStore database.

(2.2) Performing the actual propagation of updates into the database (by updating, inserting, or deleting COs and/or COVs) as indicated by the information gathered in step 2.1.

At first sight, at least the second step seems to depend on schema information and, thereby, require code generation. Actually, generating code for change propagation is only one possibility. Instead, we implemented a generic approach because the main reasons for code generation (performance enhancement, early error detection) don’t apply to these modules. Descriptive access to the database is the crucial performance factor, and early error detection is primarily useful in application development.

## 8. Conclusion

First, we have identified important criteria for API comparison. Then, existing API classification schemes have been discussed, and some deficiencies in transferring experiences to object-oriented environments have been detected. Furthermore, solutions to the primary problems of call interfaces were explained. Especially, *performance*, *error detection*, *access control*, *external coding costs*, and *learning overhead* may significantly be improved by code generation techniques (see assessment of our prototype in the rightmost column of tab 1). Moreover, the last few years have proved *substitutability* for call interfaces through several standards.

An application programming interface for a DBMS managing versions of COs on top of an ODBMS and experiences gained in its implementation have been described. We have been making heavy use of code generation concepts for query integration thereby shifting interpretation complexity to compile-time and providing early binding for operations and types to support application development. Unfortunately, *internal coding costs* are higher and need further investigation.

We tackle the impedance mismatch problem by encapsulating COs in nested cursors generated from complex-structure descriptions of FROM clauses of queries. This simplifies internal and external coding and allows for application tuning, because of complex-structure-oriented pointer swizzling and hiding from implementation-specific details like storage structures used.

**Table 1: Integration Technique Assessment**

| | call interface (past) | embedding | integrated languages | **call interface (current)** |
|---|---|---|---|---|
| compilation speed | + | o | - | **-** |
| performance | - | o | + | **+** |
| error detection | - | o | + | **+** |
| access control | - | o | + | **+** |
| external coding costs | - | o | + | **o+** |
| internal coding costs | + | o | - | **o** |
| learning overhead | + | o | - | **o+** |
| substitutability | - | + | - | **+** |

Notes: **-** : bad **o** : medium **o+** : nearly good **+** : good

Though integration has been significantly improved, the gap between descriptive query specification and procedural application development remains. The only tidy solution is an integrated language based on some extendable programming language. But as such an approach will never find high attention, we concentrate on further improvement of the integration degree of call interfaces. Especially, *compilation speed* needs to be further ameliorated.

The biggest coding problems encountered, were provoked by the difficulties of dynamic loading of generated code into running C++ applications. In addition, metadata management, generic access to objects, and providing run-time type information accounted for the main implementation overhead in our prototype. Next, we will study integration with the Java programming language [Java95], which promises built-in support for dynamic class loading and replacement, run-time type information, and secure memory management. This project will be aimed at intranet-based CAD frameworks.


## Acknowledgments

The authors would like to thank Th Härder for his helpful comments on an earlier version of this paper.


## Literature


ABGS91 Agrawal, R, Buroff, S, Gehani, N, Shasha, D: Object Versioning in Ode. Proc Int Conf on Data Engineering, ICDE'91, Kobe, Japan, pp 446-455.

ADG93 Agrawal, R, Dar, S, Gehani, N: The O++ Database Programming Language: Implementation and Experience. Proc Int Conf on Data Engineering, ICDE'93, Vienna, pp 61-70.

AM95 Atkinson, M, Morrison, R: Orthogonally Persistent Object Systems. VLDB Journal, 1995, 4:319-401.

BF95 Bancilhon, F, Ferran, G: The ODMG Standard for Object Databases. Proc 4th Int Conf on Database Systems for Advanced Applications, DASFAA'95, pp 273-283.

BFS88 Baker, D A, Fisher, D A, Shultis, J C: A Practical Language to Provide Persistence and a Rich Typing System. Workshop on DBPL, Univ of Pennsylvania, Philadelphia, Nov 1988, pp 253-268.

Ca96 (ed) Cattell, R G G: The Object Database Standard: ODMG-93 - Release 1.2. Morgan Kaufmann, 1996.

E92 Exodus Project Document: An Introduction to GNU E. Univ of Wisconsin, Report WI 53706, 1992.

Ge95 Geiger, K: Inside ODBC. Microsoft Press, 1995.

GNB93 Gala, S K, Navathe, S B, Bermudez, M E: Voltaire: A Database Programming Language with a Single Execution Model for Evaluating Queries, Constraints, and Functions. Proc Int Conf on Data Engineering, ICDE'93, Vienna, pp 283-292.

Hä87 Härder, T: Realisierung von operationalen Schnittstellen. Datenbank-Handbuch, (eds) Lockemann, P, Schmidt, J W, Springer Verlag, 1987.

Hü92 Hübel, C: Ein Verarbeitungsmodell für datenbankgestützte Ingenieuranwendungen in einer arbeitsplatzrechnerorientierten Ablaufumgebung. Dissertation, Univ of Kaiserslautern, 1992.

HNSB90 Harrison, D, Newton, R, Spickelmier, R, Barnes, T: Electronic CAD Frameworks. Proc of the IEEE, Feb 1990, 78(2):393-417.

ISO22 ISO TC184/SC4/WG7: Product Data Representation and Exchange, Part 22: STEP Data Access Interface.

ISO23 ISO TC184/SC4/WG7 N403 (Committee Draft): Product Data Representation and Exchange, Part 23: C++ Programming Language Binding to the Standard Data Access Interface Specification.

Java95 SUN Microsystems: The JAVA Language Specification - Version 1.0 beta. 1995.

Ka90 Katz, R: Toward a Unified Framework for Version Modeling in Engineering Databases, ACM Computing Surveys, 22(4):375-408, Dec 1990.

Kä92 Käfer, W: Geschichts- und Versionsmodellierung komplexer Objekte - Anforderungen und Realisierungsmöglichkeiten am Beispiel des NDBS PRIMA. Dissertation, Univ of Kaiserslautern, 1992.

KRS96 Käfer, W, Ritter, N, Schöning, H: Konfigurierungskonzepte für Entwurfsumgebungen. Univ of Kaiserslautern, Department of Computer Science, submitted for publication.

KS92 Käfer, W, Schöning, H: Mapping a Version Model to a Complex Object Data Model. Proc 8th Int Conf on Data Engineering, ICDE'92, Tempe, Arizona, 1992, pp 348-357.

LKD+88 Linnemann, V, Küspert, K, Dadam, P, et al: Design and Implementation of an Extensible Database Management System Supporting user Defined Types and Functions. Proc Int Conf on Very Large Databases, VLDB'88, Los Angeles, California, 1988, pp 294-305.

LLOW91 Lamb, C, Landis, G, Orenstein, J, Weinreb, D: The ObjectStore Database System. Communications of the ACM, Oct 1991, 34(10):50-63.

LP83 Lacroix, M, Pirotte, A: Comparison of Database Interfaces for Application Programming. Information Systems, 1983, 8(3):217-229.

Ma88 Maier, D: Why Database Languages Are a Bad Idea. Workshop on Database Programming Languages, Univ of Pennsylvania, Philadelphia, Nov 1988, pp 334-344.

Mi95 Mitschang, B: Kopplung von Programmiersprache und DB-Sprache. Anfrageverarbeitung in Datenbanksystemen - Entwurfs- und Implementierungskonzepte. (eds) Härder, Th, Reuter, A, Vieweg 1995.

MS93 Melton, J, Simon, A R: The New SQL: A Complete Guide. Morgan Kaufmann Publishers, 1993.

Ne92 Neumann, K: Kopplungsarten von Programmiersprachen und Datenbanksprachen. Informatik Spektrum, 1992, 15(4):185-194.

Ni96 Nink, U: Effizienzbetrachtungen zu SDAI auf OODBS. Informatik Xpress 8, Proc CAD'96, Verteilte und intelligente CAD-Systeme, Kaiserslautern, Mar 1996, pp 430-445.

Ri96 Ritter, N: The $C^3$-Locking-Protocol: A Concurrency Control Mechanism for Design Environments. Proc STAK'96, Softwaretechnik in Automatisierung und Kommunikation, Munich, Germany, 1996, pp 95-110.

RMH+94 Ritter, N, Mitschang, B, Härder, T, et al: Capturing Design Dynamics - The CONCORD Approach. Proc Int Conf on Data Engineering, ICDE'94, Houston, Texas, 1994, pp 440-451.

RS92 Rammig, F J, Steinmüller, B: Frameworks und Entwurfsumgebungen. Informatik-Spektrum (1992) 15: 33-43.

St92 Stroustrup, B: Die C++ Programmiersprache. Addison-Wesley 1992.

SQL3 ISO/IEC 9075-3:1995: Information technology -- Database languages -- SQL -- Part 3: Call Level Interface (SQL/CLI)

Wo94 van der Wolf, P: CAD Frameworks - Principles and Architecture. Kluwer Academic, 1994.

Zi88 Zimmermann, G: PLAYOUT - A Hierarchical Layout System, Proc 18th GI Conference, Hamburg, Springer, Informatik-Fachberichte 188, 1988, pp 31-51.