

Using Object-Oriented Databases in Software Engineering: An Overview of Five Products

Luciano Baresi

Politecnico di Milano
Piazza Leonardo da Vinci, 32
I-20133 Milano (Italy)
e-mail: baresi@elet.polimi.it

Abstract

CASE applications require sophisticated features from the underlying database management systems. Many researchers have identified object-oriented databases as a promising technology to successfully address these issues.

Aim of this paper is to provide an overview of object databases as to a set of requirements derived from software engineering. The first part of the work outlines the requirements. The analysis is principally centered on SPADE, a process-centered software engineering environment being developed at CEFRIEL and Politecnico di Milano, even if further concerns are derived from other works in literature. While the second part gives an overview of five object-oriented database management systems, both commercial products (*O₂*, GemStone, DEC Object/DB and Object-Store) and research prototype (*Ode*). After a brief presentation, it is shown whether and how each database meets the requirements.

1 Introduction

Bernstein [8], in 1987, has argued that software engineering applications require specialized database management systems (DBMS): CASE tools generate great quantities of data (for example, documents, programs, manuals and test data), whose management could be facilitated by using a database that provides services to share and access data maintaining its integrity and consistency. In the eighties, systems were based, directly or indirectly, on the relational data model, that seemed not to adequately support the posed requirements. Bernstein's bias was that the new functionalities could not be added as a layer on existing relational databases, since they would not have had the adequate power and performances. Instead the new DBMS architectures, specifically targeted for design applications, should be exploited. The same view was later shared by other works [17, 16], that pointed out that both conventional, i.e., relational, technology, and structurally object-oriented databases, were clearly inadequate for these issues.

Based on over a decade of research, object-oriented database management systems (OODBMS) came to widen the application of database technology to new kinds of

applications: computer-aided software engineering (CASE), mechanical and electrical computer-aided design (CAD), computer-aided manufacturing (CAM), scientific and medical applications, office automation and even a new generation of business applications where traditional DBMSs proved inadequate.

As explained in [12], these new DBMSs own the same data management capabilities as relational DBMSs: large amount of persistent information shared by many users. Moreover, they add the concept of object management: much more complex kinds of data, procedural data, encapsulation of data semantics and other new capabilities.

OODBMSs became rapidly a major research area. Among many papers describing single experiences, the first attempt of providing a general framework is contained in [28], while more recently the Object Database Management Group (ODMG)¹ has published the ODMG-93 standard [18] and all the ODMG participants have committed to support this standard in their commercial products in the near future (within 18 month from the publication).

This paper provides an overview of the features offered by five OODBMSs as to software engineering applications. The presentation of the databases is preceded by the definition of the requirements to be met. They are divided into two groups. The first set considers the requirements posed by SPADE: a process-centered software engineering environment (PSEE) implemented on the O_2 OODBMS at CEFRIEL and Politecnico di Milano. The second group, on the contrary, lists those features, coming mainly from [19] and [35], not preeminent in SPADE, but that could be important in other kind of systems.

The paper is organized as follows. Section 2 lists the main requirements posed by CASE applications. Section 3 gives an overview of the considered OODBMSs and, finally, Section 4 draws some conclusions.

2 Requirements

The presentation of the requirements posed by software engineering applications is organized into two subsections. Section 2.1 refers to the SPADE environment and lists its main needs, while Section 2.2 presents a set of features derived from literature.

2.1 SPADE Requirements

SPADE (Software Process Analysis, Design and Enactment) is a software process environment, being developed at CEFRIEL and Politecnico di Milano. It supplies mechanisms for the definition, analysis, enactment, and evolution of a software development process.

SPADE provides a domain-specific language for modeling software processes called SLANG (SPADE LANGuage) [5, 6, 7]. SLANG is based on high-level Petri nets and is formally defined in terms of a translation scheme from SLANG into ER nets. ER nets [25] are a mathematically defined class of high-level Petri nets that provide the designer with powerful means to describe concurrent and real-time systems. In ER nets, it is possible to assign values to tokens and relations to transitions, describing the constraints on tokens consumed and produced by transition firings.

¹The ODMG is a working subgroup of the Object Management Group (OMG) consisting of vendors of OODBMSs. Object Design, O_2 , Versant, Object Technology, Ontos, Objectivity, Servio and Itasca are some of its members.

The following sections list the main requirements that SPADE/SLANG poses to the underlying database. The interested reader can refer to [3, 4] for a more SPADE-oriented and complete description.

2.1.1 Dynamic Schema Evolution

Software production is a long-lived activity submitted to continuous modifications to cope with new requirements or to improve delivered products. This is the reason why SLANG is a reflective language: the definition of a SLANG specification (model) may be changed by the execution of the specification itself. There are several kinds of changes that may be applied to the overall model during process execution. For instance, it may be varied the structure of the model (i.e., the topology of the net) or the types of the information it is contained in.

In conclusion it should be possible to change a model while it is executed. However it would be not feasible to stop the execution and change the executor whenever parts of the model should be modified. Thus the underlying OODBMS must provide facilities for modifying an application and/or its database schema without stopping the application itself.

2.1.2 Type Versioning and Object Migration

As already mentioned in section 2.1.1, the definition of the types of the information the process deals with may be changed, even while the process is executing, i.e., it must be possible to apply changes to the schema concurrently with the execution of models that are instances belonging to the same schema (e.g., instances of a class *Model*). Schema updates must be safe: each change must leave the database in a consistent state. In particular, when a type definition is changed, the existing instances of the changed type have to be handled properly: some mechanisms are needed to migrate the previously created objects from the old definition to the new one.

An OODBMS should supply an automatic mechanism for converting objects to the new type either only when the object is accessed (lazy migration), or as soon as the new type is defined (eager migration). An alternative approach would allow user-defined migration policies providing the basic features to implement intermediate strategies between fully eager and fully lazy migration.

A complementary aspect to type migration consists of providing support for type versioning: both type survive, new objects will be created according to the new type definition, while old objects are accessed according to the corresponding type version.

2.1.3 Distribution

SPADE supports multi-user project development by allowing the distribution of running activities over a number of workstations. The model of the process that supervises and governs the activities is shared among the workstations. There are, basically, two ways to implement this scenario. The first solution is based on distributed accesses from the users' workstations to a DB server. The whole model resides on the server, that becomes a key component to determine the performance of the entire system. This approach appears to be interesting for small to medium projects, and for large projects that can be split into quite independent sub-projects. The second idea, suitable for larger projects,

is to distribute transparently the process model execution over several workstations. This implies distributing the process data, consisting mainly of the artifacts that are produced locally by each user. The tools used should not need to know anything about the physical distribution of the used data: it should be the responsibility of the database system to manage physical distribution. This approach reduces the client-server traffic on the local area network, while it introduces the need for consistency control mechanisms to preserve the consistency of the distributed databases.

2.1.4 Tool Integration

In software engineering environments, an OODBMS can be very useful in providing a common interface (guaranteed by the shared schema) to applications on distributed clients.

Very often tools share common data structures. For example, a syntax-directed editor and a compiler can share the same abstract syntax representation of the source code. In the SPADE environment some tools and the process interpreters are database applications² sharing a common schema (i.e., a common set of data type definitions) and a common base (i.e., a common set of objects representing software artifacts). In this scenario it is necessary that the underlying repository provides applications with an easy way for data communication

The OODBMS should also support tools (clients) located on heterogeneous machines, i.e., the system should be able to contemporary manage applications running on different operating systems.

2.2 General Requirements

There are some features that are frequently mentioned as requirements for a software engineering database, but that play a minor part in the SPADE environment. In fact ad-hoc SLANG nets can be designed to implement these requirements without resting on the underlying database.

Several works in literature face the use of OODBMSs from different point of view: [8] provides an extended abstract about features required by software engineering, [19] envisages process-centered environment repositories and [35] presents an approach to software process modeling based on an object-based model called PMDB+.

The features arising from these papers and are not considered as principal requirements of SPADE are briefly described in the next subsections.

2.2.1 Transaction Model

CASE environments require complex transaction models. They need constructs like nested transactions, long transactions, or explicitly programmable transaction schemes [19].

This requirement arise from the design activity, where action that must be handled as atomic ones could last for many days, for instance the modification of a code unit. It would be unacceptable that, due to a trivial hardware problem, a transaction should be aborted losing the work of many days. Thus multi-level or nested transactions are highly required in order to split a single activity into a set of subactivities. Besides this, there are

²Actually SPADE also supports tools, like Unix tools, that are not database applications. Nevertheless, such tools are not integrated at the data level.

concurrency and efficiency problems coming from exclusively locking an object or a set of objects for many days : a process that needs a locked object should wait till the end of the current transaction. On the contrary providing check-in/check-out mechanisms, also called long transactions, combined with versioning facilities, would avoid these problems.

Moreover it is necessary to properly synchronize accesses to shared objects. Locks should be placed on objects, since locking at a coarser granularity could prevent potentially parallel activities from actually proceeding in parallel. Otherwise the OODBMS should provide users with facilities to place objects on memory pages, thus by controlling object placement, programmers can avoid unexpected concurrency conflicts due to the uncontrolled placement of objects on the same locked memory pages.

2.2.2 Object Versioning

Object versioning is very useful to all the activities based on an evolving process that has to be traced during its life-time. For example, when maintaining a program, it would be cumbersome to apply all the modifications directly to the source code, without having a copy of the current version. In case of problems with the new release it would help in carrying the application back to a consistent state.

A versioned object owns different state, all sharing the same identity. An object can be referenced no matter of its current version. Cross references among object would be lost in case each version would have its own identity and it would be up to the application providing the right pointers.

Versions improve also cooperative work since alternative versions can be managed by concurrent processes that do not need to wait till the unique version (copy) of the required object becomes available.

Moreover version facility allows to control the appropriate/current version of an object or set of objects, sometimes referred to as *Configurations*, establishing different views of the same database.

Versioning of artifacts could not be considered as a major requirements since it could be seen as part of the application itself, then explicitly programmed, instead of as a service provided by the underlying database system. However specific support supplied by the OODBMS would help programmers in dealing with versioned objects and would improve application's performances.

2.2.3 Triggers and Constraints

Triggers [35] and integrity constraints [8] are used as control or monitoring mechanisms (resembling exception handling in programming languages). The use of these feature impose an higher degree of discipline and control to the users of an OODBMS. In fact, monitoring the state of the database and performing user-defined actions each time some condition is violated guaranties data consistency even against unforeseen situations.

Basically they implement active objects [35], i.e., objects that perform actions as a consequence of their state. For instance, when an object *Document* is to be stored, a constraint, controlling its state, decides whether it conforms the standards and then either stores it or notifies the user with an appropriate error message.

2.2.4 Query capabilities

The last requirement concerns query capabilities [19]. Databases are frequently accessed to retrieve information, but it is not always the case of coding the request using a programming language, a declarative interface could be more appropriate. In fact the nature of such queries cannot usually be determined in advance and then they cannot be hard-coded. The OODBMS should provide an extensible query language, since allowing only predefined operations, as in standard *SQL* for relational databases, could not be enough. Due to the high heterogeneity and, in some cases, to the complex structure of the information stored, not all the circumstances could be covered, thus users should be free to customize the language according to their specific needs.

Notice that, as the object data-model is more complex than the relational one, the query language for an OODBMS, as explained in [10], is not merely a trivial extension of the relational *SQL*, but new issues, such as object identity, complex object structure, methods and class hierarchy have to be considered. Thus highly different query capabilities are provided with the currently available OODBMSs.

3 OODBMSs

This section gives a brief description of the following OODBMSs: *O₂*, GemStone, Ode, DEC Object/DB and ObjectStore³ respectively. They were chosen both because they represent significant examples of the current available products and because they were available.

The presentation of each OODBMS is guided by the previously outlined requirements and deals with:

- the architecture of the system;
- the data model;
- the languages (interfaces) that can be used to program database applications;
- the tools offered to program and/or interact with the database;
- object persistency;
- the transaction model;
- the support to object versioning;
- the schema management, type versioning and object migration;
- the query capabilities.

The starting points of this presentation are some works that compare or describe the OODBMSs presented in the next sections. Other than those papers describing a single system, that will be referenced later, [39], [9], [1], [42] and [27] provides overviews of some systems, even if considering different aspects or/and older versions than the ones presented here.

³Here we refer to version 4.5 of *O₂*, version 4.0 of GemStone, version 3.0.3 of Ode, version 2.0.0b2 of DEC Object/DB and version 3.0 of ObjectStore. Some observations and comments reported in this paper may not be valid for different versions.

3.1 O_2

The first prototype of O_2 [15] was the result of a research project started in 1986 by the Altair consortium⁴, while the first commercial version was delivered in 1991 by *O₂Technology*.

O_2 is based on a client-server architecture, where clients are responsible for the execution of methods and applications and the server has in charge data (file) management. Clients may be either on different workstations in a network or even share the same workstation with the server. Currently, due to the client/server architecture all the information is stored in a centralized repository, but in the future actually distributed databases will be supported.

The system provides four different programming interfaces: O_2C , C , $C++$ and OQL . Users can easily design application through *O₂Tools*, a complete graphical programming environment, and add graphical interfaces by means of *O₂Look*, a tool built on top of *X Window* system and *Motif* that provides a set of high-level functions to display and edit complex objects.

Moreover O_2 offers *O₂Kit*, a library of reusable software components, *O₂Graph*, a package to create, modify and edit any kind of graph and *O₂API*, an application programming interface.

O_2 [29] maintains the distinction between data definition language (DDL) and data manipulation language (DML), respectively O_2 and O_2C . Information, defined with O_2 , consists of *objects*, instances of classes, and *values*, instances of types. The main difference is that the identity of a value depends on the information it carries on, while an object has an its own identity, a state, i.e. a value, and a behavior, determined by the methods defined in its class and by the current state. The DDL features multiple inheritance and lets the schema designer decide whether each single class attribute should be private, read-only or public, i.e., three levels of data encapsulation. Whereas, methods can be only public or private. Actually encapsulation is not orthogonal to inheritance: you can not redefine as private a property that is public in the ancestor class.

Objects' behavior is expressed in O_2C . It extends the ANSI C to support the object-oriented data model of O_2 . O_2C is also used to code *applications* and *functions*, whose signatures have to be defined in O_2 . An *application* is a set of programs having a common purpose and run directly by users, while a *function* is similar to C functions: a subprogram not related to any class; it is useful because it can be called from anywhere within the schema it is defined in: from methods, applications or other functions. O_2C also provides some useful data structures like strings, lists and sets together with powerful operators to manage them.

O_2 can also be interfaced with $C++$ applications by importing and/or exporting class definitions, and with C programs by means of O_2 libraries to which the applications are linked.

Before beginning to work with O_2 , the users have to define the *Schema* and the *Base* they want to use. A *Schema* acts as the repository for all the definitions, while a *Base* contains all the persistent objects created within an application. The set *Schema* and *Base* provide the working context: every definition, compiled code and persistent object is added to the *Schema* or *Base* directly. The basic advantage is that, through the definition

⁴Altair was a consortium funded by INRIA (Institut National de Recherche en Informatique et Automatique), by IN2 (a Siemens subsidiary) and LRI (Laboratoire de Recherche en Informatique, University of Paris XI Orsay).

of a context, O_2 allows incremental compilation of applications. The implementation of a single method can be changed without the need for recompiling the whole application, but simply by compiling what has been really changed; the same is worth for adding a new method or function.

Schemas and *Bases* features also a mechanism for information hiding at a higher level. The classes defined in a *Schema* can be imported from another *Schema*, that uses their public services without the need to know the implementation details. This is what usually happens when an application imports, for instance, the class *Date* from the *O₂Kit*. The import/export mechanism can be applied to *Bases* too, in order to share instances instead of definitions.

In O_2 the schema can be managed at run-time. Through the predefined persistent object *Metaschema*, that describe the schema of the current *Base*, O_2 provides the users with facilities to create or modify, during the execution of an application, both the definition of classes and the bodies of methods. Notice that modifying a schema may invalidate the program into which the change has been carried out. For example, persistent objects used by the current program should not be deleted.

Another interesting feature of the *Metaschema* is that it allows an application to handle data whose type is unknown at the time the program is written. The definition of the new class, together with its methods, has to be passed as a string to the predefined method `command` of the class *Meta* (the class of *Metaschema*), that adds the new class to the current schema. Then, the new class can be instantiated and its method executed by means of queries.

O_2 does not provide any direct support to type versioning and object migration, but the problem will be addressed in the future 3.1.1. Modifying class definition is safe only if there are no instances of the modified class, otherwise all the instances created according to the old class definition would become inconsistent objects.

The only way to bypass the problem is to dump⁵ the instances of a class before modifying its definition. Dumped information could be later assigned to new instances of the updated class building a sort of migrated objects. Unfortunately this mechanism does not preserve objects' identity, so that eventual references to these objects would be lost.

A very similar mechanism can be implemented through the services provided by the *Metaschema*. As described previously, the users, at run-time, can both add a new class to the schema and compile a function that translates the objects belonging to an old class into objects of the new class, copying the attribute values copied whenever possible.

Object versioning is not addressed in the current version, but it is planned to be part of the next releases 3.1.1.

Objects and values can be named, i.e., an object or value can be associated with a string label visible outside the database. Names must be statically declared using the DDL, but can be dynamically attached to objects within the code of an application, as if they were global variables.

Persistence is related to names: every named object or value is persistent, moreover every object or value which is part of another persistent object or value becomes persistent. Thus no explicit mechanism for deleting persistent object is provided, but a garbage collector will remove from the database all the object no more linked with a name or a named object.

⁵dumping, here, means saving the information carried out by instances on ordinary UNIX files.

Persistent data must be accessed within a transaction. By default an application works within a read-only transaction, i.e., persistent data can be only read, while write transactions, that allow an application to modify stored information, must be explicitly started and finished, either by a commit or by an abort operation. O_2 provides two different types of commit: besides the usual commit, the users can validate transactions, i.e. commit transactions without freeing all the temporary memory, thereby without invalidating all temporary variables and interrupting the current execution flow.

Currently the whole page an object is stored in must be locked (by a two-phase mechanism) to work on the object itself. Since there is absolutely no control by the user on object storage, unforeseen conflicts and even deadlocks can arise due to two unrelated transactions waiting for objects stored in the same page. $O_2Technology$ has planned to improve the system with a real object-level locking mechanism, that should be preceded by functionalities to let the users control how objects are actually stored, i.e., how they are clustered.

O_2 offers OQL [30], an object oriented SQL-like query language, that can be used either as an embedded function in a programming language or as an interactive ad-hoc query language. Within O_2C , C or $C++$ an OQL query becomes a string passed to an appropriate function, while interactively it allows fast and simple browsing of the database and present the results in a graphical fashion, exploiting O_2Look as graphical user interface generator.

OQL is neither computationally complete nor provide explicit update and create operators, but, since methods or functions can be invoked from within a query, it relies on the operations defined on objects for specific purposes.

3.1.1 Future Enhancements

The features presented in this section will be added to the O_2 OODBMS in the framework of the GoodStep Project⁶. Rather than developing a new system, GoodStep enhances and improves the O_2 database in order to obtain an OODBMS dedicated to support software development environments (SDE).

Views The O_2Views [37, 36] tool is implemented in O_2C on top of the $O_2Engine$, the kernel of the O_2 system. A view is defined as a *virtual schema*, i.e., a set of virtual definitions of O_2 classes derived from a real schema, that becomes the *root schema* of the view. A *virtual schema* defines a *virtual base*, the image of the real *root base* through the view, i.e., it presents the data actually stored in the base according to the new virtual definitions. In fact a view can be seen as a filtering schema through which the information of a real base is seen. Basically, a view consists of *virtual* and *imaginary classes* which define the appearance of the objects in the view through *virtual* and *hidden attributes* that respectively augment and restrict the interface of real objects.

Active Rules Rules [13] are comprised of three parts: an Event (E), a Condition (C) and an Action (A), thus their semantics is that when the event E occurs, if the condition C holds, then the action A is executed. O_2 Rules are defined as schema elements and currently associated to primitive events only. A primitive event happens when objects

⁶ESPRIT-III project 6115 GoodStep - General Object-Oriented Database for Software Processes [40, 41]

are created, values are modified, entities become persistent, messages are sent to objects, transactions start, commit, validate or abort, and also when programs start or finish. Furthermore they respect encapsulation (only public entities can generate valid events) and can be activated or deactivated by suitable messages. Due to the flat transactional model of the O_2 system, triggered rules are not seen as sub-transactions but become parts of the triggering transactions.

Object Migration The proposed solution [20] is based on migration functions that, preserving objects' OIDs, convert instances created using a class definition into objects belonging to a new class. In fact changing the identity of an object would mean also changing all the references to the object itself. Object updates can be performed either by user-defined migration functions, when supplied, or by system default transformations. These functions must be constrained to be able to ensure correctness even in the most critical situations, i.e., when an update uses information stored in different objects. The mechanism can be used to implement either eager or lazy migration policies. However lazy (or deferred) updates are preferred, since eager migration is a very expensive task when the size of the database is not trivial. Clearly changing all the instances of a class at the moment the class itself is modified is much more heavy than changing only the objects actually referred to.

Object Versioning The Version Manager (VM) does not handle single objects, but version units, i.e. sets of objects. Version units are user-defined and may dynamically evolve during their life. As soon as the current version has been set, O_2 Versioning becomes transparent to applications that have not to explicitly manage versioned objects.

From the end-user point of view, the VM is the class *Version* and its public methods are the operation allowed on each version unit. This class belongs to an O_2 system schema and it has to be imported by every schema needing to handle versions. Of course, the class *Version* can be locally refined in order to customize object versioning in accordance to the specific needs. In fact the VM provides only the basic features to define personalized mechanisms for managing versions, without enforcing any policy.

3.2 GemStone

GemStone [11], developed by Servio Corporation, became a commercial product in 1988 and currently is one of the most mature OODBMS available in the market. The system combines the concepts of an object-oriented language, Smalltalk-80, with the functionalities of database systems.

GemStone currently features a client-server architecture. Its basic components are the *Stone* monitor and the *Gem* server process:

- *Stone* manages internal resources, handles login and logout operations and coordinates concurrent accesses by multiple users.
- *Gem* supports compilation and execution of applications, manages object storage and retrieval, and verifies user authorizations. Moreover it provides programmers with a pre-defined set of classes, called kernel classes (implementing, for instance, the concepts of list, bag, string, boolean, time).

Basically *Stone* acts as a server to *Gems* for internal resources, while *Gems* become servers to applications. There are no constraints on process distribution: *Gems* can reside on any number of heterogeneous clients, but even on the server, together with the *Stone* process.

Through an evolutionary approach from a client-server to a fully distributed system, the ultimate goal is to be able to provide multiple distribution models on the top of the same underlying mechanisms.

GemStone provides a graphical environment for building applications called *Geode*: it consists of a schema editor, a database browser, a set of graphical application development tools and a workspace by which the user can interact with *OPAL*, i.e., with its data definition and manipulation language. A graphical application is based on flow diagrams, where data is sent and received by forms, i.e., frames drawn by the user to show the different fields of application objects. The system also provides an interactive alpha-numerical environment called *TOPAZ* (the workspace, provided by *Geode*, features almost the same functionality) and programming interfaces with three languages: C, C++ and, of course, Smalltalk.

OPAL [38] comes from extending the semantics of Smalltalk. Attributes belonging to a class type are distinguished in instance attributes, called instance variables, and class attributes, referred to as class variables. Instance variables are attributes of the instances of a class, thus there exists a copy of each attribute for each instance. Class variables are attributes common to all the instances of a class, stored in a single copy belonging to the object defining the class.

The user is not obliged to specify the domain of instance variables, but imposing their values to be objects of a certain class means adding integrity constraints to the database. In fact, once the integrity constraints have been included in the schema, the system becomes responsible for maintaining the database in a consistent state, otherwise it is absolutely up to the user. Anyway domain definition becomes mandatory when the user wants to perform associative queries on database objects.

OPAL supplies fully encapsulated objects, i.e., after the definition of a new class, the system automatically provides all the necessary methods for accessing the attributes, methods that can be modified by the user.

Inheritance is only single. It avoids problems due to inherited properties with the same name, but it enforces the user to build inheritance hierarchies that are "unnatural" and very complex.

Finally, *OPAL* fully supports late binding and overloading.

Since GemStone considers everything as an object, i.e., class definitions are objects too, it should be possible to manipulate class definitions and to instantiate objects of classes that do not exist on program launch. Unfortunately, the first versions of the system did not actually support schema evolution: a class could be modified only if it had not been instantiated, otherwise only the definition of its methods could be changed. This limitation is currently withdrawn and classes are freely modifiable.

Type versioning is fully supported. A new version of a class is defined by creating a new object (class definition) either with the same name or explicitly declared as a "new version" of the old class. Any new object will be instance of the new version, while the old version continue to be referenced by the old instances. Two versions do not need to share the same structure or to be in hierarchical relation, but they must belong to the same object called *class history*.

Existing objects can be migrated between two classes related to the same *class history*. No specific policies, e.g., lazy migration or eager migration, are provided:

- migration can occur when specified by the user.
- Not all the instances of a class need to migrate at the same time. Only certain instances could migrate, while others need never to be transformed.
- A default mechanism is provided, i.e., the data held in instance variables (attributes) is retained if instance variables with the same names are used in the old definition and in the new one. However the default policy can be overridden if it is needed.

Problems arise in migrating indexed instances, since indexes are not preserved after migration. Thus the indexes have to be removed, the instances migrated and then the indexes re-created.

Moreover, due to the fact that everything is an object, methods too are objects and can be defined, deleted, compiled and executed at run-time through the use of appropriate methods.

GemStone supplies only conventional, one level, transactions, which can use either an optimistic or a pessimistic policy to control concurrent accesses on objects. Since a transaction actually works on copies of data, the system can be set to check data consistency either only when a transaction commits: in case of conflicts it is actually aborted, or continuously by properly locking the objects already used by other transactions. It is important to notice that a composite object is considered as a set of pointers to its components and a reference to an object during a transaction does not mean locking the whole object of which it is an attribute, i.e., the parent object and all its components.

In GemStone persistency is orthogonal to class definition: an object can be persistent or transient, no matter of the class it belongs to. Objects are not automatically stored in class extensions, but they become persistent if directly linked to an external name (an identifier visible from the outside) or to already persistent objects. The system also gives the user clustering facilities to control object storing.

GemStone does not provide a real, independent query language, but only methods for filtering collections in order to retrieve either all the elements that satisfy a condition, or all the elements that do not satisfy a condition or only the first element that matches the condition.

3.3 Ode

Ode [21] (Object Database & Environment) is an object-oriented database, based on the C++ object model, developed by AT&T Research Laboratories. A prototype version (v. 3.0.3), not providing all the functionalities included in the original project, is currently distributed to universities and research centers. Due to the restrictions embedded in the first versions, the overview of Ode does not only address what is actually provided, but it presents also what should be added in the future.

Ode features a client-server architecture, where multiple databases can be accessed within an application. Moreover single-user applications can be run in stand-alone mode, i.e., without a server providing concurrency control.

A user can interact with Ode databases through four different types of interface: a programming language called O++ [2], an SQL-like query language referred to as CQL++ [23], a graphical browser called OdeView [22] and a UNIX file system-like interface called OdeFS [24]. Notice that only O++ and OdeFS are currently provided.

O++ is the language for data definition, data manipulation and general computation. It extends *C++* in providing facilities for managing persistent objects and their versions, for iterating over collections, for running transactions and for associating constraints and triggers with objects.

As to the schema evolution, Ode has a catalog that records the types of objects that are (or which were previously) in the database. The catalog itself consists of elements of type **metatype**. The objects' types stored can be examined by iterating over the **metatype** type extent. Modifying a type definition having instances will lead to unspecified behavior. To avoid this, in order to modify the definition of class **T**, all objects of type **T** should first be deleted. Even if a Metaschema (the catalog) is currently provided it seems only the first step towards future facilities for dynamic schema management.

The system supplies only one level, conventional, transactions extended with interesting features. Using two-version two-phase locking (2V2P)⁷, it supplies update, read only and hypothetical transactions. These ones allow users to imagine "what if" scenarios: they can change data and evaluate the consequences without affecting the database.

Persistent objects are organized in clusters. They can be stored in the objects' type extent (by default) or in explicitly specified clusters. Moreover persistent objects can be named, i.e., they are associated with labels, and they can be retrieved simply by looking for their name in the database.

Object versioning in Ode is a property of single objects and not of their classes. All persistent objects can be versioned and different objects of the same class can have different number of versions. Ode supplies both temporal and derived-from versioning⁸. Changing the contents of an object does not mean automatically to derive a new version. It is up to the application overwriting the previous content or creating a new version. Besides this, an application can handle an object either as a logical object, i. e., the object with all of its versions, or as a real object, i.e., a precise version of the pointed object.

As far as database consistency is concerned, Ode features two sophisticated functionalities: constraints and triggers, not actually provided with version 3.0.3.

Constraints are boolean conditions that can be associated with classes and must be satisfied by its instances. Constraints are checked at the end of constructor and method calls. If they are evaluated false, the transaction, of which this access is part of, is aborted and the object restored to its initial state.

Triggers monitor databases and when become true the associated actions are executed. Ode provides two types of triggers: once-only and perpetual. A once-only trigger is automatically deactivated after it has fired and it must then be explicitly reactivated. On the other hand, a perpetual trigger is automatically reactivated after being fired. Transactions representing trigger actions are executed after, not necessary immediately

⁷A number of readers and at most one writer can be operating simultaneously on the same granule. The writer has to wait for all the readers to finish before it can commit.

⁸Consider an object **a**. Its first version **a1** derives from **a**, the second version **a2** comes from **a1**, while the third version **a3** is created from **a** again. This means that the temporal history **a**, **a1**, **a2**, **a3** does not correspond to the derived-from relationship.

after, the end of the triggering transaction. If the triggering action is aborted, the trigger actions, generated by it, are aborted too.

The *Ode File System*, or OdeFS, allows database objects to be accessed and managed with standard UNIX command, just like files in a file-system. OdeFS uses an existing distributed file-system protocol (NFS), and allows file-oriented programs to work on objects.

It is based on the concept of *object file*: it looks like a regular file but it is really an alias for a database object. Because no code modification is required, proprietary applications, written to handle normal files, can access Ode objects. For instance, reading an object file means that OdeFS calls a read function provided for the object's class. In order to use objects of a certain class within OdeFS, the user must provide its interface functions, otherwise the system provides default functions.

Finally the Ode project comprehends CQL++, an SQL-like interface to Ode databases. It offers, even if it is not supplied with the version 3.0.3, a declarative interface to those users not likely to use O++. CQL++ provides facilities for creating, querying, displaying and updating objects, for defining views on them and for creating new classes. It operates upon and returns sets of objects, that can be either clusters, i.e., sets of persistent objects, or *temporary clusters*, containers where objects are stored for the duration of a session. The objects in these sets can be either of a real database class or of an *anonymous class*, a tuple obtained from the types of the expressions in the projection list. CQL++ respects information hiding. The user cannot reference private attributes or methods. Private attributes can be accessed only by calling public methods. Type specific operations, used to construct new objects and to display existing ones, must be specified by the user as part of the class definition, otherwise default operations are used.

Even if CQL++ does not support the full definitional capabilities of O++, it can be used to query a database defined and populated using O++. Similarly O++ users can access objects defined using CQL++.

3.4 DEC Object/DB

DEC Object/DB [14] came from Objectivity Inc. and subsequently bought and improved by Digital Equipment Corporation (DEC). It belongs to the family of C++-based OODBMSs: the schema of an application is translated in C++, while the application is written in C++ directly.

The system is based on a client-server architecture, where a centralized process coordinates concurrent access and storage of fully distributed data. Objects are organized according to a contained-in hierarchy. A *Federated Database*, the hierarchy root, contains the application schema and one or more *Databases*. Each *Database* can be created on a different, even heterogeneous, workstation on a local network. From this level on, all the objects are actually distributed objects. Within *Databases* there are objects called *Containers*, i.e., the units of clustering, where objects are stored. This hierarchy is highly dynamic, all the nodes are objects that can be created and/or accessed within an application, in any case the necessary information is provided by the *Federated Database*. Unfortunately, even if data storage is built in a completely distributed way, its management is centralized.

Besides a C++ interface, DEC Object/DB supplies also a C interface. The only facility provided to interact with databases is a graphical *tool manager*, i.e., a set of tools

that help the database administrator to examine a schema definition, to browse, to dump and to load a database and to manage locks on objects.

The data definition and manipulation language derives from *C++*, in the sense that it is a *C++* that, in the release we used, features only single inheritance and provides *associations* and dynamic arrays.

Basically an *association* is used to relate a class with another one, to define an attribute of the first class to be an object of the second class. It is the system that guarantees the consistency of the relation, allowing only safe operation on the involved object. *Associations*, very similar to relations in entity-relationship diagrams, can be one to one, one to many and many to many, and even bi-directional or unidirectional, i.e., the consistency is checked on the two directions, or on one way only. While defining an *association*, it can be stated that removing an object corresponds to deleting only that object or, on the contrary, removing even all the objects related with it. Unfortunately the policy has to be hard-coded, it has to be decided when defining the schema of an application, instead of managing the problem at run-time.

Modifying an application is an hard task: users have to modify source files and then recompile and relink the whole application. The same is worth for schema evolution. No support is provided. and changing or adding a class definition causes the application to be terminated, the schema modified and the application recompiled. Moreover, preserving data consistency between two versions of a schema is absolutely up to the user. The whole database must be dumped, the new version of the application released and then all the data reload. The instances belonging to a class whose definition is changed must be modified one by one before reloading the entire database.

As to concurrency DEC Object/DB supplies only conventional, one level, transactions, together with a reach set of functionalities to customize the standard policy. By default a two-phase locking mechanism is provided, but users can:

- specify how many times a lock has to be reasked in case of failure and the time interval between two subsequent requests.
- define a *container* to be a write-one-read-many (WORM) object, so that only a single process can modify the objects contained in, but other processes can read its contents as they were before the first process locked the *container*.

Moreover users can explicitly control the granularity of the objects they lock. Even if not yet available with the used version, object locking is planned, users can lock a *container*, a *database* or even the *federated database*, that would be equivalent to lock the whole database. Notice that locks can be propagated through *associations*, i.e., it can be programmed if a lock involves the single object or even the objects associated with it. Yet again this feature must be defined during schema design.

The system provides also a check-in/check-out mechanism. A process acquires an object by checking it out, preventing the other processes to work on it until it is explicitly checked in by the owing process.

DEC Object/DB lets the users program if an object should be versioned and which kind of versioning is to be applied. Both linear and branching versioning are allowed, i.e., two versions of an object may share the same parent. Furthermore merging of parallel version is user-defined. Object versions are considered as independent entities, treated as ordinary objects, even if they all are related to the same *Genealogy*. A *Genealogy* maps

the history of an object, provides the default version and gives the users a logical unitary view of the object. An application can refer either to the whole logical object, accessing implicitly the default version, or directly to a precise version.

Moreover, when two objects are linked by an *association* and a new version of one of the two objects is created, it can be programmed that:

- the association does not consider the new version and continue to point to the old version;
- the association refers to the new version, discarding the old one;
- the association is duplicated and points to both the old and the new version.

DEC Object/DB offers DEC Object/SQL++ as query language. It provides the users with facilities only to filter the objects of a collection; it is not possible to define more complex queries or use a query to invoke a method on an object.

3.5 ObjectStore

ObjectStore [34, 26], developed by Object Design, is one of the most popular OODBMS. It merges database concepts with C++ programming language. C or C++ developers already have the basic knowledge for using ObjectStore: writing an application that uses ObjectStore is just like writing an ordinary C or C++ program, except for substituting read and write operations with ObjectStore primitives.

The system is based on a multi-client/multi-server architecture:

- servers can support many client workstations,
- clients can simultaneously access multiple databases on different servers,
- a server and a client can be co-resident on the same machine.

Thus a single application can use several databases, eventually depending on different servers.

Currently ObjectStore provides three programming interfaces: C, C++ and a high-level data manipulation language (DML), an extended C++ based on AT&T C/Front C++ Language System, moreover ObjectStore for Smalltalk (see Section 3.5.1) will be introduced in the near future. The DML [31] provides users with higher-level commands for managing queries, iterations on object groups and relationships among objects⁹ and is compliant with C++ including multiple inheritance, virtual functions and parametrized types¹⁰.

As to management tools, ObjectStore supplies a *Schema Designer* to create and modify schemas and a *Browser* to display information about schema as well as about single objects store in databases.

⁹Relationships can be thought of as a pair of inverse pointers. If an object points to another, the second object has an inverse pointer back to the first. The system, maintaining the integrity of these pointers, supplies a safe way for handling 1-1, 1-n and n-m relations.

¹⁰Parametrized types are standard mechanisms for defining container classes and better reusable code. The ANSI X3J16 subcommittee, devoted to establishing standards for C++, has decreed that parametrized types will be part of the initial C++ standard.

Building an ObjectStore application has a facet not addressed in normal C++ applications: the generation of schema information as a set of C++ objects. The classes these objects are instances of are referred to as *MOP*: MetaObject Protocol, that provides an application with the functionalities necessary to manage its schema:

- Classes can be added or deleted.
- the inheritance capabilities of a class can be changed, i.e., the hierarchy of classes can be modified.
- attributes can be added or deleted. Reordering attributes is also a form of modification, due to changes in the underlying representation.
- methods can be added, deleted or have their definitions modified

Once the schema has been modified, instances of the changed classes have to be migrated in order to conform the new definitions. The values to be stored in the fields added or changed can be either determined by user-defined functions, referred to as *transformer functions*, or set to default values automatically. Moreover, in the case of the old type and new type being assignment compatible, the new field is initialized by assignment from the old value directly.

Migration actually consists of making a copy of the old object and then modifying this copy. Because of this, the system automatically modifies all the pointers to the changed instances so that they refer to the new modified instances. Once the transformation is completed, all the old instances are deleted.

ObjectStore allows also instances of a given class to be reclassified as instances of subclasses of the class itself. Again, it can be done either by the system or by user-defined functions, called *reclassification functions*. This is a special kind of schema evolution since there is instance migration without schema changes.

Persistence is orthogonal to classes, i.e. objects of any class can be either persistent or transient. Within an application, persistent objects are created using an overloaded function `new`, augmented with advanced clustering capabilities to control object storage. By default, it has to be specified the database that will contain the object, but it can be addressed a database segment, a single cluster or even a specific object near which the new object should be stored.

Once an object becomes persistent, it can be retrieved by means of *Database roots*, objects that associate persistent data with string labels. Thus finding a persistent object means looking for a string value. Since only few objects in a database have an associated *Database root*, the desired data is not often obtained directly, it may be necessary to follow a chain of pointers or even to query the retrieved object to extract one of its components.

Version management is based on the concepts of *Configuration* and *Workspace*. A *Configuration* groups together objects that are to be treated as a whole for versioning. As working with an object in a configuration means locking the entire *Configuration*, *Subconfigurations* can be implied, within the same application, to provide a different lock granularity depending on the specific purpose. *WorkSpaces* are working areas within which *Configurations* are handled and stored. In order to work on an object, a user has to check out the *Configuration* the object belongs to, obtaining a working copy, visible only within user's *WorkSpace*. This means that different users can contemporary work

on the same object (*Configuration*) since they actually handle a private copy and not the object itself. Changes become visible to the outside when user's copy is checked-in the parent *WorkSpace*. The new objects, however, do not overwrite the previous ones, but they simply generate a new version of the *Configuration* just checked-in. The management of the versioning tree, produced through check in/check out operations, is completely under user control: ObjectStore provides facilities to set the current version of a *Configuration* and allow the user to program how two parallel versions have to be merged into a new one. Basically ObjectStore does not impose any pre-determined policy but lets the users free to customize version management in accordance with their own needs.

Check-in/check-out mechanisms are often referred to as long transactions, provided by ObjectStore together with conventional and nested transactions.

One level transactions can be either read-only or write transactions, furthermore users can set a time-out on their lock requests or even use a lock probe before actually requiring an object.

Nested transactions must share the same type (write or read-only) with its parents. User-defined abortions of nested transactions roll data back to its previous consistent state, without causing the abortion of the parent transaction. On the contrary, if the system aborts a sub-transaction, all the transactions within which it is nested are aborted too. Undoing all the changes done by these transactions avoids the database being in an intermediate and inconsistent state.

ObjectStore does not supply an interactive query language, but query definitions are ordinary *C++* or *C* expression, that are parameters of specific methods evaluated against instances of the class *Collection*. Queries can be nested, i.e., the query string can itself contain a query, but they cannot embody temporary variables and function calls.

3.5.1 ObjectStore for Smalltalk and Version 3.1

The first version of ObjectStore for Smalltalk [33] will be sold with the version 3.1 of ObjectStore [32].

Object Design has partnered with ParcPlace Systems to develop ObjectStore for Smalltalk: a complete database support for managing Smalltalk objects based on ObjectStore multi-client/multi-server architecture.

It will completely support ParcPlace Systems' VisualWorks graphical application development environment: a Smalltalk compiler, application development tools and extensive class libraries (over than 650 classes).

ObjectStore for Smalltalk provides a form of "transitive" persistence: at the transaction commit all the objects linked with persistent roots will become persistent.

The dynamic class definition capability of Smalltalk will be fully supported: programs will even be allowed to store class and methods objects within ObjectStore databases.

Both *C++* and Smalltalk data could be stored in the same database, and both Smalltalk and *C++* code could operate on persistent data defined by the other language using the facilities provided by the ParcPlace Systems' VisualWorks environment.

The goal of this first release is to completely support ParcPlace Systems' VisualWorks, providing the user with some limitations: it will not support version and configuration management facilities, as well as the full functionalities for schema evolution, relationships and queries.

ObjectStore version 3.1, on the contrary, will improve version management (schema evolution of versions) and supply a new product (ObjectStore/DBconnect) to develop mixed applications between object and relation databases. It will be possible to see a relational table as if it were a set of logical objects, facilitating migration to object technology.

4 Concluding Remarks

This paper illustrates the requirements that a PSEE (SPADE) and, in general, CASE applications pose to the supporting OODBMS, and describes how current object-oriented database technology fulfills such requirements.

<i>Requirements</i>	<i>O₂</i>	<i>GemStone</i>	<i>Ode</i>	<i>Object/DB</i>	<i>ObjectStore</i>
<i>Architecture</i>	C/S	C/S	C/S	C/S	C/S
<i>Data Model</i>	O2/O2C	Smalltalk	C++	C++	C++
<i>Interfaces</i>	C C++	C C++	-	C C++	C C++ Smalltalk
<i>Transaction</i>	C	C	C	C	C N L
<i>Object Versioning</i>	No	No	Yes	Yes	Yes
<i>Type Versioning</i>	No	Yes	No	No	Yes
<i>Query Language</i>	OQL	by methods	CQL++	Object SQL++	by methods

C/S Client-Server; C Conventional Transactions; N Nested Transactions; L Long Transactions

Table 1: A Concise Overview Summary

OODBMSs proved mature enough, as summarized in Table 4, to satisfy the requirements previously described and there is a general consensus that they represent the most promising technology. However, even if a first set of requirements is met by all the systems, advanced features like object and type versioning and physical distribution of data need further development.

Finally, since the aim of this paper is only to give an overview of the current capabilities of the OODBMSs to increase their use in the software engineering community, no judgments of classifications conclude this work. It is up to interested readers to properly read the information in Table 4 and formulate their own conclusions according to their specific needs.

Acknowledgments

Prof. Carlo Ghezzi and prof. Licia Sbattella for their guidance role and for their suggestions. Ing. Luigi Lavazza for his comments on the first drafts of this work.

References

- [1] S. Ahmed, A. Wong, D. Sriram and R. Logcher. Object-oriented database management systems for engineering: A comparison. *Journal of Object-Oriented Programming*, June 1992.

- [2] AT&T. Ode 3.X User Manual. version 3.0.3.
- [3] S. Bandinelli, L. Baresi, A. Fuggetta and L. Lavazza. Requirements and Early Experiences in the Implementation of the SPADE Repository using Object-Oriented Technology. In International Symposium on Object Technologies for Advanced Software, Kanazawa (Japan), November 1993. JSSST, Springer Verlag. Lecture Notes on Computer Science n. 742.
- [4] S. Bandinelli, L. Baresi, A. Fuggetta and L. Lavazza. Experiences in the Implementation of a Process-Centered Software Engineering Environment using Object-Oriented Technology. September 1994. submitted to Theory and Practice of Object Systems.
- [5] S. Bandinelli and A. Fuggetta. Computational Reflection in Software Process Modeling: The SLANG Approach. In Proceedings of the 15th International Conference on Software engineering, Baltimore, (USA), May 1993. IEEE.
- [6] S. Bandinelli, A. Fuggetta, C. Ghezzi and L. Lavazza. The SLANG 1.0 Process Modeling Language Reference Manual. Technical Report RT93032, CEFRIEL, Via Emanuelli, 15 - 20126 Milano (Italy), September 1993.
- [7] S. Bandinelli, A. Fuggetta and C. Ghezzi. Software Process Model Evolution in the SPADE Environment. IEEE Transactions on Software Engineering. Special Issue on Process Evolution, December 1993.
- [8] P.A. Bernstein. Database system support for software engineering—an extended abstract. In Proceedings of the Ninth International Conference on Software Engineering, pages 166–168. IEEE, 1987.
- [9] E. Bertino and L.D. Martino. Sistemi di Basi di Dati Orientate agli Oggetti. Addison-Wesley Masson. In italian.
- [10] E. Bertino, M. Negri, G. Pelagatti and L. Sbattella. Object-Oriented Query Languages: The Notion and the Issues. IEEE Transactions on Knowledge and Data Engineering, 4(3), June 1992.
- [11] P. Butterworth, A. Otis and J. Stein. The GemStone Object Database Management System. Communications of the ACM, 34(10), October 1991.
- [12] R.G.G. Cattell. What are Next-Generation Database Systems? Communications of the ACM, 34(10), October 1991.
- [13] C. Collet, P. Habraken, T. Coupaye and M. Adiba. Active rules for the GOODSTEP Software engineering platform. In Proceedings of the 2nd International Workshop on Database and Software engineering - 16th international conference on Software Engineering, Sorrento, Italy, May 16-17 1994.
- [14] DEC. DEC Object/DB, System Overview. Digital Equipment Corporation, 1992.
- [15] O. Deux. the O_2 System. Communications of the ACM, 34(10), October 1991.
- [16] S. Dewal, W. Emmerich and K. Lichtinghagen. A Decision Support Method for the Selection of OMSs. In Proceedings of the Second Int. Conference on System Integration, pages 32–40, Morristown, N.J., 1992. IEEE Computer Society Press.

- [17] S. Dissmann, W. Emmerich, B. Holtkamp, K. Lichtinghagen and L. Shope. OMSs comparative study. Internal Report D2.4.3-rep-1.0-UDO-EL, ATMOSPHERE, 1991.
- [18] R.G.G. Cattell (Ed.). The Object Database standard. Morgan Kaufmann, 1993.
- [19] W. Emmerich, W. Schäfer and J. Welsh. Suitable Databases For Process-centred Environments Do Not Yet Exist. In Jean-Claude Derniame, editor, Proceedings of the Second European Workshop on Software Process Technology, volume 635 of LNCS, pages 94–98, Trondheim (Norway), September 1992. Springer-Verlag.
- [20] F. Ferrandina, T. Meyer and R. Zicari. Implementing Lazy Database Updates for an Object Database System. Technical Report 9, GoodStep, March 1994.
- [21] N.H. Gehani and R. Agrawal. Ode (Object Database and Environment): The Language and the Data Model. In Proceedings of ACM-SIGMOD 1989 International Conference Management of Data, Portland, Oregon, May-June 1989.
- [22] N.H. Gehani, R. Agrawal and J. Srinivasan. OdeView: The Graphical Interface to Ode. In Proceedings of ACM-SIGMOD 1990 International Conference Management of Data, 1990.
- [23] N.H. Gehani, S. Dar and H.V. Jagadish. CQL++: An SQL for a C++ Based Object-Oriented DBMS. In Proceedings of International Conference on Extending Database Technology, Vienna (Austria), 1992.
- [24] N.H. Gehani, H.V. Jagadish and W.D. Roome. OdeFS: A File System Interface to an Object-Oriented Database. In Proceedings of the 20th VLDB Conference, Santiago (Chile), 1994.
- [25] C. Ghezzi, D. Mandrioli, S. Morasca and M. Pezzè. A Unified High-level Petri Net Formalism for Time-critical Systems. IEEE Transactions on Software Engineering, February 1991.
- [26] C.W. Lamb, G. Landis, J.A. Orestein and D.L. Weinreb. ObjectStore. Communications of the ACM, 34(10), October 1991.
- [27] S. Leggio. Progetto SPADE: studio di fattibilità per l'implementazione mediante strumenti software diversi. Tesi di laurea, Politecnico di Milano, June 1994. In italian.
- [28] F. Bancilhon M. Atkinson and ... The Object-oriented Database System Manifesto. In In Proceedings of the First DOOD Conference. japan, 1989.
- [29] O2 Technology. The O2 User Manual, 1994. version 4.5.
- [30] O2 Technology. Object Query Language: OQL Manual, june 1994. version 4.5 - beta.
- [31] Object Design. ObjectStore User Guide. version 3.0.
- [32] Object Design. An Object Design Product Brief: ObjectStore Product Update Release 3.1, 1994.

- [33] Object Design. An Object Design Technical Brief: ObjectStore for Smalltalk, March 1994. version 1.0.
- [34] Object Design. ObjectStore: Technical Overview, 1994. version 3.0.
- [35] M. H. Penedo and C. Shu. Acquiring experiences with the modelling and implementation of the project life-cycle process: the PMDB work. *Software Engineering Journal*, pages 259–273, September 1991.
- [36] C. Santos. Design and implementation of an object-oriented view mechanism. Technical Report 6, GoodStep, March 1994.
- [37] C. Santos, S. Abiteboul and C. Delobel. Virtual Schemas and Bases. In *Proceedings of the EDBT (Extending Database Technology) Conference*, 1994.
- [38] Servio Corporation. GemStone Programming Guide, june 1994. version 4.0.
- [39] V. Soloviev. An Overview of Three Commercial Object-Oriented Database Management Systems: ONTOS, OBjectStore and O_2 . *SIGMOD RECORD*, 21(1), March 1992.
- [40] The GoodStep Team. Description of software engineering applications and requirements for an object-oriented repository. Deliverable 1, ESPRIT project 6115 GoodStep - General Object-Oriented Databases for Software Processes, March 1993.
- [41] The GoodStep Team. The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes. In *Proceedings of APSEC'94, the First Asia-Pacific Software Engineering Conference*, Tokyo, December 1994.
- [42] S. Torcello. I requisiti per un DBMS di supporto ad un PSEE, ed esperienze con alcuni OODBMS commerciali. Tesi di laurea, Università degli Studi di Milano, October 1993. In italian.