Migration in Object-Oriented Database Systems - A Practical Approach*

C. Huemer†, G. Kappel‡, S. Vieweg†

† *Institute of Applied Computer Science and Information Systems, Department of Information Engineering, University of Vienna. Liebigg. 4, A-1010 Vienna, Austria, {ch,sv}@ifs.univie.ac.at*

‡ *Institute of Computer Science, Department of Information Systems, University of Linz, Altenbergerstr. 69, A-4040 Linz, Austria, gerti@ifs.uni-linz.ac.at*

SUMMARY

Object-oriented database systems are designed to meet the requirements of advanced database applications such as computer-integrated manufacturing. These requirements may evolve in the course of time and may require the migration of the database application from one object-oriented database system to another. Traditional migration approaches for relational database systems fail when applied to object-oriented database systems. The goal of this paper is to fill this gap. The paper describes a framework for the migration of object-oriented database applications. Our approach is based on a detailed analysis of the involved database systems, of the application's database requirements, and of the resources available for the database migration. We illustrate our framework by means of a case study, which is migrating an electronic planning board system from the object-oriented database system ONTOS to ObjectStore.

Key Words: object-oriented database systems, migration, electronic planning board application

INTRODUCTION

Advanced engineering applications such as computer-aided design and computer integrated manufacturing have emerged over the past decade. The database requirements of engineering applications extend those of traditional database applications and include complex

* The support by FFF (Austrian Foundation for Research Applied to Industry) under grant No. 2/279 is gratefully acknowledged.

object modeling, version management support and long transactions, to mention just a few. Object-oriented database systems (OODBS) are designed to meet these demands and bring together traditional database functionality and object-oriented concepts. An OODBS is a database system (DBS) based on an object-oriented data model [1, 2]. Since there does not exist a single object-oriented data model, an exchange of an OODBS might cause severe changes to the applications involved. Moving an application from one database system to another is commonly known as database migration.

Database migration is the process of mapping a database application from a *source DBS* to a *target DBS*. The migration process consists of a set of conversion operations or conversion techniques that are applied to the source database application and result in a target database application. The application migrated to the target DBS is constrained to deliver 'equivalent' results. Equivalence in this context is informally defined by the application's semantics.

Current approaches to the migration of database applications focus on the migration from hierarchical to relational DBS or from applications without database support to applications based on relational database systems [3, 4, 5]. The migration techniques for relational database applications are based on distinguishing two parts in a database application: a *database interaction* part and a *computation* part. The database interaction part is usually formulated in the high-level data definition and manipulation language (DDL/DML) SQL and is embedded in some program written in a socalled host programming language such as C. The computation part is programmed in the host programming language. Approaches to the migration of relational database applications benefit from this separation. The database queries are isolated and a conversion technique is applied. The computation part of the application remains unchanged. However, the separation into database interaction and computation entails also disadvantages known as 'impedance mismatch' [2], such as different data types, declarative versus procedural language concepts, and set-at-a-time versus record-at-a-time processing. Object-oriented database systems are designed to overcome these limitations. There should not be a semantic gap between the DDL/DML on the one hand and the host programming language on the other hand. This can be achieved either by incorporating database functionality into existing object-oriented programming languages, e.g., persistent

Smalltalk, or by extending traditional database languages with object-oriented concepts, e.g., object-oriented SQL. In any of these cases the degree of integration between the programming language and the DDL/DML increases. Consequently, the distinction between database interaction part and computation part is blurred and traditional approaches to migrating database applications fail. Approaches to migrating OODBS applications face different problems and are still missing in literature. The purpose of this paper is to fill this gap. We introduce a framework for migrating object-oriented database applications between different OODBS.

The relevance of the work is due to the state of art of OODBS technology. Firstly, there does not yet exist a unique data model for OODBS. Although there are similarities between the data models each OODBS uses its own approach. This fact and the fact that the database interaction part can not be clearly separated from the computation part as explained above imply that the exchange of the underlying OODBS of an application is costly and may cause considerable changes in the application. Note, that there is a standard object-oriented data model under discussion [6]. However, the implementation of this standard in the various OODBS as well as the adherence to the standard by the applications will take some time, and thus will not replace migration efforts in the near future. Secondly, the commercial OODBS market is currently booming. Several systems are competing for market leadership, and it is not yet decided which of the systems will survive. Thus, in order to achieve greater flexibility in the choice of the underlying OODBS as well as with respect to extensibility, the feasibility of migrating applications between different OODBS must be a prerequisite for deciding on one or the other of the existing OODBS. Lastly, OODBS applications are sold to different customers who may own different OODBS. Thus, to make an OODBS application more competitive the application should either support various OODBS or it should be easy to be migrated between various OODBS.

Two approaches to migration are possible: shallow migration and deep migration [7]. With *shallow migration* we denote the task of re-engineering the source database application on the target OODBS, whereas *deep migration* focuses on the re-development of the application in order to exploit the whole functionality of the target OODBS. Shallow migration

requires a re-design of only those database components of the application that have different semantics in the source OODBS and the target OODBS or that are missing in the target OODBS and, hence, have to be simulated. In our migration framework we consider both migration techniques.

The criteria for selecting one of the approaches can be grouped into two categories: functional requirements and organizational restrictions. Additional functionality of information services is vital for any enterprise. A decision whether to use or not to use advanced functionality is based on its contribution to the corporate goals and hence, must be included in the decision model. Furthermore, organizational restrictions such as limited availability of personnel must be considered. In our case study, based on the migration framework presented below, we opted for the shallow migration for two reasons. Firstly, the main goal of the case study was to investigate the applicability of several OODBS for the application at hand, rather than to re-develop the application totally every time to take full advantage of all features of the target OODBS. Secondly, we had to perform the case study under limited personnel and time resources.

Object-oriented database systems have a short research and development history compared to relational database systems. Nevertheless, due to the amount and variety of OODBS in the marketplace, migration strategies for OODBS applications are already badly needed. To develop migration strategies properly we have to have a closer look at the features of an OODBS. These are basically the same as for any other database system, but with a unique meaning in the OODBS context. Table 1 gives a brief overview of the features of OODBS providing both a general description of each DBS feature and a specific OODBS description. Due to its influence on the understanding what OODBS really are we investigate the data model feature more closely in the following. An object-oriented database system is a database system with an *object-oriented data model*. At present, there exist several different object-oriented data models [8], and thus, different data definition and manipulation languages (DDL/DML). They are based either on object-oriented extensions of SQL, on existing object-oriented languages like C++ (e.g., ONTOS[9] and ObjectStore[9, 10]) and Smalltalk (e.g., Gemstone[11]), or they were newly developed such as the O_2 data model[9, 12.] However, there is

consensus that a data model to be called object-oriented has to exhibit core features such as complex object modeling, object identity, encapsulation, types, inheritance and overriding with late binding [2]. The features presented in Table 1 have also been used as top level evaluation schema for evaluating OODBS in Reference 13 and they have been demonstrated to be a good starting point for evaluation in that context. A indepth discussion and comparison of OODBS is found in References 2, 9, 14, 15, 16 and 17.

The work reported in this paper was part of the ESPRIT project KBL (ESPRIT No. 5161[18, 19]), whose goal was the design and development of KBL - a *K*nowledge *B*ased *L*eitstand. “Leitstand” is a german term referring to an electronic planning board system for short-term production scheduling and control [20]. In a computer-integrated manufacturing (CIM) system a Leitstand is located between the production planning and control system and the shop floor control system. For a better understanding we use the term “electronic planning board” in the rest of the paper. In the KBL project the authors were responsible for incorporating object-oriented database technology as underlying information store, and as integrative component between the electronic planning board and various other CIM components.

A FRAMEWORK FOR MIGRATING APPLICATIONS BETWEEN OODBS

In this section we present a general outline for migrating applications between OODBS. The proposed framework describes the phases that have to be performed when migrating a database application from a source OODBS to a target OODBS. The framework informally describes what has to be done in each migration phase and it defines a useful order of the steps. The framework helps to identify those database features that have to be considered for the migration leading to an implementation of the necessary code changes. For this purpose, we have to look at all the components involved in the migration process: the *source OODBS*, the *target OODBS*, the *system configuration* and the *database application*. A careful analysis of these components is absolutely necessary to gain insight into the application's database requirements and to explore how these requirements are met by each of the involved OODBS. The database features required by the application and the features supported by the OODBS should be defined within the same set of criteria in order to be comparable. Within the proposed framework, these criteria are taken from the evaluation schema of Reference 13 presented in Table 1.

The careful selection of the set of criteria as an underlying basis for the migration is a prerequisite for all further investigations, since all subtasks of the migration process operate on this set. These subtasks towards a successful migration may be grouped into four steps and ten substeps, which are shown in Table 2. Their input and output dependencies are depicted in Figure 1. The substeps are described in more detail below.

The **analysis of the system configuration** is of general interest. Since the source OODBS and the target OODBS are evaluated for a specific hardware and software environment, the system configuration implicitly influences the evaluation of the OODBS involved. This implies that, for example, the evaluation results of ObjectStore for the UNIX operating system might be different from the evaluation results of ObjectStore for MS-DOS. Thus, the analysis of the system configuration is an integrative component of the OODBS evaluation.

The task of the **analysis of the OODBS** is to evaluate, classify and compare the features of the two OODBS involved in the migration. The evaluation of the OODBS leads to a set of features supported by each system. Furthermore, a comparison of these features results in three subsets according to their effect on the migration. The three subsets are the result of mapping the source OODBS to the target OODBS (cf. also Figure 2):

- **Overlapping features** are supported by both OODBS in the same manner and have no effects on the migration. Note, that two features might be syntactically similar or identical but semantically different. Only those features that are syntactically and semantically identical cause no problems for the migration and are included in the set of overlapping features.
- **Candidate shallow migration features** are supported only by the source OODBS or they are implemented differently in the two OODBS and thus, have to be simulated in the target OODBS. These features have to be considered in further detail to provide the same database functionality as offered by the source OODBS (shallow migration).
- **Candidate deep migration features** are only supported by the target OODBS but not by the source OODBS and hence, would extend the database functionality if considered in the migration process (deep migration).

Due to the fact that there exist some interdependencies between different features, changes of the application code required by migrating one feature might cause changes when migrating another. Since related changes should be considered together we propose to cluster features that require associated changes. As an example, consider that differences in the storage management of the OODBS might require changes in the persistent class definitions. In this case the features *persistence* and *storage management* should reside in the same cluster. Such a cluster is called **cluster of interdependent functionalities**. The clusters of interdependent functionalities (CIF) relate the migration features in order to improve the development of a strategy for the implementation process. Each of these resulting clusters forms a separate migration unit, which can be migrated independently from the features outside of this specific cluster. The membership in a cluster is independent of the member-

ship in the set of shallow or deep migration features. Membership in one of the two sets of migration features is determined by the semantics of the features provided by the source and the target OODBS, whereas membership in a CIF is due to code dependencies that must be considered during the implementation of the migration. Note, that the migration of a deep migration feature affects the shallow migration features in the same CIF, which are migrated in any case. However, the opposite is not true, since the decision to migrate deep migration features is independent of their membership in a CIF and is solely based on the decision to benefit from the enhanced functionality of the target OODBS.

A careful **analysis of the application's database requirements** considering the data model requirements, querying and manipulation requirements, and integration requirements is of great importance. It is a prerequisite to be able to select those database features that have to be considered during the course of migration. The features that are necessary for the application are collected in the set of database requirements (cf. Figure 2).

During **migration analysis** we compare these database requirements with the result of the DBS evaluation, which is represented by the three different sets of features as outlined above. The main task of **shallow migration analysis** is to find the intersection of the database requirements with the candidate shallow migration features (cf. Figure 2). Equivalently, the main task of the **deep migration analysis** is to find the intersection of the database requirements with the candidate deep migration features. Those features of the candidate shallow migration features and the candidate deep migration features that are not required by the application do not have to be considered anymore. Since the overlapping features are provided by both OODBS in the same manner we do not have to develop a mapping strategy for those and thus, we do not have to investigate them in the following steps of the migration process. On the one hand, the result of the migration analysis is the set of shallow migration features, which has to be migrated to obtain the same database functionality for the application as it had been available by the source OODBS. On the other hand, the result is the set of deep migration features, which extend the database functionality because they are not provided by the source OODBS but are required by the application.

One might argue that starting the migration process with the analysis of the applications' database requirements and continuing the evaluation of the OODBS only with those features that are part of these requirements might reduce the analysis effort. This argument appears reasonable since in our proposal we analyze also features that are not of primary interest to the application. Yet, we selected this sequence of substeps for reasons of completeness and to reduce the maintenance effort in case of future re-design projects. Firstly, it might reduce the total effort when migrating other applications between the same OODBS. And secondly, our approach has advantages in the case of future changes in the application's database requirements.

For the features residing in the set of shallow migration features and in the set of deep migration features a **strategy for their implementation** into the target OODBS must be developed. According to the two different kinds of migration features, we can distinguish between a **mapping strategy for shallow migration** and a **mapping strategy for deep migration**. By developing an implementation strategy one might realize that the effort to implement a specific feature exceeds the semantic gain provided by that feature. Whether or not a certain feature is migrated has to be decided on a case by case basis and depends on the available resources. The result of this step is a mapping strategy for all those features that are included in the implementation of the migration.

The **implementation** step includes the coding of the necessary changes and furthermore the evaluation of the target application, i.e., the application running on top of the target OODBS. The programmer is responsible for **re-engineering the code** according to the developed mapping strategies leading to the target application. The last substep comprises the **evaluation** of the target application to verify if the desired database functionality is provided by the target application.

CASE STUDY: MIGRATING KBL FROM ONTOS TO OBJECTSTORE

In this section we describe the migration of the KBL application from ONTOS to ObjectStore. The main reason for studying migration in the realm of KBL, namely flexibility in terms of varying underlying OODBS, has already been discussed in the introduction. The main reason for choosing ONTOS in the first place has been its support for flexibility (see next section “A Brief Tour of ONTOS and ObjectStore”). ObjectStore was chosen as target OODBS due to a decision of our major industrial partners. The migration process is based on the framework introduced in the previous section and demonstrates the applicability of our approach. We first illustrate the functionality of the electronic planning board KBL. We then shortly present ONTOS and ObjectStore as the source OODBS and the target OODBS, respectively. Finally, we describe the migration process of KBL.

KBL - A Knowledge-Based Electronic Planning Board System

KBL was developed under the object-oriented paradigm and was implemented on top of the OODBS ONTOS. The intention of this subsection is to give an overview of the electronic planning board system KBL.A further description of KBL is found in References 18 and 19.

An electronic planning board is a computer aided graphical decision support system for interactive production scheduling. It interacts with the production planning and control system on the one hand, and with the shop floor control system on the other hand. Figure 3 depicts the integration of an electronic planning board into the manufacturing process.

At the production planning and control level a mean-term planning of products and involved resources in the manufacturing process is carried out. As a result the master production schedule is used to co-ordinate related business services such as engineering, manufacturing, and finance. A further step called material requirements planning is used to determine the actual production requirements for a set of work orders. The work orders from MRP are then co-ordinated with the production facilities available. Due to changes in the availability of resources, like workforce and machine breakdowns, the capacity planning is a very complex task. Splitting of lots, lead times and alternative routings are only some of the arising addi-

tional problems. When capacity limitations have been compensated, the orders are released for production. In order to cope with unforeseen circumstances in production, like machine breakdowns and shortage of materials, the production process has to be supervised and controlled. Shop floor control manages the production process and keeps track of the inconsistencies between the production plan and the actual production. A shop floor control system requires information about the current location of parts, tools, and operators, the estimated completion time, and remaining operations. This information must be kept on-line and gathered through shop floor data collection terminals connected with each production facility or workcell. Highly integrated with the shop floor control is the shop floor scheduling. Shop floor schedulers or electronic planning board systems assign work orders to the appropriate workcells [20]. The scheduling process has to be performed under certain optimization criteria. The manufacturing throughput time has to be minimized, due dates must be satisfied, and the utilization of resources must be optimized, to mention just a few. Due to the complex task of multilevel optimization this is performed by heuristics at a highly interactive level. The system is used as an electronic planning board and the operator manipulates Gantt diagrams representing work orders and resource data. Data collection systems at the shop floor level for reporting machine breakdowns and order completion extend an electronic planning board to a highly flexible scheduling tool.

KBL follows the functionality described above and is made up of the following components:

- *Knowledge Representation and Acquisition*: The information relevant for scheduling and control needs to be represented in a flexible manner. It must support besides others the representation of scheduling heuristics, scheduling evaluation functions and shop floor monitoring tools.
- *Scheduling Control*: The Scheduling Control subsystem represents the interface between the application and the DBS. It provides DBS functionality, interfaces to other application processes and basic scheduling routines.
- *Simulation*: Various schedules can be simulated in order to evaluate the performance of schedules under different constraints.

- *Interactive Advisor*: KBL is equipped with an Interactive Advisor, which constantly analyzes the status of the whole system. It provides the production scheduling personnel with scheduling data and advice for alternate actions.

- *Evaluation*: The Evaluation Component allows the assessment of different scheduling strategies. A detailed analysis based on built-in evaluation functions and user-defined evaluation criteria should help to improve the quality of the scheduling process.

- *Communication Interfaces*: Communication interfaces to the production planning and control system as well as to the shop floor control system are supported.

Based on the functionality described above *KBL's database requirements* can be identified as the following [21, 22]: advanced data models, meta data access, navigational and associative access, version management of schedules, and distributed data processing. *Advanced data models* and *meta data management* are essential for modeling the scheduling information in the knowledge base. Given the tight integration of the knowledge base and the scheduling control a planning board system must provide both *navigational access* of highly interrelated data and *associative access* by querying collections of data. *Version management* is required for simulating different schedules of work orders. Since an electronic planning board system is integrated with the production planning and control system and the shop floor control system *distributed data processing* plays an important role. However, in the current prototype of KBL it was not considered a main requirement on the critical path and thus, abandoned from the list of database requirements. When we refine these requirements we come up with a list of relevant database features to be evaluated for KBL as given in Table 3. These features serve as the basis for the migration framework and must be considered in detail during the migration process.

The current implementation of KBL contains the *Knowledge Representation and Acquisition* component, the *Scheduling Control* component, and the *Simulation* component. The *Interactive Advisor,* the *Evaluation*, and the *Communication Interface* are not included in the prototype implementation. KBL was implemented on top of the object-oriented DBS ONTOS in the first place. The implemented prototype is structured into the following three

modules: Scheduling Toolkit Module, Planning Board Module and Simulation Module.

The *Scheduling Toolkit Module* represents the core of the KBL system. It includes all classes and methods necessary for the scheduling of orders. It contains all production data and manages the constraints and capacity models. Furthermore, it controls access to the database. The Knowledge Representation and Acquisition component, and the Scheduling Control component are implemented by the Scheduling Toolkit Module. The *Planning Board Module* implements the graphical user interface. The *Simulation Module* implements the Simulation component and allows to schedule orders automatically. All the modules use the Scheduling Toolkit Module for the management of persistent objects in the database. The KBL database schema follows the class hierarchy depicted in Figure 4. The basic functionality of the Scheduling Toolkit is implemented in the classes Agent, Activity, and STResource (highlighted in Figure 4). The class Activity is used to describe operations in the planning board environment. These include the manufacturing specific operations such as 'drilling', 'milling' etc. The class STResource describes any kind of resources, such as materials, machines and workers. Instances of the class Agent are used to relate activities and resources and contain additional information about the status of the resource/activity relationship. This includes whether the activities have been already scheduled or not, and whether the activities consume or produce resources and to which rate.

In conjunction with other classes such as TimeStateDescriptor, IntervalObject, and AgentConstraint the classes described above represent a flexible environment for modeling manufacturing processes. A detailed account of the KBL class hierarchy can be found in Reference 19.

A Brief Tour of ONTOS and ObjectStore

In the following we briefly describe the main features of the OODBS ONTOS and ObjectStore. For further information we point to References 9, 10, 13 and 14 and to the product literature. Based on our evaluation schema for OODBS we group the evaluation features of ONTOS and ObjectStore as depicted in Table 4.

ONTOS Release 2.2 is based on C++, which implies that the DDL/DML is basically C++. According to our classification of constructing OODBS in the introduction, in ONTOS database functionality has been incorporated into the existing object-oriented language C++. It operates in a client/server environment, where the server architecture relies on the page server paradigm. It is available on the major workstation platforms. The strength of the product lies in its extensibility and in the flexible meta data management. The storage manager and the transaction manager can be modified in order to support user-defined extensions. Databases can be accessed either with C++ as DDL/DML or with an interactive SQL-Interface. The objects are accessed and referenced through indirect references called logical object references or through direct references physical object references. The access to meta data is fully supported. The dynamic creation of new classes and methods provides a high degree of flexibility. Persistence is reached by inheritance from a system supplied preexisting object class. Each persistent class requires the implementation of several methods (get_direct_type, put_object, APL-Constructor, delete_object) in order to guarantee consistent management of persistent objects. ONTOS implements object level locking and provides transactions with checkpointing. A transaction may access a single database server. Version management is not supported.

Similar to ONTOS, ObjectStore Release 2.0 is also based on C++ and operates in a client/server environment. The ObjectStore server is a page server. It is also available on the major workstation platforms. It provides access to the database either with C++ or with a C++ extension called ObjectStore DML: Both C++ and ObjectStore DML can be regarded as DDL/DML of ObjectStore. An SQL-like query language is not supported. The strength of ObjectStore is its memory architecture and the resulting performance benefits. Objects are mainly accessed via direct references, although indirect references are also supported. The use of direct references implies some restrictions on the size of the database, on the amount of data that is accessible within a single transaction, and on database reorganization, however, the advantages of direct references dominate their weaknesses for certain applications. Persistence is orthogonal to the type system, i.e., independent of any preexisting object class, and thus provides advantages in case of migrating applications to ObjectStore

(for an in-depth discussion of this point see section 4). ObjectStore provides navigational access via object references and associative access via queries over collections. In terms of extensibility and schema access, ObjectStore does not provide the flexibility of ONTOS. The meta object protocol (MOP) only provides access to class descriptions. Dynamic modifications of class descriptions are not supported. Static schema evolution is supported via an object migration tool converting the instances of the old schema to conform to the new schema. ObjectStore provides a sophisticated versioning mechanism that supports the versioning of configurations of objects. Concurrent database access is controlled by implicit page level locking and closed nested transactions.

With the discussion of the database requirements of KBL and the evaluation of the involved OODBS we have completed the first phase of the migration process and are ready to describe the migration analysis.

Migration Analysis

In this subsection we discuss the second step of the migration process, namely the migration analysis for migrating KBL from ONTOS to ObjectStore. Based on the evaluation schema of Reference 13, which is discussed in Table 1 and applied to ONTOS and ObjectStore in Table 4, we investigate the second step of the migration process. The input and output of the second step is presented in Figure 5 and 6, respectively. Figure 5 shows the flow of each feature through the first two steps of the migration process. Figure 6 depicts the intersections of the database requirements and the features supported by ONTOS and ObjectStore. It's beyond the scope of this paper to discuss mapping strategies for all migration features. Instead, after discussing the grouping of the features depicted in Figure 5 the mapping strategies for example features taken from each of the migration paths are introduced.

Both ONTOS and ObjectStore are based on the C++ **data model**. They both use the C++ basic data types in addition to some complex data types, e.g., to support various kinds of collections. Furthermore, in both OODBS the C++ data model is extended to provide typical database features such as transaction support. Concerning basic data types, ONTOS and ObjectStore have overlapping functionality (see also Section 'Basic Data Types').

Concerning complex data types and database extensions, each data model uses its own syntax and semantics. Thus, *data model* is included in the candidate shallow migration features. Since the functionality of the data model is crucial for the KBL application the data model is part of the application's database requirements, and thus, it is included in the shallow migration features, for which a strategy must be developed.

The feature **constraints and triggers** is not supported by ONTOS. Since we classify inverse relationships as a kind of constraints and since ObjectStore supports inverse relationships, we add this feature to the candidate deep migration features. Note, ONTOS offers inverse relationships only for dynamically created types but not for statically created ones. This feature is also included in KBL's database requirements since some objects in KBL are related to each other by an inverse relationship. As a consequence, we add constraints and triggers to the set of deep migration features. The mapping strategy for inverse relationships are presented in Section 'Inverse Relationships'.

Persistence is required by any database application. In ONTOS it is implemented by inheritance and in ObjectStore by declaration. Thus, we have to include persistence in the shallow migration features and have to develop a mapping strategy. For a detailed account thereof see Section 'Persistence' and Section 'Evaluation Report'.

In ONTOS, it is possible to provide each object with a synonym, which is stored in a separate **data dictionary**. The synonym can serve as unique identifier for the specific object. In ObjectStore, nothing similar exists. Therefore, the feature is part of the candidate shallow migration features. In KBL, synonyms are heavily used for the activation of objects and thus, this feature becomes a shallow migration feature.

The analysis of the **tools** is only interesting in terms of supporting the implementation process but it has no effects on the application itself. The feature has been added to the candidate deep migration features, but it does not reside in any of the resulting sets.

Query management is implemented differently in both systems. In ONTOS, persistent objects are retrieved via their synonyms or via an instance iterator over all persistent objects

that belong to a specific class and its subclasses. In ObjectStore, the entry points of the database are persistent root objects. Due to these different access methods and since KBL, like any other database application, requires query management, it is included in the set of shallow migration features.

Query optimization is also part of the database requirements of KBL. ONTOS provides only limited query optimization. ObjectStore supports indices and clustering for query optimization. Thus, query optimization is a member of the deep migration features.

Since C++ is the **host programming language** required by KBL and since both OODBS provide an interface to C++, this feature is part of the overlapping features.

ONTOS and ObjectStore provide different concepts for **schema evolution** but it was not considered a requirement in the KBL prototype implementation. Thus, schema evolution is a member of the candidate shallow migration features, only.

Both systems supply neither logical nor physical data independence, however, ObjectStore provides some object migration features. Thus, **change control** resides in the set of candidate deep migration features. Since schema evolution has not been regarded a requirement of the KBL prototype, change control is not regarded either.

ONTOS does not support a **versioning** mechanism. As a consequence, in KBL the versioning of the schedules has to be simulated. ObjectStore provides a sophisticated versioning mechanism including linear and branching versions. Since versioning is required by KBL but only supported by ObjectStore, it is part of the deep migration features.

Concerning **concurrency control**, ONTOS is superior to ObjectStore since in ONTOS it is possible to explicitly lock objects and to specify an optimistic locking strategy in addition to a pessimistic one. The optimistic lock strategy is also part of the database requirements of KBL and thus, included in the shallow migration features. Since the simulation of an optimistic lock strategy in ObjectStore would have gone far beyond our available resources, we restricted the migrated KBL application to the use of a pessimistic lock strategy.

Recovery is part of the database requirements of any application. It is included in the overlapping features since ONTOS and ObjectStore provide automatic database recovery from volatile storage but do not provide disk crash recovery.

Authorization is also included in the overlapping features since data access control is supported at the database level, both in ONTOS and in ObjectStore. Database access is controlled by the UNIX file access protocol. KBL does not require any specific access control mechanisms.

Both systems support a client/server **architecture**, which is required by KBL. Since ONTOS and ObjectStore are based on a page server we consider the architecture to be similar in both systems and to be part of the overlapping features.

One of the most important differences between ONTOS and ObjectStore in the realm of **storage management** is the disk to in-memory mapping and the activation of referenced objects. In addition, the facilities for the use of indices and clustering are different in both systems. Storage management is a shallow migration feature since the KBL prototype requires both the activation of referenced objects and the use of indices and clustering mechanisms.

The feature **distribution** is a candidate shallow migration feature because ONTOS and ObjectStore provide different concepts for distribution. It is not a shallow migration feature since the KBL prototype disregards distribution.

ONTOS does not provide any import and export **interfaces**. ObjectStore offers a third party tool to support an import interface from STEP/Express. Nevertheless, import and export interfaces are not required by KBL. Thus, interfaces are part of the candidate deep migration features but do not have to be considered for the migration of KBL.

KBL was developed for SUN workstations in a TCP/IP network. ONTOS and ObjectStore support this environment. Therefore, the feature **operational conditions** is included in the overlapping features.

Mapping Strategy for Selected Features

In this subsection we develop a mapping strategy for three selected features of the evaluation schema. These (sub)features are chosen in such a way that each of the three possible paths through the migration process is covered:

- shallow migration path: *persistence*
- deep migration path: *inverse relationships* as part of *constraints & triggers*
- overlapping path: *basic data types* as part of the *data model*

In the following, we present the mapping strategy for each of these features. To increase the understanding on how to use OODBS we briefly investigate the development of OODBS applications beforehand, both in general and based on the selected OODBS.

In general, the development of an object-oriented database application can be divided into the following phases: (a) development of the database schema and the database application, (b) registration of the schema in the OODBS, and (c) compilation and linking of the database application. Most of the commercial OODBS extend some object-oriented host programming language and generate the database schema during the compilation of the application. Thus, the development of the application is controlled by Makefile commands.

Table 5 and Table 6 show the relevant fragments of Makefiles for the compilation of the KBL application in ONTOS and in ObjectStore, respectively. Besides the typical contents of a Makefile such as compiling and linking instructions, they also include directives for the creation of the database schema. The database schema consisting of the C++ source and header files is generated during compilation and stored for subsequent access during the execution of the database application. In ObjectStore, the schema information is generated by the compiler and is stored in dedicated databases called Compilation Schema Database and Application Schema Database. ONTOS uses a database administration tool called DBATool for the creation and registration of database schemes. Furthermore, ONTOS

allows the assignment of physical disk space to databases via Makefile directives. In ObjectStore, this task is controlled by the file system.

The two approaches mainly differ in the flexibility of database creation. In ONTOS, databases must be generated and specified with the DBATool while in ObjectStore, the user is free to generate databases during run-time of the application. However, from the database administration point of view this approach lacks single-point control of database generation.

As mentioned above, the database schema is contained in C++ source and header files. In the following subsection, we describe the structure of the persistent class definitions in ONTOS and ObjectStore, respectively. Basically, the syntax is similar to C++ class definitions, however, each of the OODBS extends the C++ syntax in order to specify persistence.

Persistence

In this subsection we demonstrate the development of a mapping strategy for one of the most interesting features in the set of shallow migration features of Figure 5, namely *persistence*.

In ONTOS, persistence is achieved by inheritance from the ONTOS specific class `Object`. In ObjectStore, it is achieved by the declaration of persistent variables. For the development of a mapping strategy it is necessary to further investigate how persistence is implemented in ONTOS and ObjectStore, respectively. To achieve persistence in ONTOS, the following conditions must hold true:

- Classes must have an inheritance path through the ONTOS class `Object`.
- Classes must have a special constructor called “activation constructor” to activate an object from disk to cache memory.
- Classes must have a special member function called `getDirectType()`.
- If the class has a destructor, it should have a function called `Destroy()` to deactivate an object from main memory but not from disk.
- Classes should have the functions `putObject()` and `deleteObject()` to write / delete an object to / from the disk

If the definition of a class fulfills these requirements, and the member function putObject is invoked on an instance of this class, this specific instance is made persistent. In ObjectStore, the class definition for a persistent object includes neither an inheritance path through a specific predefined class nor specific functions similar to those in ONTOS. An object is made persistent by declaration. There is no need to call an operation like putObject to write it to secondary storage.

The code fragments presented in Table 7 summarize the persistent class definitions for the class ScheduleAgent in ONTOS and ObjectStore, respectively. The examples were taken from the KBL application and compare the definition of persistent classes and extension management.

As the persistent class definition is also influenced by the features *storage management*, *query management*, and *data dictionary*, we put *persistence* together with these features into a cluster of interdependent functionality (CIF) according to step 1 of the migration framework (see Table 2). Although *constraints & triggers* and *query optimization* are also included in this CIF, these features are omitted when performing a shallow migration.

The clustering with *storage management* is due to the fact that in ONTOS direct references are main memory pointers and behave like main memory pointers in every respect. This implies that the traversal of a direct reference requires the programmer to ensure that the referenced object is already in main memory. Otherwise the program will terminate with an exception raised in the best case, or continue with unexpected values, in the worst case. The other possibility is to use - as in KBL - indirect references via the class Reference. Reference allows objects to be referenced by using a format that is valid whether or not the referenced object is currently in main memory. The Binding() function defined for the class Reference returns a pointer to the referenced object and activates it if necessary. ObjectStore provides a very comfortable concept called ‘Virtual Memory Mapping Architecture’ for the activation of referenced objects. In ObjectStore, all pointers take the form of regular memory pointers, similar to direct references. A pointer to a persistent object that is currently not in main memory has an unmapped virtual-memory-address. In the case of

dereferencing the virtual-memory-pointer, a fault is signaled by the violation handler and the segment containing the object is transferred into the client's cache. The page containing the object is mapped into the virtual memory. ObjectStore provides also some kind of indirect references. They are mainly used for dereferencing objects in other databases and between transaction boundaries. Because of the convenience of using direct references in ObjectStore we decided to replace the indirect references in the ONTOS version of KBL by direct references in the ObjectStore version.

The clustering with *query management* stems from the fact that ONTOS offers a so called instance iterator as entry point to the database, which allows access to all objects that belong to a specific class and its subclasses. In order to simulate this functionality in ObjectStore, each persistent class includes a static persistent class variable named `extent` with domain `os_Set` containing all the instances of this class and of all subclasses, respectively. Therefore, the constructor has to include a call that inserts each newly created object into this static set, and the destructor must include a call that removes each deleted object from this set, equivalently.

The clustering with *data dictionary* is due to the fact that ONTOS provides synonyms for each persistent object. These synonyms serve as unique identifiers for the specific objects. The name spaces of the synonyms may be organized in hierarchies. This behavior is simulated in ObjectStore by embedding an identifying property (`char *ivName`) into the root class of the KBL application, `KBLObject` (cf. Figure 4 and Table 7).

In Table 8, we exemplify the use of the instance iterator in ONTOS and the corresponding simulation in ObjectStore. The example method `getScheduledSAList` operates on the set of `ScheduleAgent` objects that have been considered during the scheduling process and have already been scheduled on a specific resource. In ONTOS, it is possible to create an instance iterator for the class `ScheduleAgent` to iterate over all objects belonging to `ScheduleAgent`. Those members of the `ScheduleAgent` extent that have already been scheduled are inserted into a list, which is returned by the function `getScheduledSAList`. Since ObjectStore does not provide instance iterators we include the static persistent class variable `extent` in the

ScheduleAgent class definition containing all instances of the class ScheduleAgent. The selection of the scheduled ScheduleAgent objects is performed by querying the set ScheduleAgent::extent.

Inverse Relationships

In the following we present the mapping strategy of the subfeature *inverse relationships*, which is part of the evaluation feature *constraints and triggers*. Inverse relationships are included in the deep migration features, for which a mapping strategy has to be developed in case of deep migration. In ObjectStore, declarations of inverse relationships have to be made within the persistent class definition. Due to this fact, the feature *constraints and triggers* and the feature *persistence* must reside in the same cluster of interdependent functionalities.

Before presenting a strategy for the feature *inverse relationships*, we first define the concept of inverse relationships. An inverse relationship is a binary relationship between two objects or sets of objects being maintained in both directions. Accordingly, inverse relationships between database objects may be of arity 1:1, 1:n, or n:m. Usually, these relationships are implemented by references between the related objects or by the use of explicit relationship objects. The inverse relationships are automatically maintained according to the underlying semantics. Consider object A and object B to be related to each other by an inverse relationship. Whenever a reference is established from object A to object B, a reference from object B to object A must be also established. Conversely, when a reference from object A to object B is deleted, the reference from object B to object A has to be deleted in turn. ObjectStore allows the modelling of inverse relationships with pointer-valued or collection-of-pointer-valued instance variables, called data members in ObjectStore. The OODBS automatically maintains the integrity of the inverse relationship. Binary relationships of statically created types in ONTOS do not provide this semantics. In ONTOS, the programmer is responsible for maintaining the integrity of binary relationships by explicit calls within the application code.

As an example, consider the relationship between a manufacturing activity and the subactivities into which it can be divided. In KBL, activities such as drilling or melting are

structured in an activity hierarchy. This means that a root activity is built up by a set of subactivities, which can also be made up of subactivities and so on. Furthermore, each subactivity can be part of more than one superactivity. Therefore, each activity, except the root activity, has one or more superactivities and it might have several subactivities. There exists a n:m-relationship between the superactivities and the subactivities. In order to maintain the integrity it is necessary for each newly created activity to insert its superactivities into the set of superactivities, and subsequently to insert the activity itself into all its superactivities' sets of subactivities.

In the ONTOS implementation, the management of inverse relationships is explicitly implemented in the constructor using references between the related objects (cf. Table 9). Whenever an activity is created and the constructor gets executed the programmer must ensure that references are installed from the newly created object to its superactivities. Furthermore, the programmer is responsible for the installation of references from these superactivities to this newly created activity. In ObjectStore it is possible to define the set of superactivities to be related to the set of subactivities by an inverse relationship, called inverse members in ObjectStore (cf. Table 9). In this case the programmer has only to ensure that references are established from the newly created activity to its superactivities. The corresponding references from the superactivities to the newly created activity are automatically established and maintained by the database system. Thus, the strategy for a deep migration of the feature *inverse relationships* should be the following:

(a) Define an inverse relationship between two data members whenever there exists a binary relationship that should be automatically maintained.

(b) Remove the user-defined code segments that have been used to ensure integrity so far.

In the left column of Table 9, we present the example of the constructor of the class Activity described above. Note, that in the ONTOS application the set of superactivities and the set of subactivities are implemented as lists. ObjectStore supports inverse relationships with system-maintained references between the related objects. The declaration of inverse rela-

tionships affects each manipulation of these relationships. For example, whenever a superactivity is removed from an activity's set of superactivities, this activity is also automatically removed from the superactivity's set of subactivities. Thus, during migration implementation code changes have to be made to all procedures that operate on the inverse relationship between superactivities and subactivities, such as the destructor and the procedures that remove and add new superactivities.

Basic Data Types

As mentioned above, ONTOS and ObjectStore are based on the C++ data model. They both use the C++ basic data types and in addition some database system-specific complex data types. Since these complex data types are different in both OODBS, the main feature *data model* is included in the set of shallow migration features. Yet, the subfeature *basic data types* is an overlapping feature because both systems use the C++ basic data types. Therefore, calls that operate only on the C++ basic data types (e.g., `int foundSlotNum = 0;`) do not have to be changed during migration. As a consequence, we do not have to develop a mapping strategy for the (sub)feature basic data types.

EVALUATION REPORT

In this section we discuss the experiences gathered during the migration of the KBL application from ONTOS to ObjectStore. The purpose of the case study was to investigate the applicability of the migration framework described above. Note, that neither the implementation of the migration process was carried out in a production environment nor did we emphasize the optimization of the application.

We first describe our experiences with the involved OODBS - ONTOS and ObjectStore. We then present a qualitative evaluation of our migration framework including a discussion of our approach and an outlook on further scenarios that one may face in further migration projects. Finally, we present a quantitative evaluation of our case study including the source and target application as well as the involved personnel resources.

ONTOS and ObjectStore

In Section ‘A Brief Tour of ONTOS and ObjectStore’ a concise introduction of the functionality of the two OODBS has been given. We can conclude that both systems are based on the same DDL/DML, namely C++, they operate in a distributed workstation environment, and they provide basic database functionality such as transaction management, query management, and limited recovery. Due to the absence of user management facilities, security facilities, and ad-hoc querying facilities, both ONTOS and ObjectStore may be classified rather as persistent programming languages than as full-fledged database management systems.

The main difference between ONTOS and ObjectStore is their approach to provide persistence. ONTOS requires, firstly, that each persistent class inherits from a predefined system-specific class, and secondly, that additional methods for each persistent class are implemented. Thus, in ONTOS persistence is not transparent at all to the application programmer. In ObjectStore, persistence is reached by declaration of persistent variables. This means that in the latter case persistence and types are orthogonal in the sense that the feature persistence is realized without involvement of some predefined type, while in the former case this is not true. The main advantage of ObjectStore’s approach to provide persistence becomes

apparent when an application, having used no database system so far or having used another OODBS, is ported to ObjectStore. In this case, the application's class hierarchy doesn't have to be changed. In contrast, this would be necessary with ONTOS in the sense that an additional superclass, the predefined persistent class `Object`, has to be included in the list of superclasses that a specific class inherits from. This may cause different problems, ranging from name clashes due to multiple inheritance to a complete change of the application's class hierarchy if only single inheritance is supported. In our case study we did not face these problems, since we were migrating *from* ONTOS *to* ObjectStore.

In addition, ONTOS provides automatic management of class extensions while ObjectStore requires the implementation of user-defined containers to collect persistent objects. However, it can be easily implemented (see Table 7 and 8). Furthermore, we noticed a much tighter integration of ObjectStore with the C++ programming language than it is the case for ONTOS.

As mentioned above, we did not consider performance as a main issue in our case study. A comparison of the two systems based on the OO1 benchmark[23] and the OO7 benchmark[24] can be drawn from the benchmark results published by the respective database manufacturers.

Discussion of the Framework

The migration of database applications is part of the software maintenance process. The much tighter integration of host programming languages with object-oriented database systems is one of the main characteristics of OODBS and thus requires novel migration techniques. This was the starting point for developing the migration framework presented in the previous sections. The experiences we gained are the following. Firstly, the framework supports a structured thus observable approach to the migration problem by providing both shallow and deep migration. Due to this comprehensible approach it is easier to detect any missing link and thus it is less errorprone. Secondly, the framework reduces the effort for further migrations between OODBS because of the available analysis information of OODBS gained during the first step of the migration process. Lastly, the detailed evaluation

of OODBS shed some light on the intrinsics of object-oriented database systems and thus help to combine applications with OODBS in a more comprehensible way. However, these experiences have to be judged in the right context, i.e., only one case study has been carried out so far, and we are not aware of any other OODBS migration experience without using the presented framework. Migration analysis from non-relational to relational databases and from relational databases to object-oriented databases are available but we feel that they do not serve as a serious basis for a comparison.

In the following we discuss scenarios that were not directly covered in our case study, yet may be of concern when applying the presented framework.

What if ...

... *the database systems follow different approaches to reach persistence?*

The experiences mentioned above concern the migration of KBL from ONTOS to ObjectStore. As already stated, the architecture of ObjectStore supports the migration to ObjectStore, not only but also due to the fact that persistence is implemented by declaration. In addition, in the KBL case study the size of the application decreased considerably since several methods necessary to implement persistence in ONTOS could be deleted (see also next subsection on quantitative evaluation). Migrating applications in the reverse direction - from ObjectStore to ONTOS - is more difficult. Migrating from ObjectStore to ONTOS requires persistent classes to inherit from the ONTOS class `Object` and to implement additional methods (see Section ‘Persistence’). Consequently, the size of the application increases. In contrast, migrating applications between OODBS that follow a similar approach to reach persistence, e.g., ONTOS and Versant, requires less effort than the former case and has minor effects on the size of the application code.

... *the database systems provide different host programming languages and/or different DDL/DML?*

A host programming language is the language in which the application to access and manipulate the database is written. In contrast, the DDL/DML is the database language based on the concepts of the underlying data model. Our case study has shown that the migration

framework is well suited for the migration between OODBS supporting the same host programming language. If the set of host programming languages differs between the source DBS and the target DBS, more specifically, if the host programming language used by the database application is not supported by the target DBS a complete re-development of the application becomes necessary. In general, this is a very difficult and costly task. One would try to use tools like cross-compilers to make the translation process less errorprone. If the host programming language of the application is supported by the target DBS yet the DDL/DML of the source DBS and the target DBS are considerably different, major changes of the database interaction part of the application become necessary. Examples include the migrations between SMALLTALK-based database systems and C++-based database systems. Last but not least, if the host programming language of the application is supported by the target DBS and both DBS are based on similar yet not identical data models minor changes of the database interaction part of the application are necessary. The presented case study is an example thereof. If the DDL/DML differ only in minor respects such as different semantics of a basic data type, e.g., different ranges of floating point variables, the behavior of the relevant parts of the application can be simulated by wrapper functions or wrapper classes that mimic the appropriate behavior. In all three cases discussed above, the structured approach of the proposed framework helps to accomplish the migration.

... *the database systems are operated in different system environments?*

System downsizing from host-based computing to client/server computing may include changes of the operational conditions of the target database application. As a result the underlying operating system or user-interface system may change, e.g., from a X/Unix based to a Windows/MS-DOS based environment. These changes are reflected in the environment analysis of the migration framework; either as changes in the system configuration or as resulting changes in the database system’s functionality. Since our framework focuses on the database part of the application migration user interface specific or operating system specific migration techniques such as interface gateways may be applied but are not explicitly mentioned in our framework.

Quantitative Evaluation

In our quantitative evaluation we concentrate on two metrics: the *size of the application code* as a quantitative product metric and the *involved personnel* as a metric for man power involved in the migration process. Both the comparison of the application sizes and the involved personnel are informative and provide a measure of productivity and efficiency.

As pointed out in Section ‘KBL - A Knowledge-Based Electronic Planning Board System’ the KBL application consists of three main modules: the Scheduling Toolkit Module, the Simulation Module, and the Planning Board Module. The source application comprises 61 classes (see Figure 4) and about 46000 lines of code (LoC, without comments). The total number can be divided into 27400 LoC for the Scheduling Toolkit Module, 10400 LoC for the Simulation Module, 7300 LoC for the Planning Board Module, and 830 LoC for development utilities such as Makefiles and database loading tools.

The target application comprises the same number of classes as the source application. This is due to the fact that a shallow migration has been carried out rather than a complete redesign of the application. No additional classes had to be implemented. The total LoC for the migrated application was reduced to about 40000. This yields a reduction of 14% compared to the original size of the application. In the target application, the Scheduling Toolkit Module comprises about 21000 LoC (76%), the Simulation Module 10500 LoC (101%), and the Planning Board Module 7200 (99%). The development utilities also decreased in the amount of code (680 LoC) (82%). Table 10 gives a detailed overview of the application’s size and code modifications based on ONTOS and ObjectStore, respectively.

Our analysis of the application sizes shows a considerable reduction for the Scheduling Toolkit Module while the other modules, namely Simulation Module, Planning Board Module, and utilities, show only minor differences in size. This analysis is not surprising. It mainly stems from a simpler class definition in ObjectStore. The additional methods for the manipulation of persistent objects that must be implemented in ONTOS can be omitted in ObjectStore (-29%). A code inspection showed that the Scheduling Toolkit Module contains most of the class definitions and thus benefits from the simpler class declarations in

ObjectStore. The remaining modules contain only few class definitions. In contrast, we had to implement extension management for persistent classes in ObjectStore (+5%). ONTOS provides an automatic management of class extensions. The instances of a class are collected in a container with the same name as the class. Since ObjectStore does not support this feature it has to be provided by the application programmer. Yet the reduction due to simpler class definitions exceeded the effort for implementing class extensions. The code changes in the Simulation Module and the Planning Board Module are due to differences in the implementation of query management (see Table 7 and 8) and in the management of inverse relationships (see Table 9). The size of the Utilities Module changed considerably due to the different makefiles of the two database systems (see Table 5 and 6).

Considering the personnel involved in the migration process we distinguish three implementation phases: analysis of the application and the involved OODBS, migration analysis, and implementation of the migration. The whole migration process could be carried out in 14 weeks with 40 hours per week. The analysis of the application and of the OODBS, and the migration analysis took about 8 weeks. Note, that there was hands-on experience with ObjectStore beforehand, which helped to reduce the time to evaluate ObjectStore considerably. The implementation of the migration was carried out in 6 weeks. Table 11 gives a detailed overview of the human resources involved. In the initial planning of the migration we scheduled a bigger effort for the actual implementation process. Evaluating the whole migration process we have identified two reasons for the comparably low total migration effort of 14 weeks. On the one hand, knowledge of the application and/or of the involved database systems saves time and thus increases migration productivity locally. On the other hand, adherence to the migration framework supports a structured thus observable migration process, which saves time throughout all phases and increases migration productivity globally.

CONCLUSION & FURTHER WORK

In this paper we have presented a framework for the deep and shallow migration of applications between different OODBS. Our framework is based on several analysis steps to

perform a controlled migration process, namely the analysis and evaluation phase, the migration analysis, the development of a mapping strategy, and the implementation process. In the *analysis and evaluation phase* the application's database requirements, the hardware and software environment, and the involved database systems are considered. During *migration analysis* the information gathered during the previous phase, namely overlapping features, candidate shallow migration features, candidate deep migration features and database requirements, is used to figure out shallow migration features and deep migration features. Based on the financial and time restrictions imposed by the organization and the requirements with respect to functionality, an *implementation strategy* for shallow and deep migration features is developed and carried out in the *implementation process*. The migration framework has been successfully tested by migrating the KBL application from the OODBS ONTOS to ObjectStore.

The driving force behind the KBL migration has been to investigate the flexibility to exchange the underlying object store of the KBL application. It has been intended to sell KBL to different customers, which presumably would possess different OODBS. With the development of the migration framework and its test within a first case study we have hoped to gain insight into both a structured migration process and the intrinsics of different OODBS. Both expectations have been fulfilled.

Further investigations of this topic should include approaches for the (semi-)automatic migration of OODBS applications and the development of strategies for deep migration. Several tasks such as the re-implementation of the class definitions can be easily supported by the use of application conversion programs while others such as the selection of a mapping strategy require more effort. The latter case requires knowledge based support for the migration process. The resulting expert system based migration advisor could increase both the productivity and the quality of the migration process.

ACKNOWLEDGMENT

J. Thaler and A. Berger migrated the KBL prototype to ObjectStore.

REFERENCES

1. K. Dittrich, 'OODBS: The Next Miles in the Marathon', *Information Systems*, **15**, (1), (1990)
2. M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik, 'The object-oriented database manifesto' in: W. Kim, J.-M. Nicolas, S. Nishio (eds.), *Proc. of the 1st Conf. on Deductive and Object-Oriented Databases*, Kyoto, North-Holland, December 1989
3. J. Netze, H. Seelos, 'Scenes and strategies of data migration' (in german), *Wirtschaftsinformatik*, **35**, (4), (1993)
4. B. Shneiderman, G. Thomas, 'An Architecture for Automatic Relational Database System Conversion', *ACM Transactions on Database Systems*, **7**, (2), (1982)
5. S. Su, H. Lam, D. Lo, 'Transformation of Data Traversals and Operations in Application Programs to Account for Semantic Changes of Databases', *ACM Transactions on Database Systems*, **6**, (2), (1981)
6. R. Cattell (ed.), *The Object Database Standard ODMG-93*, Morgan Kaufmann Publishers, 1993
7. K. Dittrich, 'Migrating from conventional to object-oriented databases: a "can", a "must" - or none of both?' (in german), *Wirtschaftsinformatik*, **35**, (4), (1993)
8. D. Maier, 'Why isn't there an object-oriented data model?' in: G. X. Ritter (ed.), *Information Processing 89 - IFIP World Computer Congress*, North-Holland, 1989
9. V. Soloviev, 'An Overview of Three Commercial Object-Oriented DBMSs ONTOS, ObjectStore, and O_2', ACM *SIGMOD Record*, **21**, (1), (1992)
10. C. Lamb, G. Landis, J. Orenstein, D. Weinreb, 'The ObjectStore Database System', *Communications of the ACM*, **34**, (10), (1991)
11. P. Butterworth, A. Otis, J. Stein, 'The GemStone Object Database Management System', *Communications of the ACM*, 34, (10), (1991)
12. P. Kanellakis, C. Lecluse, P. Richard, 'Introduction to the Data Model' in: F. Bancilhon, C. Delobel, P. Kanellakis (eds.), *Building an Object-Oriented Database System: The Story of O_2*, Morgan Kaufmann, 1992
13. G. Kappel, S. Rausch-Schott, W. Retschitzegger, M. Schrefl, U. Schreier, M. Stumptner, S. Vieweg, 'Object-Oriented Database Management Systems - An Evaluation' ODB/TR 92-21, Institute of Applied Computer Science and Information Systems, Univ. of Vienna, 1993

14. S. Ahmed, A. Wong, D. Sriram, R. Logcher, ‘Object-oriented database management systems for engineering: A Comparison’, *Journal of Object-Oriented Programming (JOOP)*, **5**, (1992)
15. W. Kim, F. Lochovsky (eds.), ‘Object-Oriented Concepts, Databases, and Applications’, *ACM Press*, Reading MA, Addison-Wesley, 1989
16. M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, D. Beech, ‘Third-Generation Database System Manifesto’, ACM *SIGMOD Record*, 19, (3), (1990)
17. S. Zdonik, D. Maier (eds.), *Readings in Object-Oriented Database Systems*, San Mateo, CA. Morgan Kaufmann, 1989
18. KBL Esprit 5161, *Design, Development and Implementation of a Knowledge-based Leitstand (KBL)*, Deliverable Milestone 3, Commission of the European Community (CEC), 1992
19. KBL Esprit 5161, *Design, Development and Implementation of a Knowledge-based Leitstand (KBL)*, Final Deliverable, Commission of the European Community (CEC), 1993
20. A. Scheer, A. Hars, ‘The Leitstand - A new tool for decentral production control’ in: G. Fandel, G. Zäpfel (eds.), *Modern Production Concepts*, Springer Berlin, 1991
21. G. Kappel, S. Vieweg, ‘Database Requirements for CIM Applications’, *Journal of Integrated Manufacturing Systems*, **5**, *(4/5)*, (1994)
22. U. Schreier, ‘Database Requirements of Knowledge-based Production Scheduling and Control: A CIM Perspective’ in: R. Agrawal (ed.), *Proc. of the 19th International Conference on Very Large Data Bases*, Dublin, August 1993
23. R. Cattell and J. Skeen, ‘Object Operations Benchmark’, *ACM Transactions on Database Systems*; **17**, (1), (1992)
24. M. Carey, D. DeWitt, J. Naughton, ‘The OO7 Benchmark’, *Proc of the ACM SIGMOD Conf. ACM SIGMOD Record*, **22**, (2), (1993)

Figures:

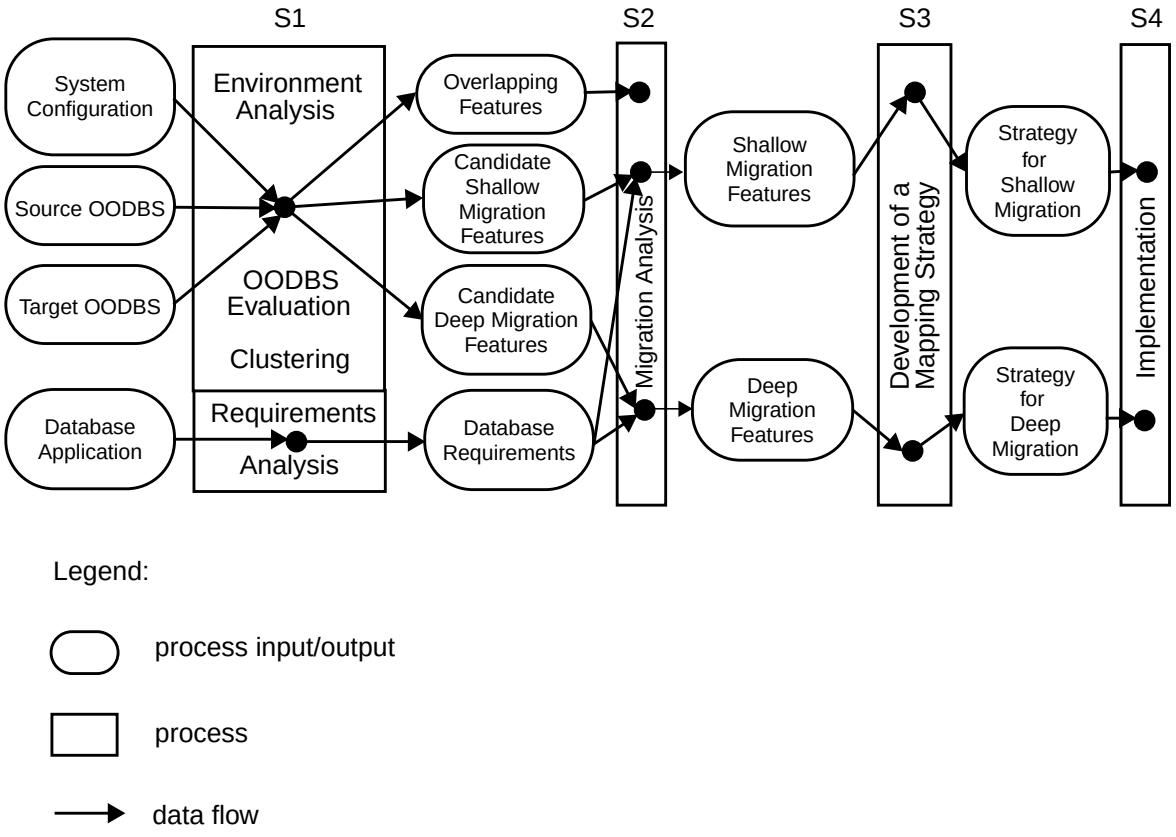


Figure 1: Model of the Migration Process

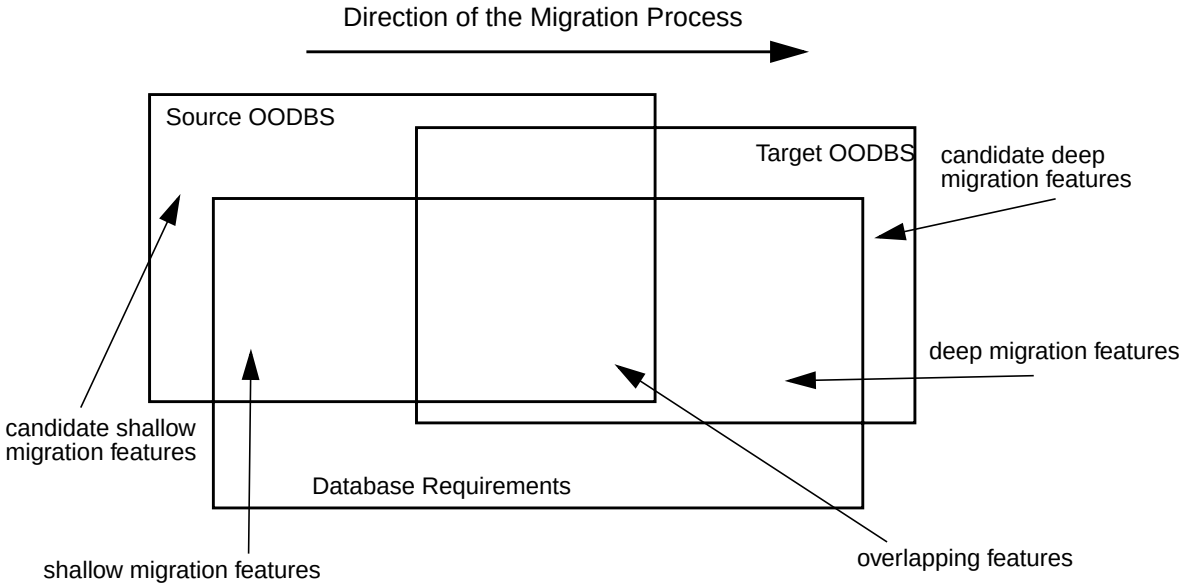


Figure 2: OODBS Features and Database Requirements

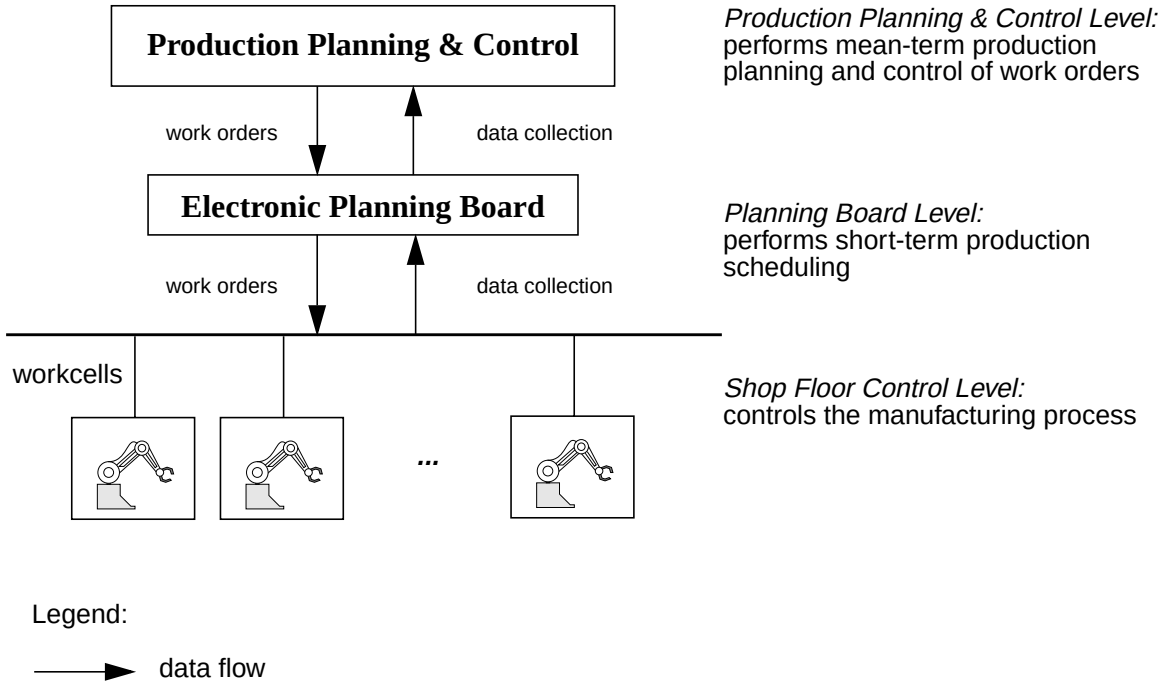
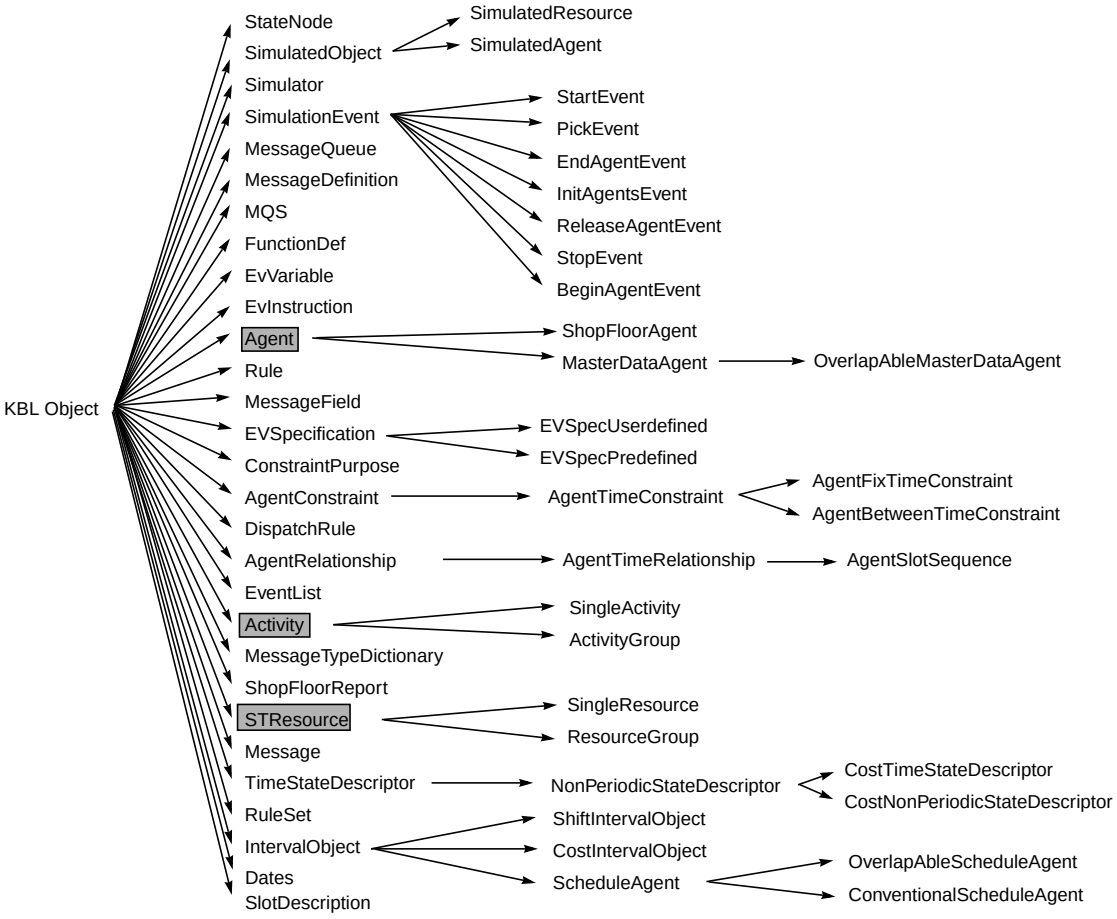


Figure 3: Integration of an Electronic Planning Board System

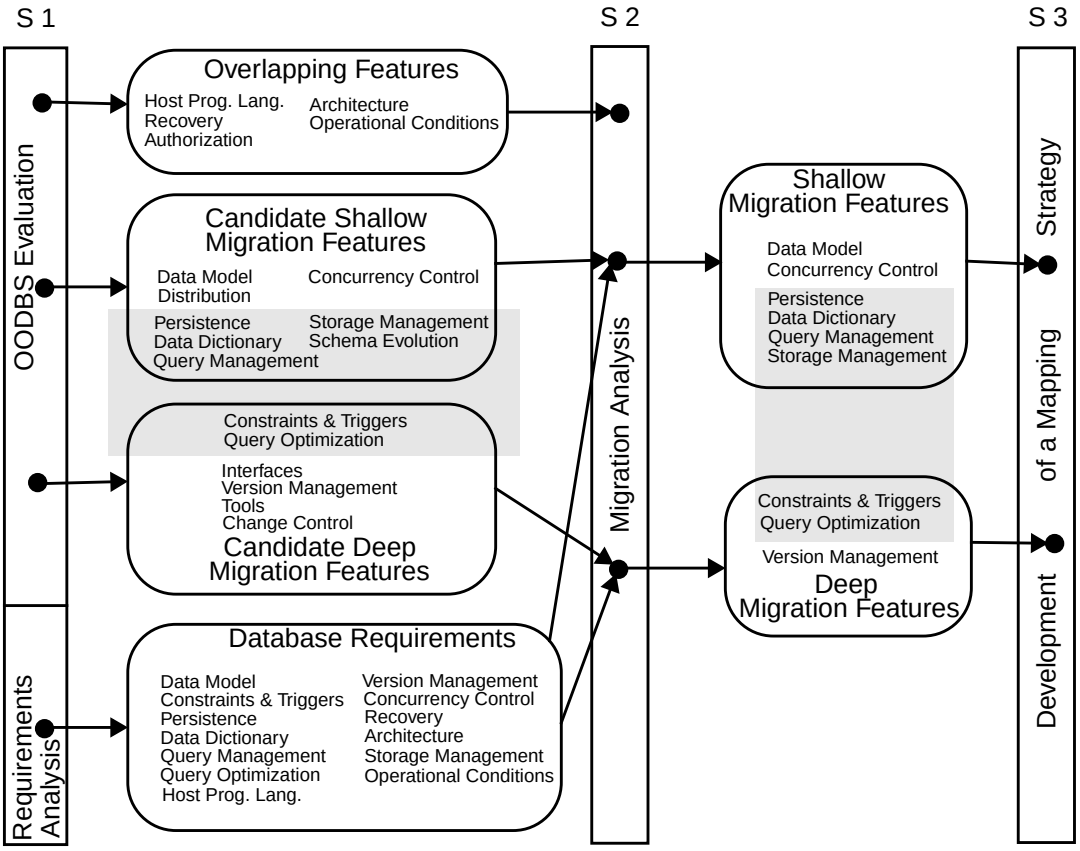


Legend:

subtype_of relationship

implements basic functionality of the scheduling toolkit module

Figure 4: KBL Class Hierarchy



Legend:

Clusters of Interdependent Functionalities

process input/output

process

data flow

Figure 5: Migration Process of KBL from ONTOS to ObjectStore

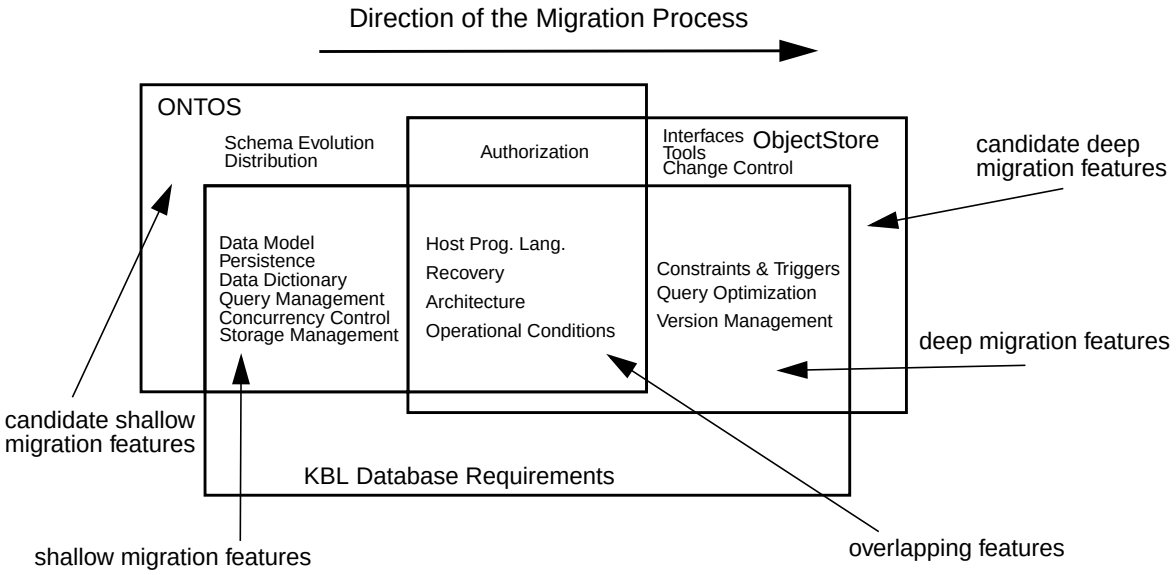


Figure 6: KBL Database Requirements and OODBS Features

Tables:

DBS Feature	*Description*	*OODBS Consideration*
Data Model	theoretical foundation for the specification of database schemes	complex object modelling, object identity, encapsulation, types, inheritance, overriding, late binding
Constraints & Triggers	specification and enforcement of integrity constraints	inverse relationships, object-oriented event/condition/action models
Persistence	data survives process boundaries	can be provided by inheritance, by declaration, by reachability from other persistent objects, and by collection membership
Data Dictionary	access to database schema information	access to database schema information using a meta object protocol
Tools	tools for application development (CASE), database inspection, user management, database archiving, and data dictionary management	(no specific OODBS consideration necessary)
Query Management	data manipulation language for insertion, deletion, update, and retrieval of database objects	navigational access via inter-object references, associative access via the specification of predicates ranging over collections
Query Optimization	efficient execution of database queries	object related indices such as class hierarchy index, nested index, path index, and multi-index
Host Programming Language	programming language for writing programs to manipulate the database	interfaces to object-oriented and non-object-oriented programming languages
Schema Evolution	management of database schema changes	global changes to the class hierarchy and local changes to single class descriptions
Change Control	techniques that are used to convert the database contents in order to conform to the evolved database schema; logical and physical data independence	object migration, schema versioning
Version Management	management of semantically meaningful snapshots in the evolution of the database contents	linear or branching version trees, static/dynamic references to versioned objects
Concurrency Control	transactions, management to reliably handle concurrent access to restricted resources, consistent management of system/media failures, user access control	advanced transaction management (e.g. nested transactions, long transactions, cooperative transactions)
Recovery		
Authorization		
Architecture	host-based/client-server/distributed architecture, disk placement and clustering techniques, disk to in-memory mapping	
Storage Management		
Distribution		
Interfaces	interfaces to other database systems and description standards	
Operational Conditions	hardware and software requirements	

Table 1: Features of OODBS

Steps	***Substeps***
(1) analysis and evaluation	• analysis of the system configuration • analysis of the OODBS • development of clusters of interdependent functionalities • analysis of the application's database requirements
(2) migration analysis	• shallow migration analysis • deep migration analysis
(3) development of a mapping strategy	• shallow migration strategy • deep migration strategy
(4) implementation	• coding • evaluation

Table 2: Steps and Substeps of the Migration Process

OODBS Features		
relevant for KBL		***not relevant***
Data Model Constraints & Triggers Persistence Data Dictionary Query Management Query Optimization Host Programming Languages	Version Management Concurrency Control Recovery Architecture Storage Management Operational Conditions	Tools Schema Evolution Change Control Authorization Distribution Interfaces

Table 3: KBL Database Requirements

DBS Feature	***ONTOS Rel. 2.2***	***ObjectStore Rel. 2.0***
Data Model	C++ data model meta-data access	C++ data model meta-data access
Constraints & Triggers	inverse members for dynamically created types only; no triggers	inverse members; no triggers
Persistence	persistence by inheritance	persistence by declaration
Data Dictionary	synonyms for objects	-
Tools	schema designer and browser	schema designer and browser
Query Management	navigational/associative access via C++; Object SQL	navigational/associative access via C++ and ObjectStore DML
Query Optimization	-	indexing of collections, clustering
Host Prog. Language	C++	C++
Schema Evolution	(partially) dynamic schema evolution	static schema evolution
Change Control	not supported	object migration supported but violating logical and physical data independence
Version Management	not supported	linear and branching versions of configurations of objects
Concurrency Control	transactions with checkpoints; object level locking	closed nested transactions; page level locking
Recovery	from volatile storage only	from volatile storage only
Authorization	UNIX-like database protection	UNIX-like database protection
Architecture	client/server environment	client/server environment
Storage Management	direct and indirect references	direct and indirect references
Distribution	transaction may access a single database server	transaction may access multiple database servers
Interfaces	-	STEP-Express
Operational Conditions	SUN workstations, TCP/IP network	SUN workstations, TCP/IP network

Table 4: Evaluation of ONTOS and ObjectStore

```
# Fragment of a Makefile for ONTOS
# Linker flags and required libraries
CFLAGS =                -Div2_6_compatible
LFLAGS =                -g
ONTOSLIBRARY =          -Bstatic -L$(LIB_DIR) -lONTOS
IV30LIBRARY = -         Bdynamic -L/interviews/lib/SUN4 -lUnidraw -lIV
X11R5LIBRARY =          -L/home/X11R5/lib -lXext -lX11 -lm
LIBRARY =               $(ONTOSLIBRARY) $(IV30LIBRARY)  $(X11R5LIBRARY)
# Source files of the application
SOURCES =       Activity.cc \
                Agent.cc \
                STResource.cc \
                ...
# Object files
OBJECTS =       ${SOURCES:.cc=.o}
main: $(OBJECTS) cplus $(LFLAGS) -o main -QUIET $(OBJECTS) $(LIBRARY)
...
# Creation of a database with the DBATool and registration of the database schema NAME in
# the directory DB_DIR managed by the database server SERVER. Every database consists of
# a kernel area and several data areas.
$(REGISTER_FLAG):
    cp $(KERNEL_DB) $(DB_DIR)/$(NAME)_Kernel
    chmod a+w $(THE_DB_DIR)/$(NAME)_Kernel
    DBATool -e register kernel $(NAME)_kern on $(SERVER) at $(DB_DIR)/$(NAME)_Kernel
    DBATool -e register database $(NAME) with kernel $(NAME)_kern
    DBATool -e create area $(NAME)_A1 at $(DB_DIR)/$(NAME)_area1 on $(SERVER)
    DBATool -e create area $(NAME)_A2 at $(DB_DIR)/$(NAME)_area2 on $(SERVER)
    DBATool -e add area $(NAME)_A1 to $(NAME)
    DBATool -e add area $(NAME)_A2 to $(NAME)
    DBATool -e set db $(NAME) primary $(NAME)_kern
...
```

Table 5: Fragment of the Makefile for ONTOS

```
# Fragment of a Makefile for ObjectStore
# including ObjectStore specific compiler directives
include $(OS_ROOTDIR)/etc/ostore.mk
# Registration of the database schema in the data dictionary
OS_COMPILATION_SCHEMA_DB_PATH= /home/KBL/compilation_schema_db
OS_APPLICATION_SCHEMA_DB_PATH= /home/KBL/application_schema_db
# Compiler and linker flags and required libraries
CCFLAGS =     -gx
LDFLAGS =     -g
LDLIBS =      -losmop -loscol -los
# Source files of the application
SOURCES =     Activity.cc \
              Agent.cc \
              STResource.cc \
              ...
# Object files
OBJECTS =     ${SOURCES:.cc=.o}
# Executables
EXENAME = main
...
```

Table 6: Fragment of the Makefile for ObjectStore

ONTOS	***ObjectStore***

// KBLObject directly inherits from Object
class KBLObject : **public** Object
{
  ...
};
 	
// IntervalObject indirectly inherits from Object
class IntervalObject : **public** KBLObject
{
  ...
};
 	
// ScheduleAgent indirectly inherits from Object
class ScheduleAgent : **public** IntervalObject
{
private:
  // indirect Reference to a MasterDataAgent object
  Reference ivMasterDataAgent;
  // indirect reference to a Resource object
  Reference ivResource;
  // Constructor which is called by the constructor
  // of the related MasterDataAgent
  ScheduleAgent (MasterDataAgent *);
  // ONTOS required function for deleting the
  // object from the database
  virtual void deleteObject (Boolean deallocate = FALSE);
public:
  // returns a pointer to the related MasterDataAgent object
  virtual MasterDataAgent *getMasterDataAgent ();
  // schedu*les the* Agent *object on the* Resource *object*
  virtual void putSingleResource (SingleResource *);
  // returns a pointer to the Resource object on which
  // the Agent object is scheduled
  virtual SingleResource *getSingleResource ();
  // returns a list including the ScheduleAgent
  // objects already scheduled
  List* ScheduleAgent::getScheduledSAList();
  // constructor
  ScheduleAgent ();
  ...
  // ONTOS required functions
  // activation constructor to activate the object from disk
  ScheduleAgent (APL *);
  // destructor
  ~ScheduleAgent ();
  // returns a pointer to the object representing the
  // class information
  virtual Type *getDirectType ();
  // deactivate the object from main memory
  virtual void Destroy (Boolean aborted = FALSE);
  // write the object to the database
  virtual void putObject(Boolean deallocate = FALSE);
};
	extern os_database *KBLdb;
	
	class KBLObject
	{
	public:
	  // static persistent set which includes all instances
	  persistent<KBLdb> os_Set<KBLObject*> * extent;
	  ...
	  // implementation of the object naming
	  char *ivName;
	  ...
	};
	
	class IntervalObject : **public** KBLObject
	{
	public:
	  // static persistent set which includes all instances
	  // of IntervalObject
	  persistent<KBLdb> os_Set<IntervalObject*>* extent;
	  ...
	};
	
	// neither IntervalObject nor ScheduleAgent inherit from
	// any predefined class
	class ScheduleAgent : **public** IntervalObject
	{
	private:
	  // direct reference to a MasterDataAgent object
	  MasterDataAgent *ivMasterDataAgent;
	  // direct reference to a Resource object
	  STResource *ivResource;
	  // constructor like in ONTOS but implemented differently
	  ScheduleAgent (MasterDataAgent *);
	public:
	  // static persistent set which includes all instances of
	  // ScheduleAgent
	  persistent<KBLdb> os_Set<ScheduleAgent*>* extent;
	  // like in ONTOS
	  virtual MasterDataAgent *getMasterDataAgent ();
	  virtual void putSingleResource (SingleResource *);
	  virtual SingleResource *getSingleResource ();
	  os_List<ScheduleAgent*>
	  *ScheduleAgent::getScheduledSAList();
	  ...
	  ScheduleAgent (); // constructor
	  ~ScheduleAgent (); // destructor
	  // ObjectStore does not require any system specific methods
	};
	// ObjectStore implementation of the constructor
	ScheduleAgent::
	  ScheduleAgent (MasterDataAgent * theMasterDataAgent)
	{
	  ...
	  // Insertion of the created ScheduleAgent object into the
	  // class extent
	  extent->insert(this);
	};

Table 7: ONTOS and ObjectStore Class Definitions

ONTOS	***ObjectStore***

List* ScheduleAgent::getScheduledSAList()
{
List *scheduledSAList;
 	
// Creation of the instance iterator for the
// class ScheduleAgent
InstanceIterator scheduleAgentIterator ((Type*)
OC_lookup ("ScheduleAgent"));
// The iterator function moreData returns the next value
// If there is no further value the iteration will terminate.
while (scheduleAgentIterator.moreData())
{
// The function getSingleResource returns a pointer to
// the Resource object on which the Agent object is
// scheduled. If a Resource object exists the object
// is inserted into the appropriate list.
if (scheduleAgentIterator->getSingleResource) != 0
scheduledSAList->Insert ((Argument) (Entity*)
scheduleAgentIterator);
}
// returns the resulting list
return scheduledSAList;
}
	os_List<ScheduleAgent*> * ScheduleAgent::
	getScheduledSAList()
	{
	os_List<ScheduleAgent*> * scheduledSAList;
	ScheduleAgent* currentScheduleAgent;
	// The foreach-statement allows to iterate over the
	// elements of the set specified as second argument.
	// The element of each iteration will be referenced by
	// the first specified argument
	foreach(currentScheduleAgent, ScheduleAgent::
	extent)
	{
	// selection criteria like in ONTOS
	if (currentScheduleAgent->getSingleResource) != 0
	scheduledSAList->insert(currentScheduleAgent);
	}
	// returns the resulting list
	return scheduledSAList;
	}

Table 8: ONTOS and ObjectStore Extension Management

ONTOS	***ObjectStore***

ONTOS class definition:
class Activity: **public** KBLObject
{
private:
  // reference to the list of superactivities
  List *ivSuperActivityList;
  // reference to the list of subactivities
  List *ivSubActivityList;
  ...
protected:
  Activity (char *name = (char *)0, ActivityGroup
  *superActivity = (ActivityGroup *) 0);
  ...
public:
  Boolean addSubActivity (Activity &activity);
  ...
};
ONTOS implementation of the constructor:
Activity::Activity (char *name, ActivityGroup *superActivity):
  KBLDirectParentClass (name)
{
  ...
  ivSuperActivityList->Insert ((Argument)(Entity*) superActivity);
  // call to maintain referential integrity
  superActivity->addSubActivity(*this);
  ...
}
	ObjectStore class definition:
	class Activity: **public** KBLObject
	{
	private:
	  // related data members are marked ‘inverse_member’
	  // followed by the corresponding data member
	  os_List<ActivityGroup*> *ivSuperActivityList
	  inverse_member ivSubActivityList;
	  os_List<Activity*> *ivSubActivityList
	  inverse_member ivSuperActivityList;
	  ...
	protected:
	  Activity (char *name = (char *)0, ActivityGroup
	  *superActivity = (ActivityGroup *) 0);
	  ...
	public:
	  Boolean addSubActivity (Activity &activity);
	  ...
	}
	ObjectStore implementation of the constructor:
	Activity::Activity (char *name, ActivityGroup *superActivity):
	  KBLDirectParentClass (name)
	{
	  ...
	  // no need for further code to ensure referential integrity
	  ivSuperActivityList->insert (superActivity);
	  ...
	}

Table 9: Inverse relationships in ONTOS and ObjectStore

Modules	ONTOS Implementation	ObjectStore Implementation							Total %
		added		deleted		modified		Total	
Scheduling Toolkit	27400	1500	5%	8000	29 %	200	1%	20900	~ 76%
Simulation	10400	300	3%	200	2%	370	4%	10500	~ 101%
Planning Board	7300	0	0%	100	1%	250	3%	7200	~ 99%
Utilities	830	400	48 %	550	66 %	0	0%	680	~ 82%
Total	45930	2200	5%	8850	19 %	820	2%	39280	~ 86%

Table 10: Migration Statistics (Lines of Code)

Task	Manpower (in weeks)
Application Analysis	3
ONTOS Analysis	2
ObjectStore Analysis	1
Migration Analysis	2
Implementation	6
Total Migration Effort	14

Table 11: Migration Effort (Manpower)