Migrating a Leitstand System between Object-Oriented Database Systems - An Experience Report

C. Huemer, G. Kappel, S. Vieweg
Institute for Applied Computer Science and Information Systems
Department for Information Engineering; University of Vienna, Austria
{ch, gerti, sv}@ifs.univie.ac.at

Abstract

Advances in technology and organizational structures effect changes in database requirements. Frameworks for the migration of database applications between different database systems accomplish the task of mapping evolving requirements to existing applications. In this paper, we describe a framework for migrating database applications between different object-oriented database systems. We illustrate our framework by means of migrating KBL† from ONTOS™ to ObjectStore™.

1. Introduction

The requirements for database applications evolve over time. The changes of the requirements may result from the demand for additional system functionality, changes in the business organization, a shift in technology or improved insight in the task to be performed. These changes must then be mapped to the existing applications. In general, with *migration* we describe the controlled mapping of evolving requirements to existing applications. Database migration results from changes in the application's database requirements. With *database migration* we denote the migration of a given database application from a source database system (DBS) to a target DBS. The application migrated to the target DBS is constrained to follow the changed requirements and to deliver 'equivalent' results. Equivalence is defined by the application's semantics. The need for database migration may have several reasons: system downsizing (e.g. from host-based computing to client/server computing), substitution and integration of database systems (e.g. migration from hierarchical to relational database system), change of the programming languages in use (e.g. from COBOL to C++), or outsourcing of database services. System downsizing and outsourcing of database services result from changes in the organization of information system services while the substitution and integration of DBSs and the change of programming languages are mainly caused by technological change.

In this paper we focus on the substitution of database systems. In particular, we emphasize on the migration of database applications between different object-oriented DBSs (ooDBS). The need for work in this area is obvious. There does not yet exist a unique data model for ooDBSs. Each ooDBS uses its own data model although there are similarities between them. Thus, the exchange of the underlying ooDBS of an application is not necessarily straightforward but may cause considerable changes in the application. Furthermore, 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

† The work reported in the paper is part of the ESPRIT project KBL (ESPRIT No. 5161), whose goal is the design and development of a Knowledge-Based Leitstand. The authors are responsible for incorporating object-oriented database technology as underlying information store, and as integrative component between the Leitstand and various other CIM components. The support by FFF (Austrian Foundation for Research Applied to Industry) under grant No. 2/279 is gratefully acknowledged.

ooDBS as well as with respect to extensibility, the feasibility of migrating applications between different ooDBSs must be a prerequisite for deciding on one or the other of the existing ooDBSs.

Current approaches to the migration of database applications emphasize on the migration from hierarchical to relational DBS or from non-DBS applications to relational DBS applications [Brodie93, Meier93, Shneiderman82, Su81]. Approaches for the migration between ooDBSs are still missing in the literature. The purpose of this paper is to fill this gap. We thereby concentrate on the shallow migration of database applications. *Deep migration* focuses on a complete re-design of the application in order to exploit the whole functionality of the target DBS. *Shallow migration* requires only a re-design of those database components of the application which have a different semantics in the source DBS and the target DBS, or which are missing in the target DBS and, hence, have to be simulated [Dietrich93]. We opted for the latter approach for two reasons. Firstly, the main goal of our project has been to investigate the applicability of several ooDBSs for the application at hand rather than to totally re-design the application every time to take full advantage of all features of the target ooDBS. Secondly, shallow migration is the more feasible approach when performing a database migration under restricted resources such as time and personnel.

The rest of the paper is organized as follows. In Section 2 we shortly describe the application of our investigations, namely the KBL application. In Section 3 we present the general framework for migrating applications between different ooDBS. Section 4 describes the migration of the KBL application from ONTOS to ObjectStore. We conclude with a report of our experiences gained during the migration process.

2. The Knowledge-Based Leitstand (KBL)

This section contains an overview of the KBL system. KBL has been developed under the object-oriented paradigm and is implemented on top of the ooDBS ONTOS. We will now shortly present the functionality of the system, describe the basic software modules and give an overview about the main classes. The intention of this section is to give an overview of the system. We refer to [KBL92, KBL93] for a further description of the Knowledge-Based Leitstand.

A Leitstand is a distributed computer aided graphical decision support system for interactive production scheduling. It interacts with the production planning and control system (PPC) and other systems at the shop-floor level. KBL is a Leitstand system which emphasizes on short term production scheduling. The main components of KBL are

- Interactive Adviser: KBL is equipped with an Interactive Advisor which will constantly analyze the status of the whole system. It provides the production scheduler personnel with scheduling support and advice for alternate actions.
- Knowledge representation and acquisition: The information relevant to 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.
- Leitstand Logic: The Leitstand Logic or Scheduling and Control Subsystem represents the interface between the Leitstand application and the DBS. It provides DBS functionality, version handling, interfaces to other application processes, and basic scheduling routines.
- Simulation of schedules: Alternate schedules can be simulated in order to evaluate the performance of schedules under different constraints.
- Evaluation of schedules: The Evaluation component of the Leitstand allows the assessment of different scheduling strategies. A detailed analysis based on built-in evaluation functions or

user-defined evaluation criteria should help to improve the quality of the scheduling process.

- Communication interfaces to other systems: Leitstand systems operate in a distributed environment. Communication interfaces to the production planning and control system (PPC) as well as to the shop-floor control system (SFCS) are supported.

The current implementation of the Leitstand contains the Interactive Adviser, the Knowledge Representation component, the Leitstand Logic, and the Simulation component. KBL has been implemented on top of the object-oriented DBS ONTOS in the first place. The system is structured into the following modules: Scheduling Toolkit Module, Planning Board Module, and Simulation Module.

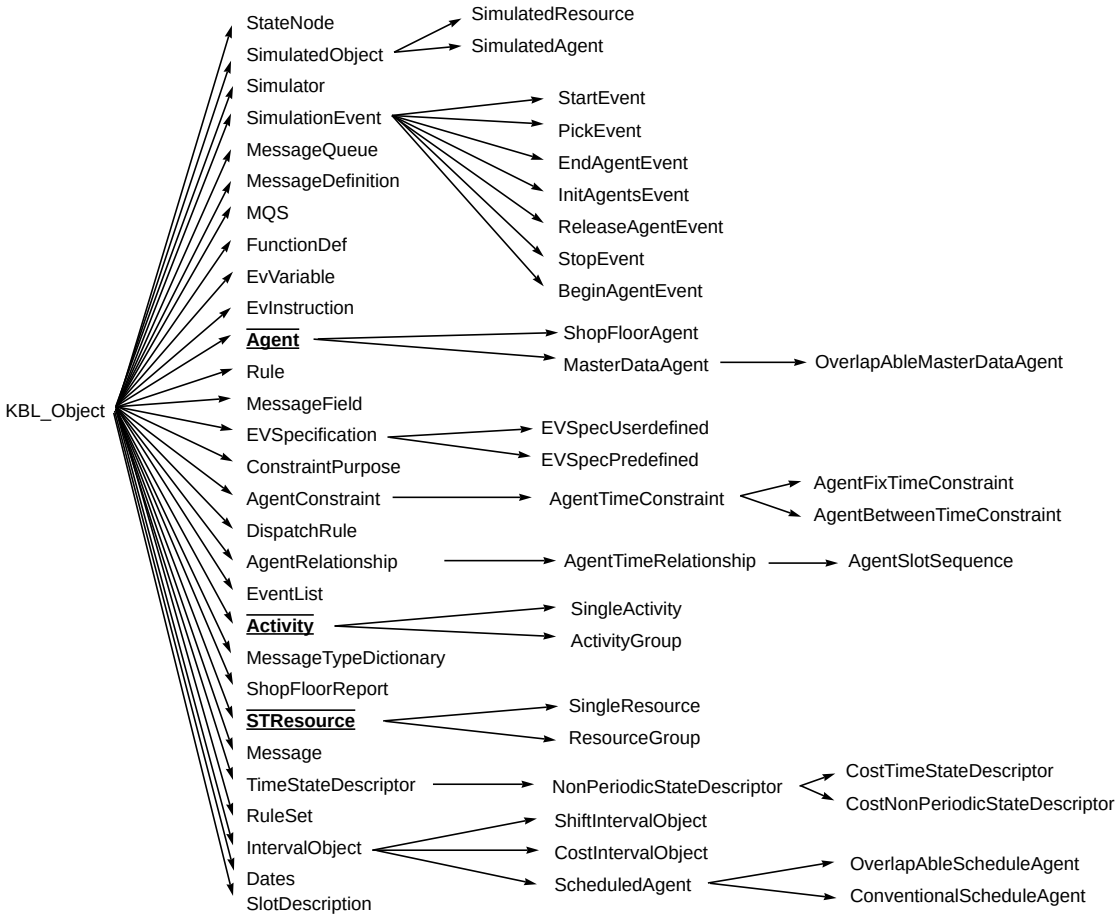

Fig.1: The KBL Class Hierarchy

The Scheduling Toolkit 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 Planning Board Module contains the routines for the management of the graphical user interface. The Simulation Module allows to automatically schedule orders. 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 Fig. 1. The basic functionality of the Scheduling Toolkit is implemented in the classes `Agent`, `Activity`, and `STResource` (highlighted in Fig. 1). The class `Activity` is used to describe operations in the Leitstand environment. These include the factory specific operations such as ‘drilling’, ‘milling’ etc. The class `STResource` describes any kind of resources (such as material, machines, workers etc.). 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 are scheduled or unscheduled, whether and to which rate the activities consume or produce other resources etc.

In conjunction with other classes such as `TimeStateDescriptor`, `IntervalObject`, and `AgentConstraint` the above described classes represent a flexible environment for the modeling of manufacturing processes. A detailed description of the KBL class hierarchy can be found in [KBL93].

3. Framework for Migrating Applications between ooDBSs

In this section we develop a general outline for migrating applications (such as KBL) between ooDBSs independent of the ooDBSs involved. Therefore, we have to look first 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 database requirements of the application and to explore how these requirements are met by each of the involved ooDBSs. The features required by the application and the features supported by the ooDBSs should be defined within the same set of criteria to be comparable. These criteria could be taken, for example, from the ooDBS Manifesto [Atkinson89] or from the 3rd Generation Database Manifesto [Stonebraker90]. We have decided that all further investigations will be based on the main topics of the evaluation catalogue of [Kappel93] (see list below), which has been developed to evaluate commercially available ooDBSs. The topics of the evaluation catalogue comprise a superset of all features discussed in [Atkinson89, Stonebraker90]:

- Data Model
- Constraints and Triggers
- Persistence
- Data Dictionary
- Tools
- Query Management
- Host Programming Languages
- Schema Evolution
- Change Control
- Versioning
- Concurrency Control
- Recovery
- Authorization
- Architecture
- Storage Management
- Query Optimization
- Operational Conditions
- Distribution
- Interfaces

The subtasks towards a successful migration and their input and output are presented in Fig. 2. These subtasks can be grouped into four stages:

S1: analysis of the involved ooDBSs and of the application's DBS requirements

S2: migration analysis

S3: development of a strategy for the implementation of the migration

S4: implementation

The *first stage* includes the DBS evaluation and the requirements analysis of the application. As the analysis of the system configuration is part of the evaluation topics (feature operational conditions) it will be included within the DBS evaluation. The *second stage* comprises the migration analysis to decide which of the features have to be considered during the migration of the specific application. The *third stage* is the development of a strategy for the migration of these features. In the *last stage* the implementation of the migration will be done according to this strategy. The activities within

the four stages are discussed in the following in detail.

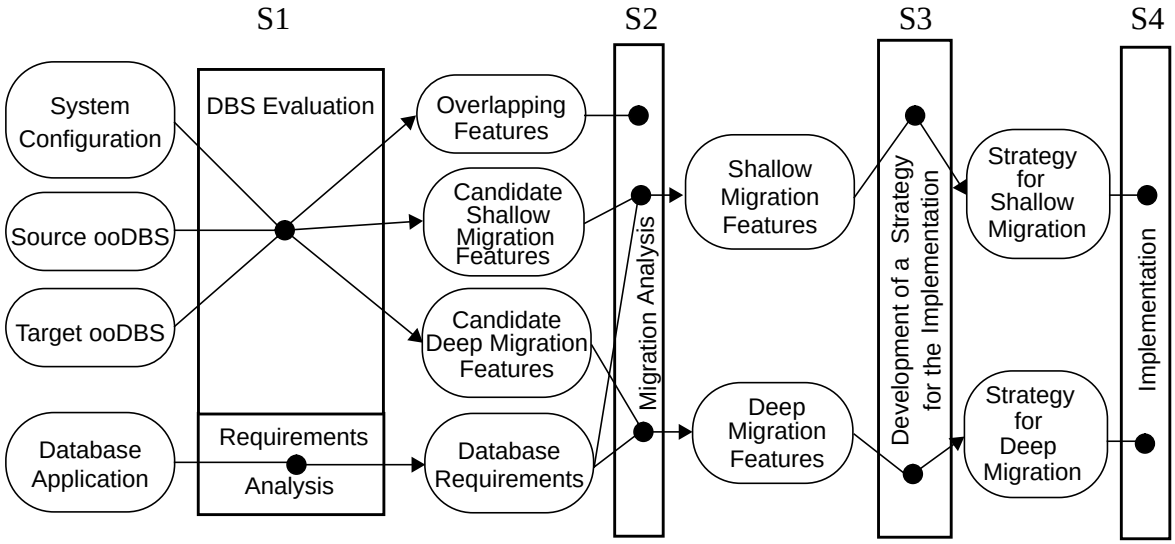

Fig.2: Model of the Migration Process

The evaluation of the ooDBSs leads to a set of features supported by each system. Furthermore, a comparison of these features will result in three subsets according to their effect on the migration:

- **Overlapping features** are supported by both ooDBSs in the same manner and have no effects on the migration.
- **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 and might be included in the migration if required by the application (deep migration).

The features required by the application are the result of the requirements analysis and are called database requirements. 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. We will remove those features from candidate migration features which are not required by the application. We also do not have to consider the overlapping features anymore, because they do not effect the migration. The migration analysis will result in two sets of features which have to be considered during the migration. The shallow migration features include those features which have to be migrated to obtain the same database functionality for the application as it was available from the source ooDBS. The deep migration features include those features which extend the database functionality, because they are not provided by the source ooDBS but are required by the application. Next, a strategy for the implementation of the migration features in the target ooDBS must be developed. The concluding task is the implementation of the migration which is done according to the developed strategy.

Splitting the first stage (S1) presented in Fig. 2 into four steps leads to the following seven steps:

1. Analysis of the hardware and software environment (= system configuration).
2. Analysis of the ooDBS
3. Development of clusters of interdependent functionality
4. Analysis of the application’s DBS requirements

5. Migration analysis
6. Development of a strategy for the implementation
7. Implementation

The **analysis of the hardware and software environment** is of general interest. But as each feature included in the evaluation catalogue will be analyzed for each ooDBS under consideration of the underlying system configuration it is implicitly included in step 2.

The task of the **analysis of the ooDBSs** is to evaluate, to classify and to compare the features of the two ooDBSs involved in the migration. It is essential to figure out the features which are supported by the source DBS and those which are supported by the target DBS. Next, we have to find the overlapping features of the source DBS and the target DBS. Note, two features might be syntactically similar or identical but semantically different. Only those features which are syntactically and semantically identical cause no problems for the migration and will be included in the set of overlapping features. The result of the analysis of the ooDBSs is on one hand the set of overlapping features and on the other hand the set of candidate shallow/deep migration features which will cause changes in the application. The set of candidate shallow migration features comprises those features which have to be changed/simulated to provide the database functionality of the source DBS when migrating to the target DBS. The set of candidate deep migration features consists of features which are provided by the target DBS but not by the source DBS, and, hence, would extend the database functionality if considered in the migration process.

Due to the fact that there exist some interdependencies between different features, changes of the application required by one feature might cause changes of another feature, too. Since related changes should be considered together we propose to cluster features which require associated changes. We call them **clusters of interdependent functionality**. Each of these resulting clusters forms a separate migration unit, which can be migrated independently from the features outside this specific cluster.

A careful **analysis of the application** 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 features which have to be considered within the migration. The necessary features of the application are collected in the set of database requirements.

The main task of the **migration analysis** is to find the intersection of the database requirements with the candidate shallow migration features and with the candidate deep migration features, respectively. Those features of the candidate shallow migration features and of the candidate deep migration features which are not required by the application do not have to be considered further. One might argue that removing those features before the analysis of the ooDBSs (by comparing the database requirements with the features of the evaluation catalogue) would reduce the effort in that part. But it is our point of view to remove those features not earlier than at this step for reasons of completeness and possible future changes in the application's requirements. The result of this step is the set of shallow migration features and the set of deep migration features. These two sets include those features which have to be considered when migrating one specific application from the source DBS to the target DBS.

For the features residing in the set of shallow migration features and in the set of deep migration features a **strategy for their implementation** in the target ooDBS must be developed. Note, that developing a strategy for the deep migration features can be omitted if only a shallow migration is required. By developing an implementation strategy one might realize that the effort to implement a specific feature exceeds the semantic gain provided by that feature. Thus, whether a certain feature

is migrated has to be decided on a case by case basis and heavily depends on the available resources. The result of this step is a mapping strategy for all those features which are included in the implementation of the migration.

The **implementation** of the migration is the programmer's task. He or she is responsible for changing the code according to the developed strategy leading to the final running application on top of the target DBS.

4. Migrating KBL from ONTOS to ObjectStore

4.1. A brief tour of ONTOS and ObjectStore

In the following we will shortly describe the main features of ONTOS and ObjectStore. For further information we refer to [Ahmed92, Kappel93, Soloview92] and to the product literature.

ONTOS (Release 2.2.) is based on C++ and operates in a client/server environment. The client/server architecture is based 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 way of meta-data management. Any of the system services (storage manager, transaction manager) can be modified in order to support user-defined extensions. Databases can be accessed either with C++ or with an interactive SQL-Interface. The objects are accessed and referenced through logical 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 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. Version management is not supported.

ObjectStore (Release 2.0.) is based on C++ and operates in a client/server environment. The ObjectStore server is a page server. It is available on the major workstation platforms. It provides access to the database either with C++ or with a C++ extension (ObjectStore DML); an SQL-like query language is not supported. The strength of ObjectStore is its memory architecture and the resulting performance benefits. Objects are accessed via their physical addresses. Although this implies some restrictions on the size of the database, the amount of data that is accessible within a single transaction, and database reorganization, the advantages of this approach dominate its weaknesses for a certain domain of applications. Persistence is orthogonal to the type system and thus provides advantages in case of migrating applications to ObjectStore. ObjectStore provides navigational access via object references and associative access via queries over collections. The embedded query language is strongly connected to C++. In terms of extensibility and schema access, ObjectStore does not provide the flexibility of ONTOS. The meta-object protocol (MOP) provides only access to class descriptions; dynamic modifications of class descriptions are not supported. Static schema evolution is supported via an object migration tool to convert the instances of the old schema to conform to the new schema. ObjectStore provides a sophisticated versioning mechanism that supports the versioning of object configurations. The concurrent access to the database is controlled with implicit page level locking and closed nested transactions.

4.2. The Migration Process

In this section we discuss step 2, step 4, and step 5 of the migration process outlined in section 3 for migrating KBL from ONTOS to ObjectStore. Note, step 3 is discussed by example in the next

subsection. The results of step 2 to step 5 are presented in Fig. 3, followed by comments which explain the grouping of each feature in turn. Fig. 3 depicts the development of a mapping strategy only for those features residing in the set of shallow migration features. This is due to the fact that our goal included only a shallow migration and not a deep migration. Due to space limitations we do not discuss the mapping strategy for each of the shallow migration features but highlight the migration strategy of a single feature, namely persistence, in the subsequent subsection.

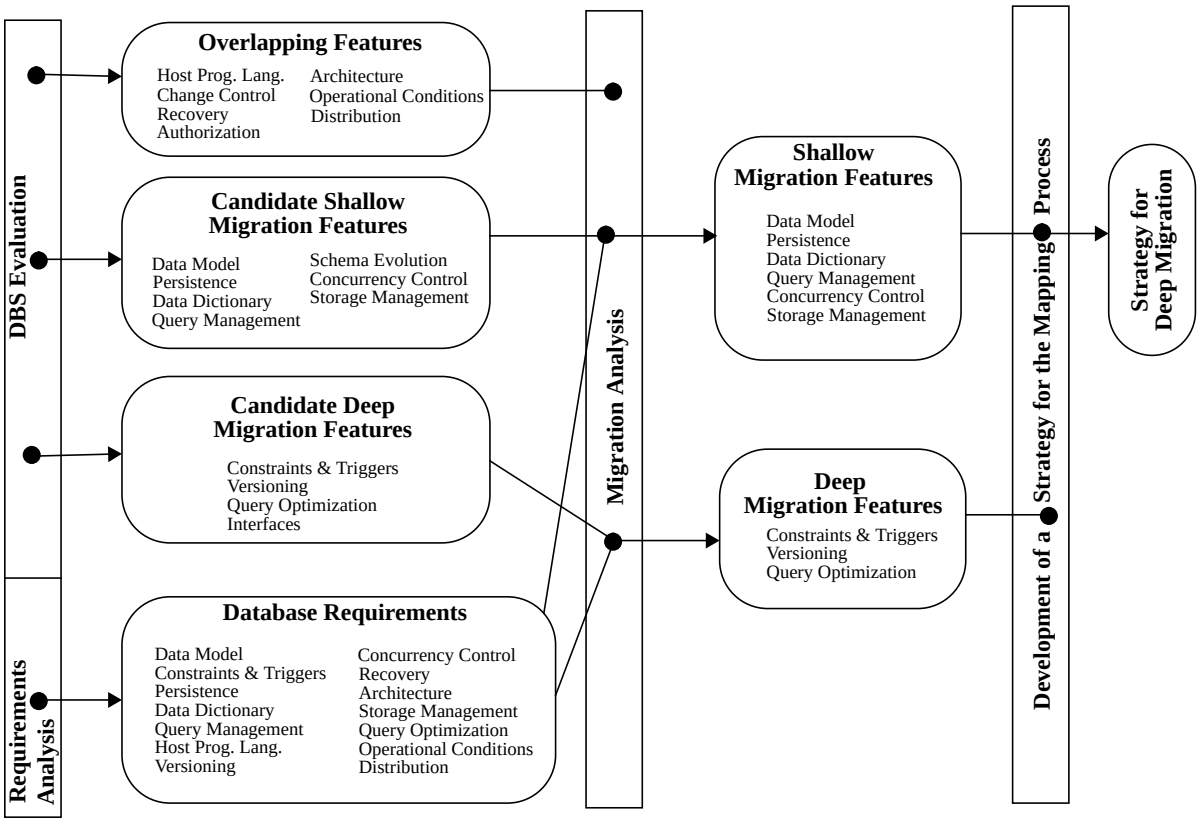

Fig.3: Migration Process from ONTOS to ObjectStore

Both ONTOS and ObjectStore are based on the C++ **data model**. They both use the C++ basic data types and some additional data types, e.g. to support collections. Furthermore, the C++ data model is extended to provide typical database features, such as transactions. For the additional data types and the database extensions each model uses its own syntax and semantics. Thus, the feature data model is included in the candidate shallow migration features. Since the functionality of the data model is crucial for the KBL application it is included in the database requirements, and, as a consequence 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 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 the database requirements because some data members are inverse to each other in KBL. As a consequence, we add constraints and triggers to the set of deep migration features.

Persistence is required by any database application. It is reached in ONTOS by inheritance and in ObjectStore by declaration. Thus we have to include persistence in the shallow migration features

and have to develop a strategy for its mapping (see 4.3. Mapping Strategy for the Feature ‘Persistence’).

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 (see query management). Thus, this feature becomes also a member of the shallow migration features.

The analysis of the **tools** is only interesting in terms of supporting the implementation process but it has no effects on the application itself. Thus, this feature 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 which belong to a specific class and its subclasses. In ObjectStore, the entry points of the database are persistent root objects. Due to this different access methods and since KBL, like any other database application, requires persistent object retrieval query management is included in the set of shallow migration features.

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

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

Both systems provide neither logical nor physical data independence. Thus, the feature **change control** resides in the overlapping features. For KBL data independence is not absolutely necessary and, thus, it is not included in the database requirements.

ONTOS does not support a **versioning** mechanism. As a consequence, the versioning of the schedules in KBL 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 lock strategy alternatively to a pessimistic one. The optimistic lock strategy is also part of the database requirements of KBL and will therefore be 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 KBL application to use 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 access control to data is supported only at the database level, both in ONTOS and in ObjectStore. Database access can be 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 especially the activation of referenced objects. Also, the facilities for the use of indexes and clustering are different in both systems. Thus, storage management is a candidate shallow migration feature. It is a shallow migration feature since the KBL prototype requires the activation of referenced objects and the use of indexes and clustering mechanisms.

Query optimization has not been considered in the KBL prototype but it will be included in the database requirements of the final product. ONTOS provides limited query optimization; ObjectStore supports index and clustering for query optimization. Thus, query optimization is a member of the deep migration features.

KBL is 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.

The feature **distribution** is also an overlapping feature since KBL does not require the distribution of the database over multiple servers.

ONTOS does not provide any import and export **interfaces**. ObjectStore offers a third party tool to import 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.

4.3. Mapping Strategy for the Feature ‘Persistence’

In this section we show by example the development of a mapping strategy for one of the most interesting features in the set of shallow migration features of Fig. 3, namely persistence.

Persistence is reached in ONTOS by inheritance from the ONTOS specific class `Object`; in ObjectStore by declaration. As this basic feature is implemented differently in the two ooDBSs, it is a member of the candidate shallow migration features. The clustering with other features according to additional affected changes will be presented below. As persistence is required by KBL it is also included in the set of the database requirements. During the migration analysis we compare the two sets mentioned above. Since persistence resides in both sets it will also become a member of the shallow migration features. Thus, we have to develop a strategy for the mapping of the ONTOS specific code to ObjectStore. For the development of such a strategy it is necessary to investigate in further detail how persistence is implemented in ONTOS and in ObjectStore, respectively. To reach persistence in ONTOS the following conditions must hold true:

- Classes must have a derivation 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 then this specific instance is made persistent. In ObjectStore the class definition for a persistent object does neither have to include a derivation path through a

specific predefined class nor does it have to include the ONTOS specific functions. An object is made persistent by declaration. There is no need to call an operation like `putObject` to write it to secondary storage.

ONTOS class definition	*ObjectStore class definition*

`class KBLObject : public Object`
`{`
`  ...`
`};`
 	
`class IntervalObject : public KBLObject`
`{`
`  ...`
`};`
 	
`class ScheduleAgent : public IntervalObject`
`{`
`private:`
`  Reference ivMasterDataAgent;`
`  Reference ivResource;`
`  // Constructor which is called by a call in the`
`  // constructor of the related MasterDataAgent.`
`  ScheduleAgent (MasterDataAgent *);`
`  // ONTOS required function`
`  virtual void deleteObject`
`  (Boolean deallocate = FALSE);`
 	
`public:`
`  virtual MasterDataAgent *getMasterDataAgent ();`
`  virtual void putSingleResource (SingleResource *);`
`  virtual SingleResource *getSingleResource ();`
`  ScheduleAgent ();`
`  ...`
`  // ONTOS required functions`
`  ScheduleAgent (APL *);`
`  ~ScheduleAgent ();`
`  virtual Type *getDirectType ();`
`  virtual void Destroy (Boolean aborted = FALSE);`
`  virtual void putObject(Boolean deallocate = FALSE);`
`};`
	`extern os_database *db;`
	
	`class ScheduleAgent : public IntervalObject`
	`{`
	`private:`
	`  persistent<db> os_Set<ScheduleAgent*>* extent;`
	`  MasterDataAgent *ivMasterDataAgent;`
	`  STResource *ivResource;`
	`  ScheduleAgent (MasterDataAgent *);`
	
	`public:`
	`  virtual MasterDataAgent *getMasterDataAgent ();`
	`  virtual void putSingleResource (SingleResource *);`
	`  virtual SingleResource *getSingleResource ();`
	`  ...`
	`  ScheduleAgent ();`
	`  ~ScheduleAgent ();`
	`};`
	
	ObjectStore implementation of the constructor:
	
	`ScheduleAgent::ScheduleAgent`
	`  (MasterDataAgent * theMasterDataAgent)`
	`{`
	`  ...`
	`  // Insertion of the created ScheduleAgent into`
	`  // the extent of the class`
	`  extent->insert(this);`
	`}`

Table 1: Persistent Class Definitions

As the persistent class definition is also influenced by the features storage management (persistent references) and query management, we had to cluster persistence together with these features according to step 3 of the migration framework.

The clustering with storage management is due to the fact that in ONTOS direct references are main memory pointers and behave in all respects like main memory pointers. This implies that the traversal of a direct reference requires the programmer to ensure that the referenced object is already in memory. Otherwise the program will, if lucky, terminate with an exception raised, or if unlucky, continue with unexpected values. The other possibility is to use - as in KBL - abstract 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 memory. The `Binding()` function defined for the class `Reference` returns a pointer to the referenced object, and activates it if necessary. ObjectStore, on the other hand, 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, where a pointer to a persistent object currently not in memory has an unmapped virtual-memory-address. In 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 also provides some kind of abstract references but they are mainly used for dereferencing objects in another database or between transaction boundaries. Because of the great convenience of using direct pointers we decided to replace the abstract references in the ONTOS version of KBL by direct references in the ObjectStore version of KBL.

The clustering with query management is due to the fact that ONTOS offers a so-called instance iterator as entry point to the database which allows access to all objects which belong to a specific class and its subclasses. In order to simulate this ONTOS functionality in ObjectStore each persistent class includes a static persistent class variable named `extent` of type `os_Set` containing all the instances of this class and of its subclasses, respectively. Table 1 summarizes the persistent class definitions for `ScheduleAgent` in ONTOS and ObjectStore, respectively.

5. Experience Report

In this section we report our 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 our migration framework described above. Note, neither the implementation of the migration process was carried out in a production environment nor did we emphasize on the optimization of the application. In our evaluation we concentrate on two metrics: size of the application code and involved personnel for 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 '2. The Knowledge-Based Leitstand (KBL)' the Leitstand 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 Fig. 1) and about 45000 lines of code (LoC, without comments). The total number can be divided into 27400 LoC for the Scheduling Toolkit Module, 10300 LoC for the Simulation Module, 7300 LoC for the Planning Board Module, and 800 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 we emphasized on a shallow migration rather than on a complete re-design of the application. No additional classes had to be implemented. The total LoC for the migrated application reduced to about 40000. This yields a reduction of 15% from the original size of the application. The Scheduling Toolkit Module comprises about 21000 LoC (75 %), the Simulation Module 10400 LoC (100%), and the Planning Board Module 7300 (99%). The development utilities changed little (700 LoC) (82 %). Table 2 gives a detailed overview of the size of the application based on ONTOS and ObjectStore, respectively, and their modules.

Our analysis of the application sizes shows a considerable reduction for the Scheduling Toolkit Module, while the other modules (Simulation Module, Planning Board Module, and utilities) show only minor differences in size. The reduction in size is not surprising. It mainly results 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. A code inspection showed that the Scheduling Toolkit Module contains most of the class definitions and thus benefits most from the simpler class declarations in ObjectStore. The remaining modules contain only few

class definitions. Furthermore, we noticed a much tighter integration of ObjectStore into the C++ programming language than it is the case for ONTOS. On the opposite, we had to implement extension management for persistent classes in ObjectStore. ONTOS provides the automatic management of class extensions. Each instance of a class is collected in a container with the same name as the class. ObjectStore does not support this feature; it has to be provided by the application programmer. Nevertheless, the reduction due to simpler class definitions exceeded the effort for implementing class extensions.

The above mentioned experiences focus on migrating KBL between ONTOS and ObjectStore. As mentioned above the migration between the two ooDBSs in the reverse direction - from ObjectStore to ONTOS - is more difficult. This is due to the fact that the both systems differ in their paradigm to reach persistence. Migrating from ObjectStore to ONTOS requires persistent classes to inherit from the ONTOS class `Object` and to implement additional methods (see 4.3. Mapping Strategy for the Feature 'Persistence'). Consequently, the size of the application would increase. On the opposite, migrating between ooDBSs 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.

Modules	Lines of Code		
	ONTOS Implementation	ObjectStore Implementation	%
Scheduling Toolkit	27380	20726	~ 75%
Simulation Module	10359	10424	~ 100%
Planning Board Module	7325	7288	~ 99%
Utilities	837	690	~ 82%
Total	45901	39128	~ 85%

Table 2: Application Statistics

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

Table 3: Migration Effort

Considering the personnel involved in the migration process we can distinguish three implementation phases: analysis of the application and the ooDBSs, migration analysis, and implementation of the migration. The whole migration took 14 person weeks. The analysis of the application and of the ooDBSs, and the migration analysis took about 8 weeks. Note, there has been hands-on experience with ObjectStore beforehand. The implementation of the migration was carried out in 6 weeks. Table 3 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. The experience we gained is that the above presented framework helps to reduce the implementation effort and therefore helps to increase productivity in the migration process.

6. Conclusion

In this paper we have presented a framework for the shallow migration of applications between different ooDBSs. The framework is based on several steps of analysis to develop a controlled migration strategy. Our approach was successfully tested by migrating the KBL application from ONTOS to ObjectStore. Further work in this area will investigate approaches on one hand for the

™ Versant is a trademark of Versant Object Technology Corporation

(semi-)automatic migration of applications between ooDBSs, and on the other hand for the development of strategies for the deep migration of ooDBS applications.

Acknowledgment

The authors are grateful to J. Thaler and A. Berger for their valuable help in implementing the prototype.

References

[Ahmed92] S. Ahmed, A. Wong, D. Sriram, R. Logcher; Object-oriented database management systems for engineering: A Comparison; Journal of Object-Oriented Programming, June; 1992

[Atkinson89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik; The object-oriented database manifesto; Proc. Conf. on Deductive and Object-Oriented Databases; 1989

[Brodie93] M. Brodie, M. Stonebraker; DARWIN: On the Incremental Migration of Legacy Information Systems; GTE Laboratories; TR-0222-10-92-165; 1993

[Dittrich93] K. Dittrich; Migration von konventionellen zu objektorientierten Datenbanken: soll man, muß man - oder nicht?; Wirtschaftsinformatik 35/4; 1993

[Kappel93] 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 for Applied Computer Science and Information Systems; Univ. of Vienna; 1993

[KBL92] Esprit 5161 KBL; Design, Development and Implementation of a Knowledge-based Leitstand (KBL); Deliverable Milestone 3; 1992

[KBL93] Esprit 5161 KBL; Design, Development and Implementation of a Knowledge-based Leitstand (KBL); Final Deliverable; 1993

[Meier93] A. Meier, R. Haltinner, B. Widmer-Itin; Schutz der Investitionen beim Wechsel eines Datenbanksystems; Wirtschaftsinformatik 35/4; 1993

[Shneiderman82] B. Shneiderman, G. Thomas; An Architecture for Automatic Relational Database System Conversion; ACM TODS, Vol. 7, No. 2, 1982

[Soloviev92] V. Soloviev; An Overview of Three Commercial Object-Oriented DBMSs ONTOS, ObjectStore, and O_2; SIGMOD Record, Vol. 21, No. 1; 1992

[Stonebraker90] M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, Ph. Bernstein, D. Beech; Third-Generation Database System Manifesto; SIGMOD Record, Vol. 19, No. 3, Sept. 1990

[Su81] S. Su, H. Lam, D. Lo; Transformation of Data Traversals and Operations in Application Programs to Account for Semantic Changes of Databases; ACM TODS, Vol. 6, No. 2; 1981