Management of Data Model Evolution in Object-Oriented Database Systems

C. Huemer
Institute of Applied Computer Science and Information Systems; University of Vienna
A-1010 Vienna, Austria
ch@ifs.univie.ac.at

G. Kappel
Institute of Computer Science
Univ. of Linz
A-4040 Linz, Austria
gerti@ifs.uni-linz.ac.at

S. Vieweg
Institute of Applied Computer Science and Information Systems; University of Vienna
A-1010 Vienna, Austria
sv@ifs.univie.ac.at

Abstract

Object-oriented database systems are designed to meet the requirements of advanced database applications. These requirements may evolve in the course of time and must be managed consistently at all levels of abstraction of a database system - the database level, the database schema level, and the data model level. Approaches to the management of changes at the database level and the database schema level have been investigated in the literature. Approaches to the management of changes at the data model level are still missing. In this paper we introduce a framework for handling evolution at the data model level based on database migration. Our approach consists of a detailed analysis of the involved database systems and the application's database requirements. We illustrate our framework by means of migrating a production planning and control system from the object-oriented database system ONTOS to ObjectStore*.

1 Introduction

Object-oriented database systems (ooDBS) are designed to meet the requirements of advanced database applications such as Computer Integrated Manufacturing [6, 10]. These requirements refer to complex object modelling, extended query and manipulation features, and long transactions, to mention just a few. However, these requirements may change in the course of time. Change management accomplishes the task of managing these changes consistently and provides techniques for coping with the modifications of database systems [21]. In this paper we will introduce an approach to change management at the data model level.

The controlled evolution of a database system may occur in various ways. The management of the changes may be classified along two orthogonal dimensions: the *level of abstraction* and the *transformation mode*. In other words, we can ask the following questions: What may be changed in a database system? And how are these changes managed?

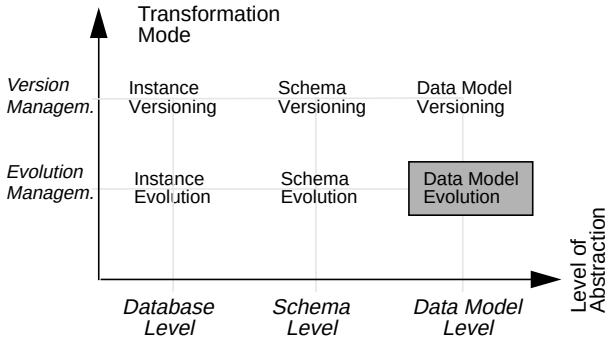

Figure 1: Aspects of Change Management

From the point of view of data modeling, each database system can be divided into multiple *levels of abstraction*. Each of these levels provides a description of the entities and their relationships at the next lower level. The levels are related in a *intension-extension dimension* [14,22]. Based on this ordering we can group the changes occurring in a database system into the following: changes at the database level, changes at the database schema level, and changes at the data model level. The second dimension is the *transformation mode* of the above mentioned changes. By transformation we denote the way the changes of the database are managed in general. We distinguish two policies to manage changes in database systems, *evolution management* and *version management*. Evolution denotes the process of change in a certain direction. In connection with ooDBS we consider evolution as the development of the ooDBS in the course of time. The original entities are replaced by the modified entities and become inaccessible after transformation. In contrast, version management (or *versioning*) describes the management of changes in a more flexible way. The history of changes in the database is recorded for retrospective access.

Following the two dimensions of our classification - abstraction and transformation - we can identify the following aspects of change management

* All products mentioned herein are trademarks of their respective manufacturers

PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON ADVANCED DATABASE TECHNOLOGIES AND THEIR INTEGRATION (ADTI'94), OCTOBER 26-28, 1994, NARA, JAPAN

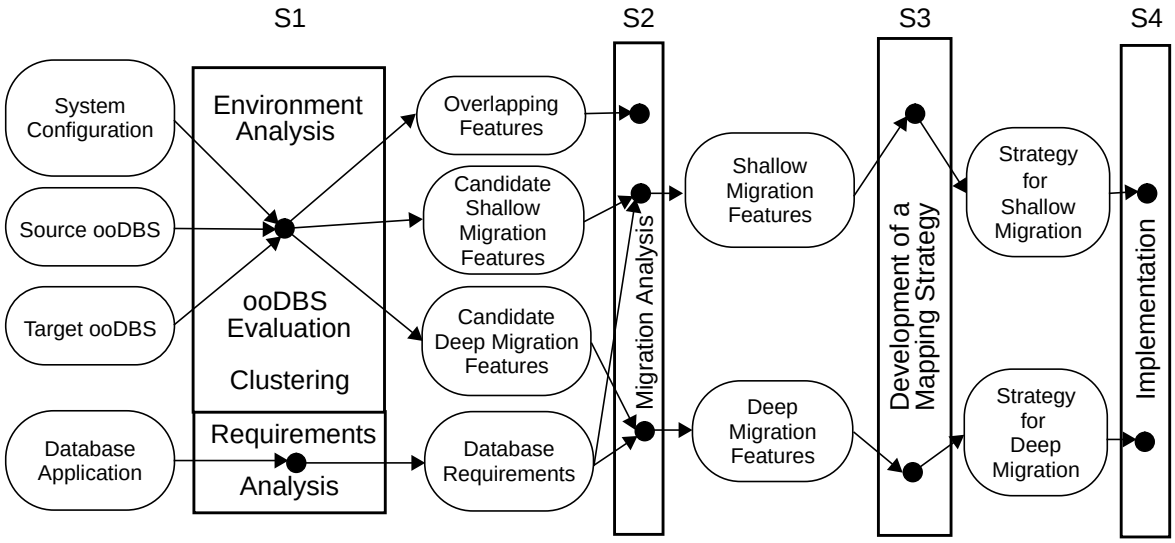

Figure 2: Model of the Migration Process

(depicted in Figure 1): instance evolution, instance versioning, schema evolution, schema versioning, data model evolution, and data model versioning.

Traditional database systems offer *instance evolution*. The database is transformed from one consistent state into another by a sequence of read and write operations arranged within transactions. As soon as a transaction reaches a committed state the original data become inaccessible. No recording of the changes in the database is performed [7]. In case of recording the database changes we face *instance versioning*. There are multiple approaches to record these changes. If changes are recorded automatically by the database system using some time-stamping mechanism we face temporal database systems [20]. However, in certain application domains such as design applications temporal database systems are not always appropriate. These applications require a logical application driven notion of history. In contrast to temporal databases, instance versioning allows for the application driven recording of database changes. A new version of a data item is generated in terms of the application's semantics; it is a "meaningful snapshot of a design object at a point in time" [11]. The evolution of requirements involves changes not only at the database level but also at the data description level. *Schema evolution* deals with the management of modifications at the schema level and the resulting necessary modifications at the database level [2,23]. As the old schema is inaccessible after evolution, the instances must be converted to conform to the new schema. *Schema versioning* extends schema evolution in such a way that old and new schemas are supported simultaneously, thus old instances need not be converted [3,4,17].

With *data model evolution* and *data model versioning* we describe the management of changes at the data model level. Data model evolution is the management of changes with loss of the old data model information. Data model versioning extends data model evolution in the same way as schema versioning extends schema evolution. The old data model remains valid for each of the subsequent levels. The same holds true for database schemas and the instances that follow the old data model.

Changes at the database level and at the schema level, especially for ooDBS, have been extensively investigated in literature. Approaches to the management of changes at the data model level for ooDBS are still missing. The purpose of this paper is to fill this gap. We will present an approach to data model evolution in ooDBS based on a database migration framework - a pragmatic yet feasible approach to data model evolution. Our approach is illustrated by means of a case study.

Database migration is the process of mapping a database application from a *source database system* to a *target database system*. The migration process consists of a set of conversion operations or conversion techniques that are applied to the source application and which results in a target application [8]. Migrating from source database systems to a target database system is thus part of the maintenance process of database applications. Traditional (i.e. relational) approaches to database migration are based on the distinction of the database interaction part and the computation part of a database application [16,19]. Due to the inherent

integration of these two parts in ooDBS and thus reducing the impedance mismatch, traditional approaches fail when applied to ooDBS and thus motivate the development of new techniques.

In ooDBS, two approaches to migration are possible: shallow migration and deep migration [5]. 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 only requires a re-design of those database components of the application which have a different semantics in the source ooDBS and the target ooDBS or which are missing in the target ooDBS and, hence, have to be simulated. In our migration framework we consider both migration techniques.

The data models of the source DBS and the target DBS need not necessarily be related by an evolutionary modification step such as the release of a new product version leading to new data model features. Rather the data model of the target DBS may be completely different selected in the course of continual requirements analysis of the application domain leading to the transition from one database product to another. Although based on different assumptions these two tasks are related and result in the need for migration concepts.

This paper is organized as follows. In Section 2 we present a framework for the migration of object-oriented database applications. In Section 3 we will discuss experiences with the presented framework gathered during a migration project. We conclude with a summary and an outlook on further research that has to be done in this area.

2 Migrating Object-Oriented Database Applications

In this section we present a general outline for data model evolution. The proposed framework describes the single phases which have to be performed when migrating from one data model to another. We informally describe what has to be done in each of the migration phases and define a useful order of the steps.

The framework helps to systematically select those data model features which have to be considered for the migration leading to an implementation of the necessary code changes. For this purpose, we analyze all the components involved in the migration process - the *source ooDBS*, the *target ooDBS*, the *system configuration*, and the *database application* - based on the same set of criteria. The analysis of the involved ooDBS is based on a set of criteria developed from [1,9,10,18].

The subtasks of the migration framework may be grouped into the following four phases: *analysis and evaluation*, *migration analysis*, *development of a mapping strategy*, and *implementation*. The input and output dependencies of these phases are depicted in Figure 2. In the following we will describe these phases in detail.

Analysis and Evaluation (S1)

Since the source ooDBS and the target ooDBS are evaluated for a specific hardware and software environment, 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 the features of the two ooDBS leading to a set of features supported by each system. Furthermore, a comparison of these features will result in three subsets (which correspond to the three areas within the dark borders shown in Figure 3) according to their effect on the migration.

- **Overlapping features** are syntactically and semantically identical in both ooDBS and - as a consequence - 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 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 of another. Since related changes should be considered together, we propose to cluster features which require associated changes. Each of the so-called *clusters of interdependent functionalities (CIF)* forms a separate migration unit, which can be migrated independently from the features outside of this specific cluster. The membership in the clusters is independent from their membership in the set of shallow or deep migration features. Membership in the CIF is due to code dependencies that must be considered during the implementation of the migration whereas membership in the sets of shallow and deep migration features is determined by the semantics of the features provided by the source

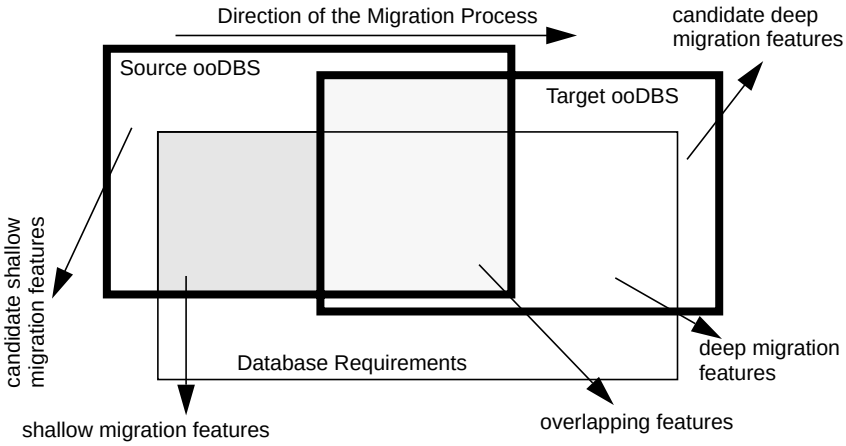

Figure 3: ooDBS Features and Database Requirements

and the target ooDBS, respectively.

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 collect those features which have to be considered for migration.

Migration Analysis (S2)

During *migration analysis* we compare these database requirements with the result of the DBS evaluation, which is represented by three different sets of features as outlined in Figure 2. The main task is to find the features which are relevant for the migration and to remove those features which are not required by the application or which are provided by both ooDBS in the same manner (overlapping features). Therefore, the result of the migration analysis is the set of shallow migration features, which are defined by intersecting the database requirements with the candidate shallow migration features, and the set of deep migration features, which are defined intersecting the database requirements with the candidate deep migration features.

Development of a Mapping Strategy (S3)

For the features being member of the set of shallow or deep migration features a *strategy for their implementation* into the target ooDBS must be developed. According to the two different kinds of migration features, this step results in a *mapping strategy for shallow migration* to obtain the same database functionality for the application as it was available from the source ooDBS and in a *mapping strategy for deep migration* to extend the database functionality.

Implementation (S4)

The *implementation* process includes the *re-engineering of the code* according to the developed mapping strategies and furthermore the *evaluation* of the target application (= the application running on top of the target ooDBS).

In the following we will present our experiences with the migration framework presented above.

3 Case Study: Data Model Evolution in KBL

In this section we will describe the migration of the KBL application (Knowledge-Based Leitstand), a prototype system for interactive production planning and control. KBL was developed within an ESPRIT project and is based on the object-oriented paradigm. It is implemented on top of the ooDBS ONTOS. We used the framework described above to migrate KBL from ONTOS to ObjectStore. We refer to [12,13] and to the product literature for a further description of the Knowledge-Based Leitstand and the involved ooDBS, respectively.

Analysis and Evaluation (S1)

The *application's database requirements* of KBL can be identified as the following [10,15]: 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 controlling the scheduling information in the knowledge base. When given the tight integration of the knowledge base and the Leitstand logic a Leitstand must provide both *navigational access* of highly interrelated data and *associative access* by querying collections of data. *Version management* is required for the simulation of different schedules of work orders. Since a Leitstand system is integrated with other components such as the shop floor management *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.

Migration Analysis (S2)

In this subsection we will discuss the second phase of the migration process, namely the migration analysis for migrating KBL from ONTOS to

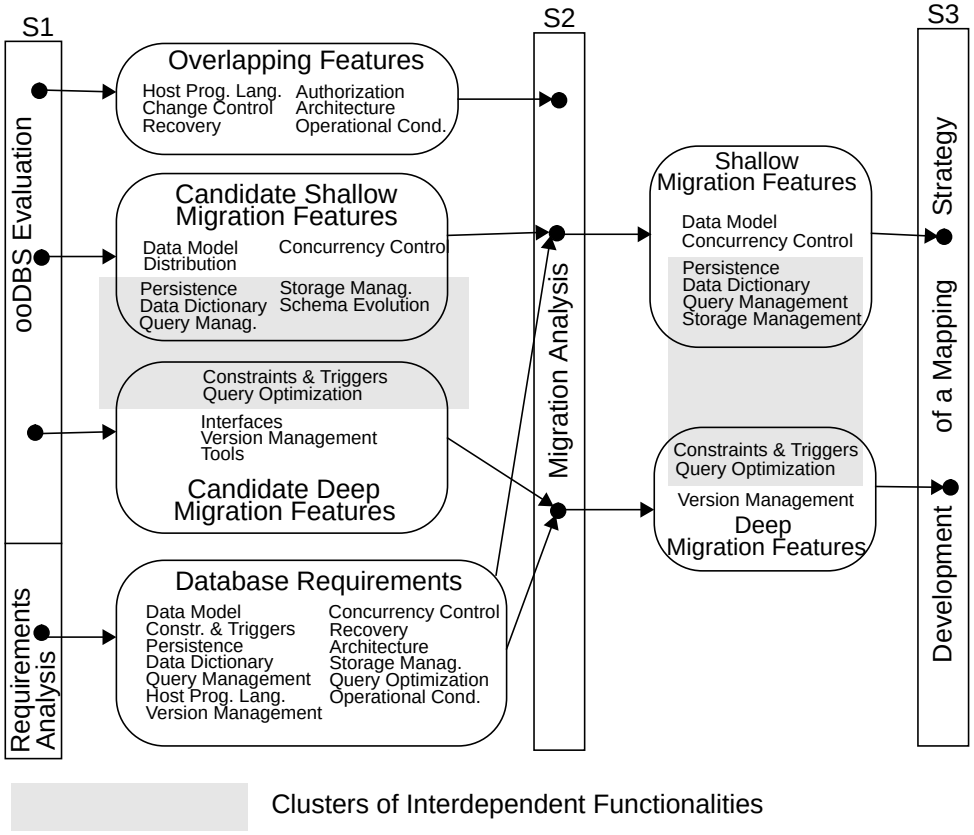

Figure 4: Migration Process of KBL from ONTOS to ObjectStore

ObjectStore. Our investigations in the course of this specific migration are based on the main topics of the evaluation schema of [9]. Figure 4 presents the grouping of the main topics of the evaluation schema into the various sets of migration features. For brevity, we will only explain the most important topics, namely *data model*, *persistence* and *query management.*

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, 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. Concerning additional 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.

Query management is implemented differently in both systems. In ONTOS it is possible to provide each object with a synonym which can serve as a unique identifier stored in a separate data dictionary. Persistent objects can be retrieved via these 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. All other objects stored in the database can be reached by navigational access. Due to these 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.

Persistence is required by any database application. In ONTOS persistence is reached by inheritance. Therefore, each persistent object must be an instance of a class which directly or indirectly inherits from the ONTOS specific persistent root class `Object`. Writing objects to disc require an explicit call of the `Object`'s method `putObject()`. Furthermore, each persistent class definition in ONTOS must include some additional ONTOS specific methods, e.g. to activate/deactivate an object to/from disc. Since in ObjectStore persistence is reached by declaration a persistent object can be an instance of any class. It has simply to be declared persistent or to be created with a persistent constructor. There is no need to call an operation like `putObject()` to write it to secondary storage nor to equip the class definitions with additional methods. Since these two paradigms to implement persistence are completely different we have to include *persistence* in the shallow migration features and have to develop a mapping strat-

egy.

Development of a Mapping Strategy and Implementation (S3, S4)

In this subsection we will show the development of a mapping strategy for the main features *query management* and *persistence*.

Query management in ObjectStore neither offers an instance iterator nor synonyms for object retrieval. In order to simulate the instance iterator 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 all subclasses, respectively. Therefore, the constructor has to include a call which inserts the created object into this static set and the destructor must include a call which removes the deleted object from this set. The synonyms have been simulated by embedding an identifying property (char *Name) into the root class KBLObject of the KBL application. Since these measurements effect the persistent class definitions in ObjectStore (see further below) the features *query management* and *persistence* reside in the same cluster of interdependent functionality (CIF).

The fact that *persistence* is reached by declaration simplifies the persistent class definition in ObjectStore. In ObjectStore we do not have to care about a derivation path from some system defined root class and we can remove all additional methods included in the ONTOS persistent class definitions.

Below we present the persistent class definitions in ONTOS and in ObjectStore and 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.

ONTOS class definition

```
class KBLObject : public Object
// KBLObject directly inherits from Object
{
  ...
};
class IntervalObject : public KBLObject
// IntervalObject indirectly inherits from Object
{
  ...
};
class ScheduleAgent : public IntervalObject
// ScheduleAgent indirectly inherits from Object
{
private:
  Reference ivMasterDataAgent;
  // indirect Reference to a MasterDataAgent object
  Reference ivResource;
  // indirect reference to a Resource object
  // Constructor which is called by the
  //constructor of the related MasterDataAgent
  ScheduleAgent (MasterDataAgent *);
  // ONTOS required function
  virtual void deleteObject
     (Boolean deallocate = FALSE);
  //delete the object from database
public:
  // returns a pointer to the related
  // MasterDataAgent object
  virtual MasterDataAgent
     *getMasterDataAgent ();

  // schedules 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 *getSingleResrce ();

  // returns a list including the ScheduleAgent
  //objects already scheduled
  List* ScheduleAgent::getScheduledSAList();
  ScheduleAgent (); // constructor
  ...
  // ONTOS required functions
     ScheduleAgent (APL *);
  // activation constr. to activate the object from disk
  ~ScheduleAgent ();          // destructor

  // ret. a pointer to the object repr. the class info
  virtual Type *getDirectType ();
  virtual void Destroy
     (Boolean aborted = FALSE);
  // deactivate the object from main memory
  virtual void putObject(Boolean deallocate
     = FALSE);   // write the object to the database
};
```

ObjectStore class definition

```
extern os_database *db;
class KBLObject
{
public:
  // static persistent set which includes all instances
  // of KBLObject
  persistent<KBLdb>
     os_Set<KBLObject*> * extent;
  ...
  char *ivName; // impl. of the object naming
  ...
};
```

```
class IntervalObject : public KBLObject
{
public:
  // static persistent set which includes
  // instances of IntervalObject
  persistent<KBLdb>
     os_Set<IntervalObject*>* extent;
  ...
}
class ScheduleAgent : public IntervalObject
// neither IntervalObject nor ScheduleAgent
// inherit from any predefined class
{
private:
  MasterDataAgent *ivMasterDataAgent;
  // direct reference to a MasterDataAgent object
  STResource *ivResource;
  // direct reference to a Resource object
  ScheduleAgent (MasterDataAgent *);
  // constr. as in ONTOS but implemented differently

public:
  // static persistent set which includes all
  // instances of ScheduleAgent
  persistent<db>
     os_Set<ScheduleAgent*>* extent;

  virtual MasterDataAgent
     *getMasterDataAgent (); // as in ONTOS
  virtual void putSingleResource
  (SingleResource *); // as in ONTOS
  virtual SingleResource
     *getSingleResource (); // as in ONTOS
  os_List<ScheduleAgent*>
     *ScheduleAgent::getScheduledSAList();
  ...
  ScheduleAgent ();     // constructor
  ~ScheduleAgent ();    // destructor
  // no system required methods
};
```

// ObjectStore implementation of the constructor:

```
ScheduleAgent::ScheduleAgent
(MasterDataAgent * theMasterDataAgent)
{
  ...
  // Insertion of the created ScheduleAgent
  // object into the class extent
  extent->insert(this);
}
```

ONTOS implementation:

```
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 in the iteration; 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->getSingRes)
          != 0
    scheduledSAList->Insert
          (Entity*) scheduleAgentIterator);
  }
  return scheduledSAList;
}
```

ObjectStore implementation:

```
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 spec. argument
  foreach(currentScheduleAgent,
ScheduleAgent::extent) {
// selection criteria like in ONTOS
if (currentScheduleAgent->getSingleResource)
  != 0)
     scheduledSAList->
     insert(currentScheduleAgent);
  }
  return scheduledSAList;
}
```

4. Experiences & Further Research

The original KBL application consists of several modules comprising 60 classes and about 45000 lines of code (LoC, without comments). Due to the fact that we carried out a shallow migration rather than a complete re-design 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 15% compared with the original size of the application. The reduction 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. Furthermore, we noticed a much tighter integration of ObjectStore into the C++ program-

ming language than it is the case for ONTOS. On the contrary, we had to implement extension management for persistent classes in ObjectStore since ObjectStore does not support this feature. Nevertheless, the reduction due to simpler class definitions exceeded the effort for implementing the management of class extensions.

The experiences mentioned above concern the migration of KBL from ONTOS to ObjectStore. Migrating applications between the two ooDBS in the reverse direction - from ObjectStore to ONTOS - is more difficult. This is due to the fact that the two systems differ in their paradigm to implement persistence. Migrating from ObjectStore to ONTOS requires persistent classes to inherit from the ONTOS class Object and to implement additional methods. Consequently, the size of the application would increase. On the contrary, migrating between ooDBS that follow a similar approach to reach persistence (e.g. ONTOS and Versant) requires less effort than the former case and probably has minor effects on the size of the application code.

Considering the personnel involved in the migration process we can distinguish three implementation phases: analysis of the application and the involved ooDBS, migration analysis, and implementation of the migration. The whole migration process was carried out in 14 weeks (person-weeks). The analysis of the application and of the ooDBS together with the migration analysis took about 8 weeks. Note, that there was hands-on experience with ObjectStore beforehand. The implementation of the migration was carried out in 6 weeks. In the initial planning of the migration we scheduled a bigger effort for the actual implementation process. The experience we gained is that the framework presented above helps to focus on the main efforts and therefore leads to increased productivity in the migration process.

Further investigations in this topic will include approaches for the (semi-) automatic migration of ooDBS applications, the development of strategies for deep migration, and approaches to data model versioning based on this approach.

References

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

[2] J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, H. Kim; Data Model Issues for Object-Oriented Applications; ACM TOIS; Vol. 5, No. 1; 1987

[3] A. Björnerstedt, C. Hultèn; Version Control in an Object-Oriented Architecture; in: [Kim89a]; 1989

[4] H. Chou, W. Kim; A Unifying Framework for Version Control in a CAD Environment; Proc. of the VLDB Conf., 1986

[5] K. Dittrich; Migrating from conventional to object-oriented databases: a “can”, a “must” - or none of both? Wirtschaftsinformatik, 35/4; 1993

[6] J. Encarnação, P. Lockemann; Engineering Databases, Connecting Islands of Automation Through Databases; Springer Verlag; 1990

[7] J. Gray, A. Reuter; Transaction Processing: Concepts and Techniques; Morgan Kaufmann; 1993

[8] C. Huemer, G. Kappel, S. Vieweg; Migrating a Leitstand System between Object-Oriented Database Systems - An Experience Report; Proc. of the STAK’94 Conf.; 1994

[9] 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; Inst. of Applied CS and Information Systems; Univ. of Vienna; 1993

[10] G. Kappel, S. Vieweg; Database Requirements for CIM Applications; to appear in Journal of Integrated Manufacturing; 1994

[11] R. Katz; Toward a Unified Framework for Version Modeling in Engineering Databases; ACM Computing Surveys, Vol. 22, No. 4, 1990

[12] Esprit 5161 KBL; Design, Development and Implementation of a Knowledge-based Leitstand (KBL); Deliverable Milestone 3; Commission of the European Community (CEC), 1992

[13] Esprit 5161 KBL; Design, Development and Implementation of a Knowledge-based Leitstand (KBL); Final Deliverable; Commission of the European Community (CEC), 1993

[14] L. Mark, N. Roussopoulos; Metadata Management; IEEE Computer, Vol. 19, No. 12, 1986

[15] U. Schreier; Database Requirements of Knowledge-based Production Scheduling and Control: A CIM Perspective; Proc. of the VLDB Conf.; 1993

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

[17] A. Skarra, S. Zdonik; Type Evolution in an Object-Oriented Database; in: B. Shriver, P. Wegner (eds.); Research Directions in Object-Oriented Programming; MIT Press; 1987

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

[19] 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

[20] A. Tansel et.al.; Temporal Databases; Theory, Design, and Implementation; Benjamin/Cummings Series on database systems and applications; Redwood City, CA; 1993

[21] C. Thompson; DARPA Open Object-Oriented Database Preliminary Module Specification, Change Management Module; Version 2, March 23, 1993

[22] S. Vieweg; Managing Evolving Requirements in Object-Oriented Database Systems; Ph. D. Thesis; Dept. of Information Engineering; Univ. of Vienna; 1994

[23] R. Zicari; A Framework for Schema Updates in An Object-Oriented Database System; Proc. of the IEEE Data Engineering Conf.; 1991