Functionality for Object Migration Among Distributed, Heterogeneous, Autonomous DBS

Elke Radeke
Cadlab
Cooperation University of Paderborn & SNI AG
Bahnhofstr. 32, 33102 Paderborn, Germany
elke@cadlab.de

Marc H. Scholl
University of Ulm
Faculty of Computer Science
89069 Ulm, Germany
scholl@informatik.uni-ulm.de

Abstract

In current enterprises, data is distributed over a multitude of heterogeneous, autonomous database systems. These systems are often isolated and an exchange of data among them is not easy. On the other hand, decreasing time-to-market periods and raising techniques like Concurrent Engineering require good support in *dynamic change of data location in various granularities*. Data will not reside in the database system of a project group or department all the time but need to be moved or duplicated concurrently to others. To fulfill this industrial requirement, we develop functionality enabling different granularities of data (object) migration *among multiple database systems*. The underlying architecture and concepts are being derived from a requirement analysis and extend a federated DBS approach.

1 Introduction

Database systems are essential components of today's information systems. For more than two decades, **different database systems** have been built or acquired in the enterprises. Due to various reasons they are distributed on several computers, support different data models, and are difficult if not impossible to merge. The *technical reason* is that no DBS is well appropriate to all application domains, e.g. an electronic circuit database must provide fast access to complex circuit diagrams for simulators and routers while a price database has to provide fast access to individual records for billing applications. Moreover, it has *economical reasons* because there are various DBS on the market supporting the same application domain but differing in service and price. Often there are also *organizational reasons* when any department

can choose on its own a DBS without a global enterprise strategy. So enterprises live and will live in a world of distributed, heterogeneous, and autonomous database systems.

During several years, enterprises made huge investments for their various database systems: many database tools and programs are self-implemented or acquired and gigabytes of data are stored in the databases. But in most enterprises, links among these distributed, heterogeneous, and autonomous systems are missing or only in the minds of the DBS users.

Nevertheless, users and the whole enterprise will **benefit from functionality enabling the migration** of data/objects among the various DBS:

- It enhances the *availability* of data, e.g. by replicating data. Techniques like *Computer Supported Cooperative Work* (CSCW) [9] or *Computer Aided Concurrent Engineering* (CACE) [2] are supported by making data of project groups or departments with a specific DBS available to others with different DBS. These techniques are becoming important in enterprises in order to decrease time-to-market periods.
- It supports the *migration of applications* to other DBS by transferring their existing data to the target DBS.
- A controlled *reduction of the number of DBS* is eased [12]. Legacy systems can be eliminated or DBS supporting the same application domain may be combined in an enterprise and object migration serves for transferring still relevant data to some remaining DBS.

Our approach provides data/object migration functionality in a software layer on top of the multiple DBS. It extends the approach of a federated database system (FDBS) with migration capabilities. Users can dynamically transfer data of various granularities among the DBS of an enterprise or cooperation.

In contrast to gateway database systems offering the importation of foreign data from specific other DBS, we enable the migration of objects among *arbitrary* database systems, no matter if they possess a gateway feature or not. Additionally, the federation approach allows to globally control the migrated objects, e.g. to guarantee data consistency between replicas, while most gateway systems do not control redundancy between connected database systems. Hence, the risk for data inconsistency decreases. Since our approach may also preserve global identity of migrated objects, existing global FDBS applications do not have to be recoded when some object moves from one DBS to another. On the other hand, gateway systems do not provide object identity spanning over all connected database systems.

Also tools supporting the migration from one isolated database system to another [3, 10] are

restricted to a given set of database systems, here only two. This makes object migration across all database systems quite difficult for an enterprise. It requires different migration tools and, hence, often different techniques. In contrast, our FDBS approach offers a uniform object migration mechanism to migrate data among multiple database systems. Moreover, real world entities that are split over multiple DBS can be transferred as a single object to some DBS.

In some (homogeneous, non-autonomous) distributed database systems where a single DBS is distributed over multiple sites (e.g. computers), there are also mechanisms to move or replicate data from one site to another [11, 4]. But the purpose is different to that in FDBS. In distributed database systems, object migration is realized automatically according to access statistics in order to speed up data access. Such an automatic change of object locality, in general, is not desirable for autonomous DBS. It could eliminate objects from a DBS which are still required by some of its local applications. Instead we allow to move, replicate, and copy objects on user demand among the DBS. Nevertheless, we adopted migration concepts from distributed DBS, e.g. the feasibility to deeply migrate complex objects [4]. Due to a different architecture of FDBS which also considers autonomy, differentiated concepts are required. For example, different data visibilities result in various migration degrees.

The **structure of this paper** is as follows: Section 2 lists requirements for object migration among DBS according to the characteristics distribution, heterogeneity, and autonomy. A corresponding architecture which couples the DBS for the migration process is derived in Section 3. Section 4 introduces the migration concept of our approach and corresponding migration functionality is developed in Section 5. The paper is concluded in Section 6 and an outlook on future activities is given.

2 Requirements

Distribution, heterogeneity, and autonomy pose various requirements on object migration among DBS. This section lists general coupling requirements (CR) and specific migration requirements (MR) for these three DBS characteristics. General coupling requirements occur because source and target DBS must be coupled somehow for the migration process. They are also valid for data access across DBS (but are not complete for any kind of data access). In addition, we identify requirements specific for object migration as a specific form of data access across DBS.

Distribution

Data are distributed over *multiple databases*. Some of them may be stored redundantly in several databases or split over multiple databases. So one real world entity is mapped to data of one or many databases. The same holds for meta data.[1] Distribution raises the following requirements:

MR1: Migration shall support both a movement of data among databases and a duplication.

MR2: In case an object was already stored redundantly in source and target then omit data transfer during migration by default.

CR1: Control of redundant data across DBS (as consequence of MR2).

MR3: Data which is duplicated during migration (MR1) shall be either treated as replicas with redundancy control or as independent copies without such control.

MR4: A migrated object must have its meta data as prerequisite in the target database. In case of absence, the target schema has to be extended.

MR5: Real world entities with data split over multiple databases can change their data location either for all their data or for some of them.

MR6: If the coupling software offers transparent access on multiple databases, then object migration has to preserve the real world entity at the coupling interface. Neither its data/relationships nor their global visibility have to change by the migration of an object.

Heterogeneity

Source and target DBS may *differ in hardware* or *operating software*, in the *data model*, the *schema* syntax and semantics, as well as the *data*.

CR2: Heterogeneous hardware and operating software require the agreement on a common data exchange.

CR3: Heterogeneous data models require a mapping among the data model elements, i.e. between their data description elements and operations.

CR4: Heterogeneous schemata require a mapping among the schemata, i.e. among the meta data. Thereby not only the syntax of the schemata is relevant but also their semantics.

MR7: If an object migrates between two heterogeneous DBS, it requires an equivalent to its

[1] By separating the properties distribution and heterogeneity, we only consider data distribution over multiple databases in this paragraph but do not mean distribution over different hardware or operating software which is often meant by distributed DBS.

meta data in the target schema. If no such equivalent meta data is defined in the schema mapping (see above), the target schema as well as the schema mappings have to be extended.

Autonomy

Database systems retain their *separate and independent control.* Each DBS determines its own data representation and functionality, decides what is visible to other DBS, chooses when and how to communicate with others, when to execute external operations and may abort external operations if, for instance, local constraints are not met.

CR5: When the DBS chose different/heterogeneous kinds of data and functionality then this heterogeneity has to be solved (see above).

CR6: The DBS have to notify what data is accessible and what functionality may be used.

CR7: The DBS have to agree on a common communication.

MR8: Object migration has to leave consistent data in source and target DBS.

These requirements determine our architecture and concept for object migration among distributed, heterogeneous, autonomous DBS which are presented in the following sections.

3 Architecture for Coupling Multiple DBS

In order to migrate data among various DBS, the databases have to be coupled somehow. There are two alternative architectures to achieve this (Fig. 1): (1) a coupling layer for each DBS realizes the mapping to the others or (2) a single coupling layer with a canonical data model maps to all DBS back and forth. In both alternatives, users can access the DBS directly via its DBS interface or use the coupling layer in order to access data of multiple/other DBS.

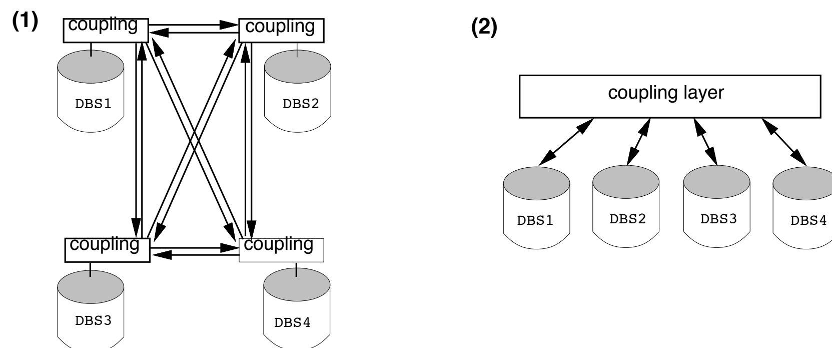

Figure 1: Alternative architectures:(1) point-to-point-connections, (2) canonical coupling layer

The first alternative requires $n * (n - 1)$ mappings between DBS with n be the number of DBS while the second alternative needs n mappings. But current enterprises have a multitude

of data management systems, so n is big (e.g. 20 for a department of an industrial project partner [12]). Moreover their distribution, heterogeneity, and autonomy pose many tasks to the mapping as we saw in the previous section, e.g. data model and schema transformation as well as redundancy control. Therefore we chose the second alternative for our architecture in order to decrease implementation effort and ease extensibility with further DBS.

Fig. 2 presents our architecture in more detail. It represents a federated database system (FDBS) [15] preserving the autonomy of the distributed heterogeneous component database systems (CDBS). In the following, we will show how this architecture fulfills the general coupling requirements (CR) of the previous section. The migration requirements (MR) will be considered in the subsequent section.

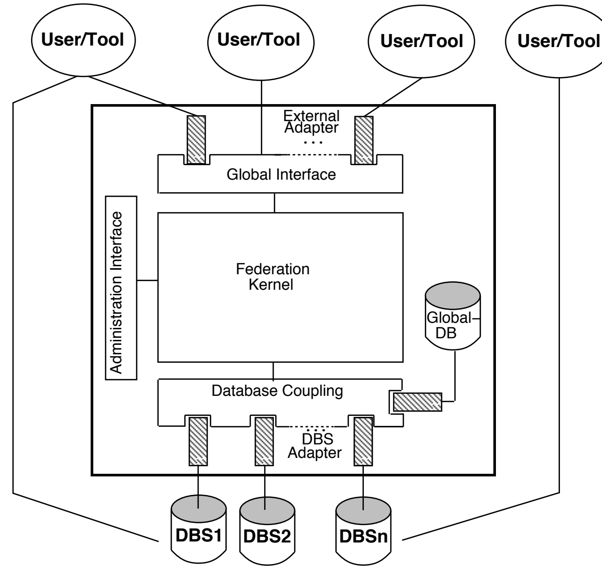

Figure 2: FDBS architecture

The *DBS adapters* of the coupling layer realize a notified communication with the various CDBS (CR2, CR7), e.g. with hierarchical, network, relational, and object-oriented DBS, or file systems. Moreover they transform the heterogeneous data and operations into a canonical data model (CR3, CR5). Our approach considers an object-oriented canonical data model due to its expressive power [14]. It is based on the upcoming standard for object oriented database systems, the ODMG object model [1] which provides concepts like objects, classes, class inheritance, object identity. To ease the dynamic extension of the federation with new DBS, there is a uniform layer (*DB coupling*) on top of the DBS adapters.

Data are accessed in the FDBS either locally by the DBS interfaces of the autonomous DBS or globally via the *global interface*. Latter one is an extension of the ODMG C++ interfaces

(ODL, OML, OQL), a.o. added by functionality for schema mapping (CR4, CR6), global redundancy identification, and object migration. It allows a uniform and transparent access to data of all CDBS. Other interfaces are realized as *external adapters* on top of it, e.g. application specific interfaces.
Meta information as well as global data that do not map to any DBS are stored in an auxiliary database, the *Global-DB*.
By the *administration interface*, the FDBS is started, stopped, initialized, and DBS are coupled/decoupled to/from the FDBS. The *federation kernel* realizes tasks such as query decomposition, global transaction management, and global redundancy control (CR1).

4 Concept for Object Migration

To support object migration among distributed, heterogeneous, autonomous DBS, we extend the federation approach with corresponding functionality. The migration is invoked via the global interface of the FDBS and is called "object" migration because we use a canonical object-oriented data model. By object migration, objects can be migrated among arbitrary component database systems.
The functionality for object migration in FDBS is based on the migration framework we developed in [13]. We summarize briefly its concepts in this section before deriving the migration functionality. Important characteristic is that compatibility to existing applications which access the global/external interface of the FDBS is preserved, a change of data locality by migration will be transparent to them (MR6).

4.1 Base Model

During object migration, the object's data attributes are transfered in a consistent state from some source CDBS to a target CDBS (MR8). The object's class or an equivalent class is required as prerequisite in the target CDBS for this (MR4, MR7). By default, only single objects are migrated so that relationships and related objects are not transfered. In this paper, we extend this by also considering relationships and related objects during migration optionally (Section 5.2.2).

Example: An FDBS example containing two sales database systems, one for each sales district, will illustrate object migration for customer data (Fig. 3).

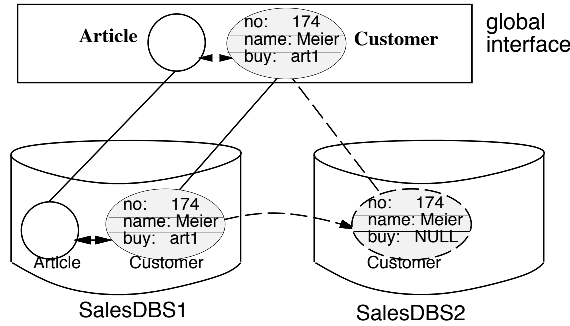

Figure 3: Object migration among database systems of an FDBS

The customer object is of global class *Customer* which was mapped during schema transformation 1:1 to the local class *Customer* of SalesDBS1 and SalesDBS2. It has two data attributes *name* and *no* as well as relationships to some article object representing the bought article. Previously, the customer was associated to sales district 1 and all his data attributes and relationships were stored in *SalesDBS1*. When he moves to another town associated with the second sales district, his attributes *name* and *no* are transfered from SalesDBS1 to SalesDBS2 during object migration. The class of the object, *Customer*, need not be migrated because an equivalent local class exists in SalesDBS2.[2] Related objects are not migrated by default so that the objects relationship will not be visible in the target CDBS. However, at the global interface it remains accessible and object migration is transparent to the global applications. In case a migration operation transfers the customer object to the target CDBS and afterwards deletes the customer together with its relationship (referential integrity), the relationship becomes an inter-database relationship and is stored in the Global-DB.

4.2 Migration Dimensions

In order to differentiate "how" an object is migrated and what data are considered during object migration, there are three orthogonal dimensions:

Object Kind

An analysis of the assignment between globally accessible objects and local CDBS objects resulted in the following object kinds within an FDBS (Fig. 4):

1. *Globally new*: global object not assigned to a CDBS object but stored in Global-DB
2. *Globally invisible*: local object of some CDBS not assigned to any global object

[2] Class migration is treated as a separate issue which is out of the scope of this paper.

3. *Federated*: global object assigned to some local objects without filtering data
   (a) *unique*: assigned to exactly one local object of a CDBS
   (b) *multiple*: assigned to equivalent local objects of multiple CDBS
   (c) *union*: composed of local objects of multiple CDBS/Global-DB (may overlap)
4. *Reduced federated*: similar to federated objects with subclasses (a) unique, (b) multiple, (c) union, but filtering is allowed (see filter processor in [15]). Thus not all data is visible in the global object of at least one assigned local object.

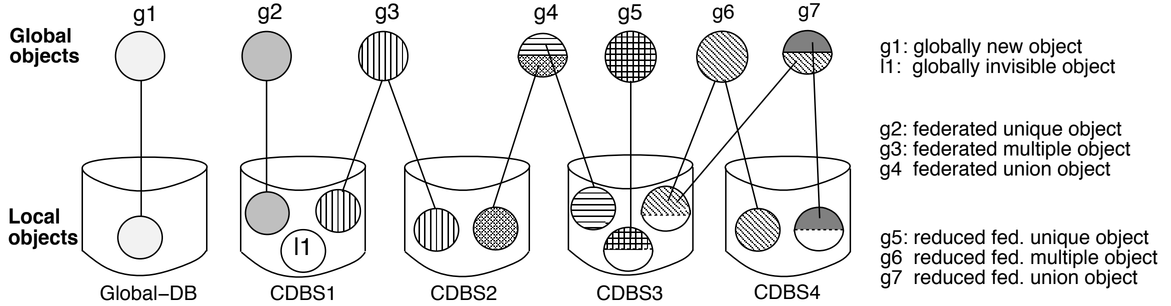

Figure 4: Object kinds in an FDBS

Migration Degree

The second dimension for object migration specifies how much of a global object shall be migrated (MR 5). It contains the following alternatives:

A) *Partial*: All globally visible data of a global object stored in a given source CDBS is migrated to a target CDBS.

B) *Locally complete*: For a global object, both globally visible and invisible data are migrated from a given source CDBS to a target CDBS.

C) *Globally complete*: All globally visible data of a global object is migrated from all CDBS to a target CDBS.

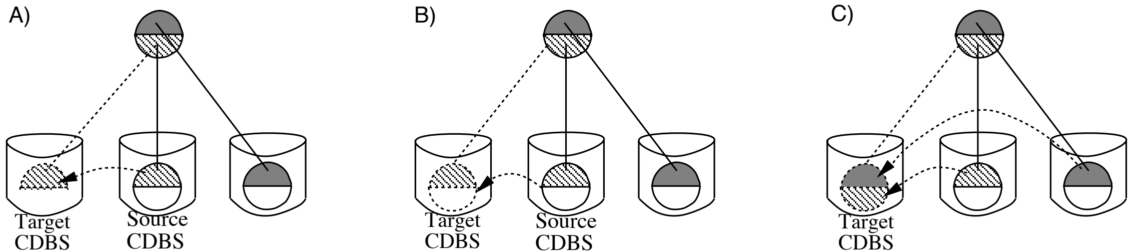

Figure 5: Migration degrees: A) partial, B) locally complete, C) globally complete

Operation Primitive

The third dimension defines how an object migrates and offers as alternatives (MR1, MR3):

1) *Absolute movement*: Objects are transferred into the target and deleted in the source CDBS. Global identity of the object remains the same at the global interface.

2) *Replication*: Objects are transferred to the target CDBS but not deleted in the source CDBS. The duplicates have the same global object identity and the FDBS may guarantee data consistency between them [16].

3) *Independent copy*: Objects are also duplicated to the target CDBS, but get a different global object identity. Data consistency is not guaranteed between the duplicates.

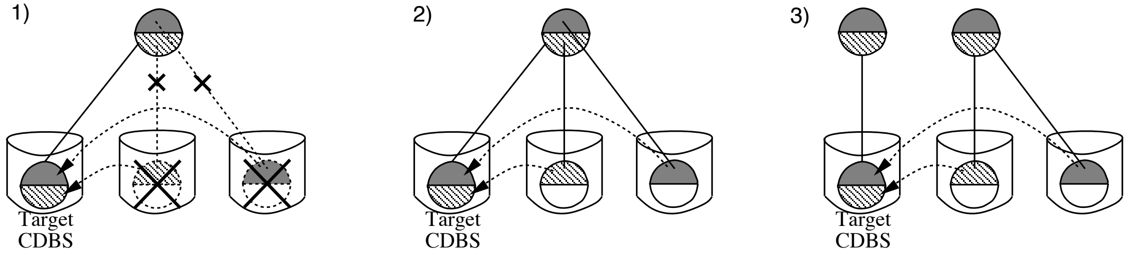

Figure 6: Operation Primitives: 1) absolute movement, 2) replication, 3) independent copy

The three migration dimensions determine the various capabilities of the object migration functionality specified in the next section. Thereby an important combination of migration dimensions allows a reduction of redundancy across DBS: if some object was already stored redundantly in source and target CDBS, i.e. is a (reduced) federated multiple/union object, the operation primitive 'absolute movement' will remove the data in the source DBS and will not duplicate it into the target again (MR 2). Also operation primitive 'replication' will not duplicate the data but will retain it in the source CDBS.

5 Interface Functions for Object Migration

We provide the users/administrators with a set of generic operations to invoke object migration via the global interface of an FDBS. Their formal semantic is defined by the migration dimensions. We offer both base operations for migrating single objects and advanced operations to migrate multiple objects at once. This functionality supports an enterprise with dynamic migration of data in various granularities.

5.1 Base Operations

We distinguish two kinds of migration operations: implicit and explicit. *Implicit migration operations* enable object migration by changing an objects class. These operations are already

known from existing DBS, e.g. COMIC [7] and COCOON [17]. In FDBS, they require an object migration if the classes are mapped to different CDBS during schema integration. Although an FDBS approach, namely O*SQL [8], includes class change operations, the underlying object migration was not elaborated. We fill this gap by specifying class change operations in terms of our object migration dimensions. While these operations allow a transparent object migration, they restrict on the operation primitives absolute movement / replication, on the migration degree globally complete and, moreover, require specific type/class mappings. Therefore we add *explicit migration operations* which enable the specification of all migration dimensions (also configurable) but in most cases do not allow transparent object migration. Both implicit and explicit operations together result in a flexible object migration mechanism.

In the following, we specify the migration operations and extend the global interface with corresponding functionality. For each operation, a programming language independent specification is presented in the specification notation IDL used by ODMG [1]. Then its usage is illustrated by an example for a C++ binding which extends the ODMG C++ OML by corresponding methods.

5.1.1 Implicit Migration Operations

Implicit migration operations change the objects class. They implicitly invoke an object migration for those cases where source and target class are mapped to different CDBS. We require that both classes are defined in the application schema (federated/external schema according to the 5 level schema architecture of [15]) and have a common superclass. During object migration all common data of both classes are transferred from the object from source class to target class. Attributes of the target class with no corresponding in the source class are initialized with default values (e.g. NULL).

Example: Assume a university FDBS contains a StudentDBS and EmployeeDBS (Fig. 7). The federated schema has a class *Student* mapped to a local class/type in StudentDBS, a class *Employee* mapped to EmployeeDBS, and their superclass *Person*. A global application can change the class of a student object to *Employee*, e.g. because the student ends his study and starts work as university employee. Because the two classes are mapped to different CDBS, implicitly a migration of the personal data from StudentDBS to EmployeeDBS is required. At the global interface nothing changes, the person only changes the semantic for the university, global identity remains the same. Hence existing

global applications accessing that person do not have to be recoded.

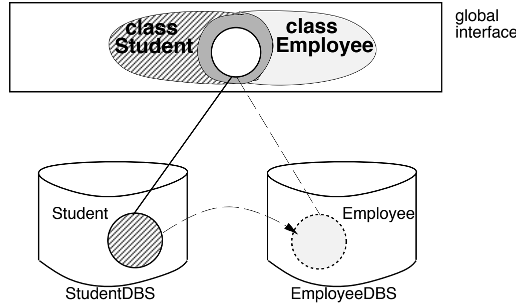

Figure 7: Example for implicit migration

At the global interface, we offer two implicit migration operations with the following specification given in the programming language independent notation of ODMG:

shift (o:Object, new_class:Class_ID)
moves an object from one class to another class.

add (o:Object, new_class:Class_ID)
allows to associate an object to an additional class.

The migration dimensions are specified for the implicit migration operations as follows: The migration dimension *operation primitive* is fixed by the selected operation: 'shift' means absolute movement and 'add' replication. The operation primitive "independent copy" cannot be realized by class-change-operations because they retain object identity. The *object kind* is implicitly known by the specified object, i.e. by its object transformation information (mapping from FDBS-global object identifier to CDBS-local identifier). The *migration degree* is fixed as globally complete, i.e. all globally visible data of the object stored in arbitrary CDBS is migrated to that/those CDBS where the target class is mapped to.

Example: To illustrate the use of the implicit migration operations, again we regard the university FDBS example. If a student, created by a local immatriculation application of StudentDBS, at some point of time terminates study and starts work as an employee in his university, a global management application may change the semantics of the person. It moves the student from class Student to Employee by the implicit migration operation `shift`.
In case a student starts work as a student assistant for the university, a global management application may replicate the student's data from class Student to Employee by the migration operation `add`. Thus, local applications of EmployeeDBS also regard the

new student assistance, but for global applications the student retains global identity and is visible as a single person at the global interface although he is stored in both StudentDBS and EmployeeDBS. Moreover, the FDBS may guarantee the consistency between the replicates.

The following code demonstrates the use of the operations for the C++ binding:

```
//absolute movement of a student into class Employee
any_student = Student [matr_no == 0815];       //object selection
any_student->shift (Employee);                 //object movement

//a student is added into class Employee but remains also in Student
another_student = Student [matr_no == 4712];
another_student->add (Employee);
```

Both implicit migration operations are transparent, i.e. the user has to specify neither source nor target CDBS. But implicit migration operations do not support all tuples of migration dimensions. They require that source and target class are different, map to different CDBS, and have a common superclass.

5.1.2 Explicit Migration Operations

By explicit migration operations all combinations of migration dimensions are supported. In general, they represent non-transparent operations, but we will also figure out some special cases in this section which work transparently.

We offer an explicit migration operation for each *operation primitive*, i.e. for absolute movement, replication, and independent copy. Each of them requires as input an object to be migrated as well as source and target schema. The object then is migrated from the CDBS corresponding to the source schema to the CDBS associated with the target schema.

Example: As an example which absolutely requires explicit migration operations and will not do with implicit migration operations we use the sales FDBS (Fig. 8) where customers are stored in two CDBS SalesDBS1 and SalesDBS2 depending in which sales district the person live. When a customer moves into another sales district its data has to be migrated into the other sales DBS. This cannot be realized by implicit migration operations because only a single global class exists for customer. Instead some explicit migration operations are required with source and target as input. Source and target are specified uniquely in terms of their associated schemata.

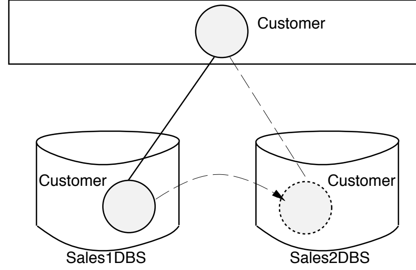

Figure 8: Example for explicit migration

The specification of the explicit migration operations is as follows:

move (o:Object, source:Schema_ID, target:Schema_ID)
moves the object *o* from the CDBS corresponding to *source* to that of *target* with the migration degree defined implicitly by the schemata.

replicate (o:Object, source:Schema_ID, target:Schema_ID)
duplicates object *o* from the CDBS corresponding to *source* to that of *target* with the migration degree defined implicitly by the schemata and manages both as replicas.

copy (o:Object, o_copy:Object, source:Schema_ID, target:Schema_ID)
duplicates object *o* from the CDBS corresponding to *source* to a new global object stored in the CDBS corresponding to *target*. The migration degree is defined implicitly by the schemata.

The migration dimensions determine these operations as follows: The *operation primitive* is uniquely mapped to one explicit migration operation. The given object, again, specifies the migration dimension *object kind*. The source schema implicitly defines the *migration degree*: according to the five-level schema architecture of [15] this means: the specification of a local or component schema corresponds to a locally complete migration, an export schema defines partial migration degree, and a federated or external schema specify globally complete migration degree. Moreover, the target schema specifies uniquely the target CDBS. For this, a local, component, or export schema can be given because each of them is uniquely associated with one CDBS. Source and target schema are specified by an identifier which is generated by the FDBS for each schema of the FDBS schema layers.

Example In order to illustrate the use of the explicit migration operations, we extend the sales FDBS by considering DBS of further enterprise areas becoming now an FDBS for Concurrent Engineering (CACE-FDBS). It contains database systems for design (De-

signDBS), for production (ProductionDBS), and two for sale (SalesDBS1, SalesDBS2). Using a C++ binding of the explicit migration operations, objects are transferred as follows among the database systems:

```
obj2=obj->copy (Sales1LocalSchema,Sales2LocalSchema);
                  //copy of 'obj' from SalesDBS1 to SalesDBS2 with
                  //locally complete migration degree

obj->replicate (ProductionExport,Sales1Export);
                  //object replication with partial migration degree

obj->move (CACEGlobalSchema,Global-DBExport);
                  //absolute movement with globally complete migration
                  //degree to Global-DB
```

Configuration of Explicit Migration Operations

In general, the migration operations are not transparent because they require location information for the source and target schema. This is not transparent for the lower three schema levels of Sheth and Larson's schema architecture. But we allow also to configure this information. Therefore, the user statically specifies some parameters which can be left out in the dynamic calls of the explicit migration operations. The following configurations are possible:

Fixed Source

When migration shall be done from a single CDBS, e.g. if this CDBS is planned to be eliminated from the FDBS, then the user can configure a fixed source. This is invoked by the following FDBS operation:

fix_migration_source (source: schema_ID)

Fixed Target

When migration is realized constantly to a single CDBS this CDBS can be configured as fixed target. For example, if data shall be transferred from some legacy DBS to a DBS of new database technology.
A target is fixed by the following FDBS operation:

fix_migration_target (target:Schema_ID)

Fixed Source and Target

If both source and target CDBS are configured, the explicit migration operations are completely transparent. They do not require any CDBS parameter.

For some special combinations of *object kind* and *migration degree* we provide also transparent access, although only a target CDBS is configured: Partial or locally complete migration to a fixed CDBS for global new, federated and reduced federated unique objects:
The source CDBS is unique and implicitly known for these object kinds and the target CDBS is configured.

Migration Calls Considering Configuration

If migration source and/or target is configured for an FDBS then this information can be left out for the dynamic calls of the migration operations. For configured information the user can specify default values, e.g. the constant FIXED. Such default parameters can also be generated by the configuration information for some language bindings and do not have to be mentioned in the operation call. In the C++ binding, for instance, default parameters can be generated for the last parameters. If the target CDBS is configured, the user can leave this last parameter for his operation calls at all. In case both source and target are fixed then no CDBS has to be given so that the calls of explicit migration operations become transparent. Also for the special transparent case of partial or locally complete migration to a fixed CDBS for global new, federated and reduced federated unique objects where the source CDBS is implicitly known this parameter can be left.

To enhance flexibility, we allow both to use the configured information and to overwrite it for specific migration calls. Therefore the configuration is not restricted to *the absolute* source/target but can determine a frequent source/target, too. If a user leaves out some last parameters, the configured information is taken. In case he specifies some source and/or target in his operation call this is valid for the migration.

Example: Assume in the CACE-FDBS SalesDBS1 and SalesDBS2 shall be combined to a single DBS (SalesDBS2) because both sales districts became a single one. Then SalesDBS1 is specified as fixed source for migration operations and SalesDBS2 as fixed target. The following code using a C++ binding shows the configuration as well as dynamic migration calls considering the configuration resp. overwriting the configuration temporarily.

```
 //Configuration of source and target
fix_migration_source (Sales1Local);
fix_migration_target (Sales2Local);

 //migration call using configured source and target
```

```
 // (generated as default parameters)
obj->move ();            //absolute movement from SalesDBS1 to SalesDBS2
                         //with locally complete migration degree

 //overwrite the source but use the configured target
obj->replicate (ProductionLocal);
                         //replication from ProductionDBS to SalesDBS2

 //overwrite the target but use the configured source
obj->copy (FIXED, ProductionLocal);
                         //copy from SalesDBS1 to ProductionDBS

 //overwrite both source and target
obj2 = obj->copy (DesignExport, ProductionExport);
                         //copy from DesignDBS to ProductionDBS
```

5.2 Advanced Operations for Migrating Multiple Objects at Once

While we restricted on the migration of single objects in the previous section, here we extend the functionality to migrate object sets and object graphs (objects with related objects) invoked by a single operation of the global FDBS interface. Thus the users can apply advanced migration operations and do not have to self-implement them. The advanced functionality is realized once in the system and many users can benefit from it. Moreover, these multi-object-operations can be mapped to most DBS more efficiently than multiple single-object-operations invoked by the user. They require, in general, less DBS calls.

Analogously to the base functionality, we specify each operation in the programming language independent manner first and then illustrate its usage for a C++ binding. Their formal semantics is defined by a mapping to the base operations. Since the specification and mapping of the advanced shift and add operations as well as the advanced move and replicate operations are analogously, we specify them in common.

5.2.1 Migration of Object Sets

In order to migrate a specified set of objects, we allow to apply the implicit and explicit migration operations also on object sets. It implies an iterative execution of object migration for all objects given in the object set (Fig. 9). This allows to first query some objects using comfortable query operations and then migrate the selected objects.

The specification of these operations as well as their mapping to the base operations is given in the following. Therefore the advanced shift and add as well as the advanced move and

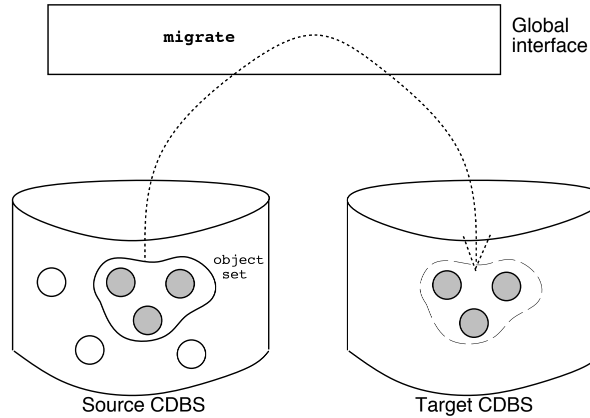

Figure 9: Migration of an object set with a single global operation

replicate operation are specified in common due to their analogous definition and mapping to the base operations.

shift/add_set (os:set<Object>, new_class:Class_ID)

$\hookrightarrow$ *FOR EACH obj* $\in$ *os:* *shift/add (obj, new_class);*

move/replicate_set (os:set<Object>, source:Schema_ID, target:Schema_ID)

$\hookrightarrow$ *FOR EACH obj* $\in$ *os:* *move/replicate (obj, source, target);*

copy_set (os:set<Object>, os_copy:set<Object>, source:Schema_ID, target:Schema_ID)

$\hookrightarrow$ *FOR EACH obj* $\in$ *os:*

copy (obj, new_obj, source, target); insert(new_obj, os_copy);

As mentioned in Section 5.1, source and/or target CDBS can be configured for the advanced explicit migration operations move, replicate, and copy.

Example The following example illustrates the use of the operations for object-set migration in a C++ binding. It demonstrates how an article assortment can be dynamically changed between the two sales districts. Article data that was experimentally introduced in the test district 1 and successfully adopted after a year can be easily duplicated into SalesDBS2.

```
 //selecting article objects with creation-date 1994
obj_set = Article {creation-date == 1994};

 //locally complete replication of all in 1994 newly and successfully
 //introduced articles
obj_set->move_set (Sales1LocalSchema, Sales2LocalSchema);
```

As in the object migration base model, we do not transfer relationships of the objects by default. But this is extended in the following section.

5.2.2 Object Set Migration Considering Relationships

To migrate a specified set of objects together with their relationships among another, we extend the previous operations with an additional parameter. So not only isolated objects but whole data structures can be transferred among the CDBS (Fig. 10).

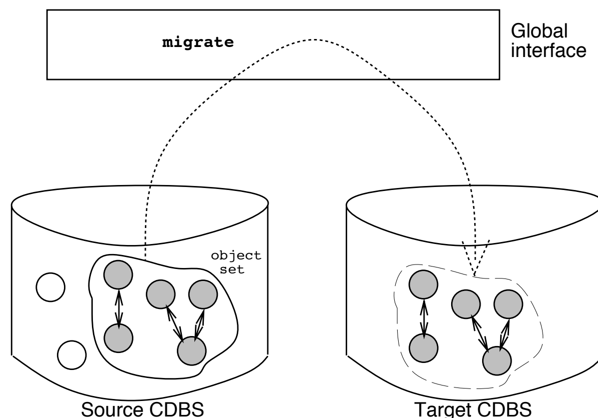

Figure 10: Migration of an object set including the relationships among another

Relationships to be considered during the migration can be restricted by the user. He can limit (1) the relationship kind (2) can specify a specific relationship. A restriction on some relationship kinds is ingenious if the canonical data model of the FDBS offers multiple relationship kinds. For example, COMIC [7] offers general relationships (called relate) and part-of relationships (contain) while PCTE [5] distinguishes five predefined relationship kinds and allows users to specify some further kinds. Then only relationships of the user-specified kind are considered. It can also be limited on specific relationships by mentioning its attribute name.

The migration operation transfers iteratively each object of the object set together with relationships to other objects of the set with the given relationship (kinds).

In the following we present the specification of the operations. For migrating the relationships we introduce an internal operation `set_rel` which transfers an attribute representing a relationship into the target: set_rel (o1:Object, o2:Object, r:Relationship, target:CDBS_ID); establishes a relationship *r* in CDBS *target* between the objects *o1* and *o2*.

shift/add_set_with_rel (os:set<Object>, new_class:Class_ID, rr:RelRestriction)

↪ *FOR EACH o ∈ os:*

shift/add (o, new_class);

FOR EACH relationship r of o fulfilling the restrictions rr

FOR EACH obj ∈ r: IF obj ∈ os: set_rel (o, obj, r, CDBS_of_new_class);

move/replicate_set_with_rel (os:set<Object>, source:Schema_ID, target:Schema_ID, rr:RelRestriction)

↪ *FOR EACH o ∈ os:*

move/replicate (o, source, target);

FOR EACH relationship r of o fulfilling the restrictions rr

FOR EACH obj ∈ r: IF obj ∈ os: set_rel (o, obj, r, target);

copy_set_with_rel (os:set<Object>, os_copy:set<Object>, source:Schema_ID, target:Schema_ID, rr:RelRestriction)

↪ *FOR EACH obj ∈ os:*

copy (obj, new_obj, source, target);

insert (new_obj, os_copy);

FOR EACH relationship r of o fulfilling the restrictions rr

FOR EACH obj ∈ r: IF obj ∈ os: set_rel (o, obj, r, target);

Example: The following example migrates the two married customers Adam and Eve from SalesDBS1 to SalesDBS2 when they move town. Thereby it transfers their relationship *married_with* as well. The C++ binding uses constants to specify relationship kinds: INCL_RELATE for relate-relationships, INCL_CONTAIN for contain-relationships, and INCL_RELATE_CONTAIN for both. It can be easily extended for further relationship kinds by offering additional constants.

```
//selecting both customer objects (with no = 1011 and no = 1012)
obj_set = Customer {no == 1011 || no == 1012};

//locally complete absolute movement of their data incl. the relationships
obj_set->move_set (Sales1LocalSchema, Sales2LocalSchema, INCL_RELATE);
```

An important restriction of this kind of migration is that object sets (in general) only contain objects of a single class. Therefore only relationships are considered among objects of *one* class. In the next section we introduce functions which migrate arbitrary relationships.

5.2.3 Recursive Migration of Object Graphs

The migration of object graphs can be used to migrate a data structure composed of arbitrary related objects at once (Fig. 11). In contrast to the operations of the previous section which migrate object sets together with their relationships, here only a single object has to be specified and recursively all related objects, their related objects and so on can be migrated. Furthermore relationships between objects of different classes can be transferred.

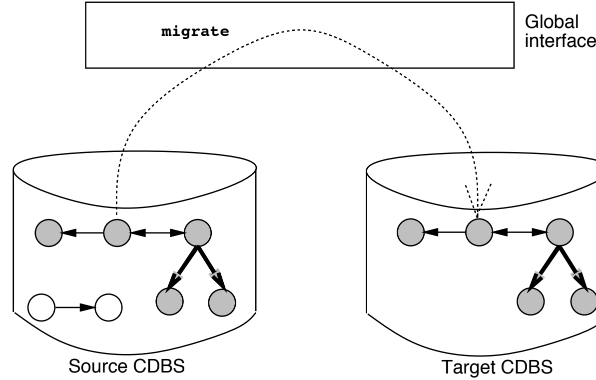

Figure 11: Migration of an object graph with a single global operation

During object migration, the specified object and recursively all objects related to it, their related objects, etc. are transferred to the target together with their relationships. The relationships to be considered can also be limited. Since objects of arbitrary class can be in an object graph, the user can limit not only (1) the relationship kinds and (2) the the relationship, but also (3) types/classes of the related objects. By an optional parameter, the recursion level can be fixed so that related objects are only migrated up to this depth. Migration degree and operation primitive are defined only once and are valid for all objects of the object graph. For graph traversal and cycle recognition in the object graph, general graph algorithms [6] can be taken.

The operations for migrating object graphs are specified as follows:

shift/add_graph (o:Object, new_class:Class_ID, rr:RelRestriction, deep:Integer)

$\hookrightarrow$ *shift/add (o, new_class);*

IF deep > 0

FOR EACH relationship r of o fulfilling the restrictions rr

FOR EACH obj $\in$ r $\wedge$ class of obj = source_class:

shift/add_graph (obj, new_class, rr, deep-1);

set_rel (o, obj, r, CDBS_of_new_class);

move/replicate_graph (o:Object, source:Schema_ID, target:Schema_ID, rr:RelRestriction, deep:Integer)

↪ *move/replicate (o, source, target);*

IF deep > 0

FOR EACH relationship r of o fulfilling the restrictions rr

FOR EACH obj ∈ r:

move/replicate_graph (obj, source, target, rr, deep-1);

set_rel (o, obj, r , target);

copy_graph (o:Object, o_copy:Object, source:Schema_ID, target:Schema_ID, rr:RelRestriction, deep:Integer)

↪ *copy (o, o_copy, source, target);*

IF deep > 0

FOR EACH relationship r of o fulfilling the restrictions rr

FOR EACH obj ∈ r:

copy_graph (obj, tmp_obj, source, target, rr, deep-1);

set_rel (o, obj, r , target);

Note that the implicit operations can only consider objects of a single class since they define a fix source class. But in contrast to the object set operations of the previous section, the user migrates only related objects by specifying a single starting object.

Example: Assume for our CACE FDBS that customers are migrated together with all articles they bought. That means when a customer moves town into another sales district both his personal data and his related articles are migrated between the sales database systems. Then we only have to specify the customer object and invoke recursive migration of all related objects of type Article.

```
//locally complete absolute movement of customer c with all his articles
c->move_graph (Sales1LocalSchema, Sales2LocalSchema, Article);
```

Moreover, a design object of DesignDBS composed of multiple parts which themselves contain parts etc. can be easily transferred to the ProductionDBS using a single operation. Such complex objects frequently occur in engineering areas and hence object graph migration gives good support in Concurrent Engineering where these data are exchanged among DBS.

```
//partial independent copy of a complex design object with ALL its components
d->copy_graph (DesignExport, ProductionExport, INCL_CONTAIN);
```

6 Conclusion

Based on a requirement analysis, we presented an architecture and concepts for migrating objects among distributed, heterogeneous, and autonomous DBS. New functionality is incorporated into the chosen architecture of federated database systems. It allows to migrate data among DBS in various granularities dynamically. We introduced migration operations and presented a C++ binding which seamlessly extends the ODMG standard. The operations allow to (a) implicitly migrate objects by changing their class or (b) explicitly migrate objects from one DBS to another. While the former are transparent, but only support few migration cases, the latter support a wide area of object migration requests, but do not allow a transparent migration in most cases. Together, they offer a flexible object migration functionality. While base operations allow to move, replicate, or copy single objects across DBS, advanced operations enable the migration of whole object sets or object graphs with a single operation.

We already validated the need for object migration by some industrial project cooperations where in particular the coupling and stepwise reduction of the multiple existing DBS was requested [12]. Currently, we are developing an FDBS which couples primarily relational database systems (Entire, Oracle), object-oriented database systems (Siframe-OMS), and file systems (Unix). It will be extended by further DBS adapters, e.g. considering specific data(base) systems of our industrial project partners. In order to fulfill the industrial requirements, the FDBS will also incorporate object migration functionality.

References

[1] CATELL, R.G.G. *The object database standard: ODMG'93.* Morgan Kaufmann Publisher, 1994.

[2] CHAPPELL, C., STEVENSON, C. *Concurrent Engineering: The Market Opportunity.* OVUM, 1992.

[3] DALE, R. Database migration: keeping a steady course. *Database Programming and Design 3(4)*, pages 30–38, 1990.

[4] DOLLIMORE, J., NASCIMENTO, C., XU, W. Fine grained object migration. In *Proc. Int'l Workshop on Distributed Object Management (Edmonton, Canada)*, pages 181–186, 1992.

[5] European Computer Manufactures Association (ECMA). Standard ecma-149: Portable common tool environment: Abstract specification, version 1.0, 1990.

[6] EVEN, S. *Graph Algorithms.* Computer Science Press, 1979.

[7] KACHEL, G., RADEKE, E., HEIJENGA, W. COMIC - A step toward future data models. Cadlab-Report 3/92, 1992.

[8] LITWIN, W. O*SQL: a language for multidatabase interoperability. In *Proc. IFIP-DS5 Semantics of Interoperable Database Systems (Lorne, Australia)*, pages 114–133, 1992.

[9] MARCA, D., BOCK, G. *Groupware: Software for Computer-Supported Cooperative Work*. IEEE Computer Society Press, 1992.

[10] MEIER, A., DIPPOLD, R. Migration and co-existence of heterogeneous databases; practical solutions for changing into the relational database technology (in german). *Informatik-Spektrum 15(3)*, pages 157–166, 1992.

[11] ÖSZU, M.T., VALDURIEZ, P. *Principles of Distributed Database Systems*. Prentice Hall Publishing, 1991.

[12] RADEKE, E., SCHOLL, M.H. Federation and stepwise reduction of database systems. In *Proc. 1st Int'l Conf. on Applications of Databases (Vadstena, Sweden)*, 1994.

[13] RADEKE, E., SCHOLL, M.H. Framework for object migration in federated database systems. In *Proc. Int'l Conf. on Parallel and Distributed Database Systems (Austin, USA)*, 1994.

[14] SALTOR F., CASTELLANOS M., GARCIA-SOLACO M. Suitability of data models as canonical models for federated databases. *SIGMOD RECORD 20(4)*, pages 44–48, 1991.

[15] SHETH, A., LARSON, J. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys 22(3)*, pages 183–236, 1990.

[16] SRINIDHI, H.N. Management of redundant data in interoperable environments. In *Proc. 2nd Int'l Workshop on Interoperability in Multidatabase Systems (Vienna, Austria)*, pages 236–239, 1993.

[17] TRESCH, M. Dynamic evolution in object databases (in german). PhD thesis, University of Ulm, Germany , Feb. 1994.

Contents

1 Introduction **1**

2 Requirements **3**

3 Architecture for Coupling Multiple DBS **5**

4 Concept for Object Migration **7**

4.1 Base Model 7

4.2 Migration Dimensions 8

5 Interface Functions for Object Migration **10**

5.1 Base Operations 10

5.1.1 Implicit Migration Operations 11

5.1.2 Explicit Migration Operations 13

5.2 Advanced Operations for Migrating Multiple Objects at Once 17

5.2.1 Migration of Object Sets 17

5.2.2 Object Set Migration Considering Relationships 19

5.2.3 Recursive Migration of Object Graphs 21

6 Conclusion **23**