Proc. Int'l Conf. on Parallel and Distributed Information Systems, PDIS '94, Austin, Texas, September 1994, pp. 195-202

A Classification of Multi-Database Languages

Markus Tresch *

IBM Almaden Research Center
650 Harry Road (K55/801)
San Jose, CA 95120, USA
tresch@almaden.ibm.com

Marc H. Scholl

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

Abstract

This paper defines a formal classification of multi-database languages into five levels of database integration with increasing degree of global control and decreasing degree of local autonomy. First, the fundamental interoperability mechanisms are identified for each of these levels. Their consequences on local autonomy as well as implementation draw-backs are discussed. Second, various multi-database languages are classified into these categories. In addition to our own language COOL, other proposals are analyzed, including SQL*Net, Multibase, Superviews, VODAK, Pegasus, and O*SQL.*

1 Introduction

Novel data-intensive information systems are characterized by cooperating (autonomous and heterogeneous) database systems and therefore increasingly require openness of database management systems for a cooperation with other services, be they data managers or other service providers. Hence, the area of interoperable multi-database systems (MDBSs) has attracted a lot of recent attention. Practical solutions typically consist of several DBMSs that are loosely integrated via data extraction – data conversion – data upload cycles. This requires extensive and error-prone application programming, yet guarantees only a minimum of data consistency. The challenge for future cooperative systems is to provide flexible and scalable mechanisms to support system-controlled interaction among different data management systems.

A wide variety of problems need to be solved in order to make MDBSs work: data model transformation, schema integration, MDBS query languages and optimization, MDBS transaction management, and data and application migration. This paper concentrates on MDBS language aspects for integration of data (schema and instance level) from different component databases. We concentrate on homogeneous multi-databases, separating the issue of data model transformation, and assuming that all schemas have been transformed into a uniform data model.

Multi-database systems are built up of several component database systems (CDBS) managing local component databases $DB_1, DB_2, \ldots$. An MDBS is supposed to provide global operations (queries and updates) on objects stored in different CDBSs consistently, while CDBSs should continue autonomous processing of local operations. The structure of each DB_i is given by a component schema and the structure of the multi-database is given by the global (federated) schema [15]. A federation dictionary (FD) contains (meta) information about the distribution and integration of schemas.

The contribution of this paper is a classification of MDBS languages into five integration levels ranging from loosely coupled databases, through three levels of federated DBMSs, to fully integrated, distributed DBMSs. These levels are separated by the way how objects in CDBSs that represent "the same" real world entity can be identified and tied together in the MDBS. They are also a measure for the degree of autonomy that component systems have to give up as the price for tighter cooperation.

In Section 2, we review the basic interoperability mechanisms. In Section 3, we define the classification into five levels of MDBS integration. The database language COOL* [12, 14] is used as a platform, where all constructs are given sound and formal semantics. However, the classification is data model/language independent. In Section 4, we classify and compare various current related MDBS languages accordingly

*Work done while at Faculty of Computer Science, University of Ulm, Germany.

An extended version of this paper is available as Technical Report UIB 94-07 from the same university.

by mapping some of the proposed constructs to their COOL* counterparts. Section 5 gives an outlook to future work.

2 Basic Interoperability Mechanism

In an MDBS, *entity objects* (objects of the real world) are to be distinguished from *proxy objects* (their approximation in one database) [5]. One particular entity object can be represented by multiple proxy objects in different component databases.

Let o_i and o_j be two proxy objects from different CDBSs, representing the same (real world) entity object. Due to local autonomy, the OID domains of different CDBSs are pairwise disjoint, such that no two proxy objects from different CDBSs can be the same (identical). Object integration requires mechanisms to integrate proxy objects o_i, o_j, if they represent the same entity object, such that the MDBS treats them as one single object in global queries and updates. OIDs are not adequate to globally identify objects, since they are internal representations within *one* CDBS. Entity objects can only be globally identified by characterizing values ("value identifiability" – a generalization of identification keys from relational systems).

One approach would be to link local proxies via translation tables maintained in the federation dictionary. We formalize this by functions with special semantics ("the same"), defining a global MDBS integrity constraint, which is known to the global query and update operations. Such partial, injective, single-valued functions are called $same_{i,j}$ [14]:

define function $same_{i,j}$: $\mathbf{object}_i \rightarrow \mathbf{object}_j$

same-functions are inter-database functions with domain $\mathbf{object}_i$ in database DB_i and range $\mathbf{object}_j$ in database DB_j, and returning for a given DB_i-proxy object the "same" DB_j-proxy object (if any). Having *same*-functions, global object identity can now be defined:

Definition 1. (Global object identity) The global object identity of multi-database objects o_1, o_2 is defined as $=_{gl}$: $\mathbf{object} \times \mathbf{object} \rightarrow \mathbf{bool}$ where

$$
\begin{aligned}
& o_1 =_{gl} o_2 \iff \\
& (\exists i : object_i(o_1) \wedge object_i(o_2) \wedge o_1 =_i o_2) \\
& \vee (\exists same_{i,j} : \mathbf{object}_i \rightarrow \mathbf{object}_j : \\
& \quad object_i(o_1) \wedge object_j(o_2) \wedge o_2 =_j same_{i,j}(o_1)).
\end{aligned}
$$

From now on, two objects are the same, if they stem from the same CDBS and are identical in it, *or* if they have been defined (by the user/DBA) to be the same using *same*-functions.

The goal of schema integration is to find out what the common (structural) parts in the local schemas are and to define correspondences among them. Our matter of concern is not to find another schema integration methodology for resolving structural and semantic conflicts. Rather we are interested in identifying (and later classifying) the necessary basic abstraction mechanisms for elementary database integration. It is quite common to most object models, that databases contain a meta database with objects representing every schema element of the application schema. In COOL* for example, objects of the meta database represent persistent variables, functions, types, classes, and views. As for "ordinary" object integration, we use *same*-functions for schema integration, but now applied on schema objects of the meta database (see Section 3.2 for an example).

same-functions are to be understood as the basic, data model independent abstraction mechanism for object *and* schema integration, used within this paper. Schema integration methodologies/strategies [2], can be implemented using *same*-functions as base technology. Instead of *same*-functions, one may alternatively think of global query expressions or relations (tables) mapping between objects from different CDBSs. Concrete implementation alternatives for such *same*-functions for different data models are discussed in the next section.

3 Five Levels of MDBS Integration

We now formally define a classification of MDBS languages into five levels of database integration with increasing degrees of global control and decreasing degrees of local autonomy. This classification refines [15] that distinguishs between losely and tightly coupled database systems only.

Integration level 0 represents non-integrated MDBSs. This is the weakest form of database coupling, where component systems are fully autonomous. Neither objects, nor schemas are integrated. Level 0 is a kind of ad hoc data "integration". Global transaction management allows to process objects from different CDBSs within one global transaction: each individual query/update statement works on only one CDBS.

Level IV represents fully integrated (maybe physically distributed) databases. Participating component systems completely lost their local autonomy. Though objects might be physically distributed, these systems have one single logical database schema. Distribution

is therefore logically transparent.

In between these two extremes, levels I, II, and III describe federated database systems (FDBS). They are the most challenging architectures, because on the one hand, their objects and schemas are subject to some global control, and on the other hand, participating CDBSs have retained some local autonomy. In the sequel, we focus on these levels, i.e., on federated object database systems. Though we use the COOL* multi-database language for illustration purposes, the conceptual ideas can be transferred to other languages.

3.1 Level I: Composition

Integration level I is called *schema composition*. It is the elementary process to combine multiple CDBSs DB_i into one composite schema GDB, and is therefore the foundation for establishing a federated database system. Schema composition places only minimal requirements on the degree of integration between participating systems. It just imports the names of all schema elements from CDBSs and makes them globally available. The type and class systems of the local databases are combinded, without establishing connections between composite systems. As an anchor, basic data types of component systems are assumed to be identical.[1] This ensures that at least values of elementary data types can be compared between component systems. Local object type and class hierarchies of the CDBSs are then put together – in a so far trivial way – by defining a new global top type (the common supertype of all local root types) and a new global top class (the common superclass of all local root classes).

In COOL* for example, names of persistent variables, functions, types, classes, and views are made globally available.[2]

EXAMPLE 1: Consider a university environment, where data about students are stored in a library database *LibDB*, a student database *StudDB*, and an employee database *EmplDB*. The following COOL* statements compose these three CDBSs into one global schema *UnivDB*:

define database *UnivDB*
  import *LibDB, StudDB, EmplDB*
end.

[1] The internal representations of integer, string, boolean, ... are identical, or alternatively, an equivalence preserving transformation exists.

[2] In the sequel, we use the naming convention that schema components are suffixed by "@" and the name of the local schema. For example, class *Books* in *LibDB* has as globally unique name "*Books@LibDB*".

A global hierarchy of object types is created with a new top type **object@***GDB*, of wich all top types of the CDBSs (**object@***DB_i*) are made direct subtypes. COOL* has a type lattice, therefore, a new bottom type **bottom@***GDB* is made common subtype of all local bottom types. Similar, a global class hierarchy is established, with the top element **Objects@***GDB* as common superclass to all local top classes **Objects@***DB_i*. For other data models, the effect will be similar. ◇

Schema composition creates a global meta schema as well. This is the meta schema of GDB and has the structure of the union of the meta schemas of each DB_i. Though the concrete meta schema depends on the used data model, the idea of a composite meta schema remains unchanged for any other approach.

Once two (or more) schemas are composite, queries can be formulated that involve multiple CDBSs. Recall composition *UnivDB* from Example 1. Since composition made basic data types and name spaces globally available, comparing names of customers (from *LibDB*) with names of students (from *StudDB*) is legal. Hence, the following valid nested query selects those customers being students as well:

$$\textbf{select}[\emptyset \neq \textbf{select}[name(c) = name(s)](s : Students)](c : Customers)$$

Unfortunately, the possibilities of inter-database queries are very limited up to now. E.g., the following more elegant solution of the same query is not allowed:

$$\textbf{select}[c \in Students](c : Customers)$$

Since objects of class *Students* are of type "student" and the type of c is "customer" and the two types "student" and "customer" are not (yet) related, the selection predicate $c \in Students$ would be rejected by the MDBS type checker.

Schema composition (Level I) is not yet "real database integration". No *same*-functions exist and no two objects can be the same (identical), unless they originate from the same DB_i. Furthermore, type and class systems are integrated only at the very top level.

3.2 Level II: Virtual Integration

Level II is called *virtual integration* and forms the next increased degree of database cooperation. *Views* (derived/computed classes, external schemas [13]) can now be used to build a uniform, virtual interface over multiple databases. Views spanning CDBSs define persistent links between component systems and/or combine classes from different systems.

A federation dictionary (FD) is now required to store global information. However, since cooperation is restricted to virtual integration, the federation dictionary contains meta data, that is, *instance*-**in***dependent* information only, e.g., definitions of multi-database views (i.e. queries). Instance-dependent information, like e.g. object identifiers (OIDs) or object values, must not yet be stored in the federation dictionary, forming the main restriction of integration level II and preventing from tight cooperation. In COOL* for example, the **extend** query operator defines new functions, derived by a query expression. This possibility can be used to define a view, connecting two CDBSs. E.g. the following view stores together with each employee (of *EmplDB*) the books (of *LibDB*), that she/he lent, defining new function *lbooks*:

define view *Employees* **as**
extend[*lbooks* :=**select**[*name*(*e*) = *name*(*lent*(*b*))]
(*b* : *Books*)](*e* : *Employees*)

Inter-database link *lbooks* from *EmplDB* to *LibDB* is made persistent, and the definition of the link (the query) is stored in the global FD.

At integration level II, proxy objects from different CDBSs representing the same real world entity can be integrated. For any two component databases DB_i and DB_j, a query expression is given that determines for a DB_i-object the corresponding DB_j-object (if any). In COOL* for example, derived *same*-functions (cf. Definition 1) from DB_i to DB_j are possible at level II, by using **extend** views, similar to the above *lbooks* example.

EXAMPLE 2: To integrate objects of class *Students@StudDB* with objects of class *Employees@EmplDB*, if they have identical names, a *same*-function is defined by the following view:[3]

define view *Students* **as**
extend[$same_{\text{StudDB,EmplDB}}$:= **pick**(
select[*name*(*e*) = *name*(*s*)](*e* : *Employees*))]
(*s* : *Students*) ◇

We now focus on schema integration, that is, defining correspondences between schemas of different CDBSs. We make use of the fact that every schema element is represented by an object in the meta database (cf. Section 2). In COOL*, e.g. functions are unified by defining a *same*-function from meta type *function*@DB_i to meta type *function*@DB_j. After that, the multi-database language treats these two integrated functions as if they where one single global attribute.[4]

EXAMPLE 3: To unify functions *name@StudDB* and *name@EmplDB*, the following *same*-function is defined on the composite meta schema of *UnivDB*:

define view *Functions@StudDB*
as extend[$same_{\text{StudDB,EmplDB}}$:= **pick**(
select[*fname*(*f*) = *name* ∧ *fname*(*g*) = *name*]
(*g* : *Functions@EmplDB*))]
(*f* : *Functions@StudDB*)

fname(f) is a meta function, returning the name of a function, represented by meta object *f*. ◇

Now, all prerequisites for virtual CDBS integration are defined:

EXAMPLE 4: Local schemas are composite by importing *LibDB*, *StudDB*, and *EmplDB*. Then, class *Students* is extended with a *same*-function, and meta class *Functions@StudDB* is extended to integrate *name@StudDB* and *name@EmplDB* properties. Finally, view *Persons* defines a union over the extended classes *Students@StudDB* and *Employees@EmplDB*, spanning multiple CDBSs.

define schema *UnivDB* **as**
import *LibDB, StudDB, EmplDB;*
define view *Students@StudDB* **as**
extend ...; // *see Example 2*
define view *Functions@StudDB* **as**
extend ...; // *see Example 3*
define view *Persons* **as** *Students@StudDB*
union *Employees@EmplDB;*
end.

The extent of view *Persons* is the union of the base class objects. Customer objects and student objects having equal names are defined through the *same*-function to represent the same real world object, and will therefore appear only once in the union view. The type of a union view is given by the intersection of the base class functions. Since types of *Students* and *Employees* are disjoint, except for integrated functions

[3] The **pick** operator does a set collapse, returning the object from a singleton. It returns undefined if the set is empty, and raises a run-time excpetion if the set contains more than one object.

[4] Notice that, 1. not only the unification of functions, but of any meta object, representing variables, types, classes, or views, is possible; 2. the signatures of schema elements to be unified must be compatible, that is, they must have same names and structures; 3. unifying schema elements my cause value conflicts, that is, two attributes e.g. may be unified though they have different local values. The discussion of these issues is out of the scope of this paper; we refer to [14].

name@StudDB and *name@EmplDB*, there is one single function, *name*, applicable to these objects. ◇

3.3 Level III: Real Integration

Level III is called *real integration* and forms the next increased degree of database cooperation without the need of completely giving up local CDBS autonomy. The use of the FD is enhanced to store *instance-dependent* information (e.g. object values, OIDs). This does not say that all objects from CDBSs are copied into the FD. As a consequence, CDBSs are loosing further autonomy, since they must inform the MDBS upon local updates (e.g. object deletion), in order to insure that copies of values/OIDs are deleted in the FD as well (cf. consistency of multiple representations).

In general, schema integration at integration level III is not any more limited to views. In COOL* for example, stored inter-database functions are now allowed.

EXAMPLE 5: Consider again MDBS *UnivDB*. An inter-database function *favourite_book* from *StudDB* to *LibDB* can be defined, which is not derived by a query, but stored explicitely and needs therefore the enhanced FD to store its values:

define function *favourite_book* :
student@StudDB → *book@LibDB*

A special case of that are stored *same*-functions. ◇

Notice, that this gives really advanced possibilities, since we do not need to know a query to retrieve *same* objects from other CDBSs. This was not possible at level II.

Additional global schema augmentation possibilities of level III are: (i) object types, that are subtypes of different CDBSs and therefore contain functions from multiple CDBSs, (ii) classes that are subclasses from different CDBSs, and (iii) variables that can hold objects from multiple CDBSs as values. These global schema augmentations are only visible to the MDBS and are not known to CDBSs. Not only MDBS queries respecting the global object identity are available, but general updates, spanning multiple CDBSs are possible as well.

In COOL* for example, there is a generic update operation **gain**[t](o), adding object type t to object o [12]. As long as type t and object o stem from the same database, the **gain** operation works as in one centralized database. However, if o and t are from different databases, the semantics becomes unclear, since an object can usually not get a type from an other database. One realization of this **gain** operation for MDBSs might work such that a *same* object o' of o is created in the database where type t is defined and a local gain operation is performed, making o' an instance of t.

This realization maps the multi-database **gain** operation to a sequence of operations, that can be executed within one single CDBS. Since an object o' of DB_j is assigned to be the *same* object as o of DB_i, stored *same*-functions are needed, that are only possible at level III or higher.

It is important to understand, that the above global **gain** operation cannot be implemented, using derived (Level II) *same*-functions. To be even more general, although the above realization of **gain** is just one possible way of how to do it, we argue, that there is no other realization of such an operation in any other language, that can be done, using virtual (Level II) mechanisms exclusively.

3.4 Summary

Table 1 gives a comparison of the main characteristics of integration levels 0 to IV. Notice, that mixed levels of integration may coexist, where e.g. some objects/classes are virtually integrated, whereas others are really integrated. A language is called "of level n", if it contains at least one mechanism of level n and none of level n+1.

4 Classification of Interoperability Mechanisms

We now concentrate on the use of the above classification in order to compare related multi-database approaches. For this purpose, we selected a couple of (well known) multi-database languages (SQL*Net, Multibase, Superviews, VODAK, Pegasus, and O*SQL) and identified their main static (schema) and dynamic (operational/language) interoperability mechanisms. According to that, these languages are classified into level I, II, or III.

4.1 *connect-to*-Statement of Oracle SQL*Net and INGRES/Star

With special software packages, like e.g. Oracle SQL*Net [11] or INGRES/Star [4], many relational database system products allow for the definition of connections between multiple database systems, making distribution of data more transparent.

After establishing connections to multiple databases, by a `CONNECT TO <database>` statement for example, queries can join tables from different component databases. However, the join predicate

Table 1: Five Levels of Multi-Database Integration

	Multi-DBS	**Federated DBS**			**Distr. DBS**
	Level 0	*Level I*	*Level II*	*Level III*	*Level IV*
logical schema integration	schemas not integrated	schemas composite	schemas virtually integrated	schemas really integrated	schemas completely integrated
proxy-objekt unification	fully disjoint sets of objects		derived *same*-functions	stored *same*-functions	one set of objects only
global query and update operations	global transactions	restricted global operations	queries using global object identity	updates using global object identity	as in central DBS
federation dictionary (FD)	not necessary	used for instance-independent information only		used for instance-dependent information too	not available

is only allowed to compare between basic data types, which follows directly from that only these basic data types are integrated over CDBSs. Therefore, *connect-to*-statements are equivalent to schema composition and hence to integration level I.

4.2 Multi-Database Views in Multibase and Superviews

Multibase [6] and Superviews [9] provide a uniform retrieval interfaces (no updates) on top of multiple database systems, using global views. Thus, both approaches correspond to integration level II.

Multibase integrates pre-existing databases via view mappings, building global entity types out of local attributes. Queries must be given, describing how global entities and their values are derived from local entities. One may, for example, define that two entities with equal key value globally appear only once (cf. proxy object integration).

Superviews describes virtual integration using a set of integration operations. It does not provide a general view mechanism based on a query language. Thus, together with each integration operation, a transformation of global queries into queries of local classes is defined.

Some integration operations are restricted in use. E.g. the operation **add**, augmenting the global schema with a new attribute. While this is a level III mechanism in general, (cf. Section 3.3), Superviews allows only for adding attributes with constant values, which is, in contrast, possible at integration level II, because it can be realized storing instance-independent information in the FD only.

4.3 Generalizations of VODAK

VODAK [10] integrates databases via generalizations over classes of multiple CDBSs. To support different semantic relationships between proxy objects and attributes, multiple kinds of generalizations are identified and enumerated. All of which are equivalent to virtual integration and therefore to cooperaion level II. Consider for example the following VODAK role-generalization:

```
class TAXPAYING-EMPL
   role-generalization-of:
      UNIV-EMPL, COMP-EMPL
   object correspondence rules:
      UNIV-EMPL.SS# = COMP.EMPL.ID#
   attributes: BORNON
      identical: UNIV-EMPL->BIRTHDATE
                 COMP-EMPL->BIRTHDATE
end TAXPAYING-EMPL
```

To show, that this generalization is a level II mechanism, we sketch its reduction to (derived) *same*-functions and a **union** view: First, a derived *same*-function from *CompEmpl c* to *UnivEmpl u* is defined, unifying objects with $ss\#(u) = id\#(c)$, (cf. Example 2). Second, functions $birthdate@DB1$ and $birthdate@DB2$ are unified using a *same*-function on the meta database (cf. Example 3). Finally, classes are integrated by a **union** view *TaxpayingEmpl*, (cf. Example 4), which is now equivalent to the above VODAK generalization.

In COOL*, we require that functions to be unified have identical names, wich is not necessary in VODAK. However, renaming parts of a schema can be done at level II (see Section 4.6 below).

4.4 *unifier*- / *image*-Functions in Pegasus

Pegasus [1] internally describes type and object integration using two system functions: $unifier(t)$ defines for each CDBS type t exactly one unified type of the global schema. $image(o)$ returns for each lo-

cal object o at most one unified global object.[5] The default assumption is $unifier(t) = t$ and $image(o) = o$ and can be overriden by defining global inter-database types. The following statement, for example, integrates three local types $NStud, EStud, WStud$ from different CDBSs into one global type *Student* (HOSQL syntax [1]):

```
CREATE TYPE Student
ADD UNDERLYING TYPES NStud, WStud, EStud
UNDER Student
(WStud.Image(x) AS SELECT s
  FOREACH Student s WHERE ssn(s) = ssn(x))
(EStud.Image AS STORED)
```

Corresponding *unifier* and *image* functions are created automatically by the system. For each underlying type, *unifier* is set to *Student*, e.g. $unifier(NStud) = Student$. For *NStud* objects, *image* is the default mapping $image(o) = o$. For *WStud*, it is a derived mapping, given by a HOSQL SELECT expression.

So far, these are level II mechanisms. However, for *EStud* the *image* function is a stored function, that is, $image(o)$ is undefined until an instance of *Student* is assigned explicitely. As we know, this needs a federation dictionary storing instance-dependent information and requires therefore integration level III.

Notify, that Pegasus is mainly a level II system (derived *unifier* and *image* functions), except of some very few mechanisms, like e.g. stored *image* functions, that are of level III.

4.5 *merge*-Operation in O*SQL

O*SQL [7] is a comprehensive multi-database language, providing e.g. functions and types spanning multiple databases. They can be derived from an O*SQL query expression, resulting therefore in a level II integration. Whether stored inter-database functions and types augmenting the global schema are allowed as well is unclear from the available paper. However, such possibilities are language extensions, resulting in integration level III and further loss of local CDBS autonomy.

In O*SQL, proxy objects are integrated by a *merge*-operation. E.g., the expression `merge :o1, :o2` unifies objects $o1$ and $o2$, and

```
select merge(ss#(e) e s)
for each Empl e Stud s where ss#(e) = ss#(s)
```

describes a kind of object-unifying join, integrating employees and students with equal *ss#*. In both cases, a global table of "same" objects must be allocated in the FD. Notice, that the semantics of the select operation is not that of a derived *same*-function, since the result is stored (materialized). The O*SQL *merge*-operation is therefore a level III mechanism.

[5] The following constraint always applies o *instance_of* $t \Rightarrow$ $image(o)$ *instance_of* $unifier(t)$.

4.6 Discussion – Information Capacity

We presented interoperability mechanisms of some selected multi-database languages, as summarized in Table 2. Of course, the enumeration of languages was not complete. We considered those systems, focusing in object and schema issues. Other approaches, discussing for example mainly MDBS transactions, architectures, or data model heterogeneity are not taken into account yet.

Table 2: Selected Interoperability Mechanisms of Integration Levels I – III

Level	*Concepts and Mechanisms*
I	COOL* schema composition (Sect. 3.1)
	Oracle SQL*Net [11], INGRES/Star [4] *connect to*-statement
II	COOL* (Sect. 3.2), Superviews [9], Multibase [6] MDBS-views
	COOL* derived *same*-functions (Sect. 3.2)
	VODAK generalizations [10]
	Pegasus *unifier*-functions and derived *image*-functions [1]
III	COOL* stored *same*-functions (Sect. 3.3)
	COOL* update operations [12]
	Pegasus stored *image*-functions [1]
	O*SQL *merge*-operation [7]

One may ask, whether there isn't a general notion on how to find out, what kind of mechanism is of what particular integration level. It shows, that the key to answer this question is *change of information capacity* [3, 8].

Definition 2. (Information Capacity) The information capacity $\mathcal{DB}_S$ is the set of all potential states a database can take with given schema S.

The capacity of a database is therefore given by its schema. Hence, changing the schema of a database may directly have an impact on its capacity. We say, a schema change is capacity *preserving* (CP) / *augmenting* (CA), if it does preserve / augment the information capacity of the database.

For multi-databases, the *global information capacity* is given by the composite (global) schema, reached by schema composition at integration level I. Any further database (schema or object) integration mechanism may now change this global information capacity.

An interoperability mechanism is of level II, iff it preserves the information capacity of the global (composite) database. Any kind of adding derived (virtual) information, like MDBS views e.g. in COOL*, Superviews, and Multibase, generalization of VODAK, derived *same*-functions of COOL*, and derived *unifier*- and *image*-functions of Pegasus, are CP mechanisms and therefore of level II. Furthermore, adding attributes with constant values (cf. Section 4.2), as well as renaming schema elements (cf. Section 4.3) is CP.

An interoperability mechanism is of level III, iff it augments the information capacity of the global (composite) database. Any kind of adding stored and not any more derived information is CA and therefore of level III. Adding stored *same*-functions of COOL*, stored *image*-functions of Pegasus, and the *merge*-operation of O*SQL are examples of CA changes. Finally, most of the generic update operations of COOL* (e.g. **gain**) are level III operations as well, because they define implicitely new functions, and therefore augment the global information capacity.

5 Conclusion and Outlook

The contribution of this paper is a formal classification of multi-database languages into five levels with increasing strength of database integration and decreasing degree of local autonomy. The utility of this classification is twofold:

1. A designer of a new multi-database language is able to understand, what kind of concepts and mechanisms he is allowed to include into his language, in order to build a multi-database system of a particular, desired integration level. As a consequence, local CDBS autonomy and the possibilities for designing global query and update operations are well known. COOL* e.g. is defined as a scalable MDBS language.

2. A multi-database language may be classified into level I to IV according to the implemented concepts and mechanisms. This is very helpful to understand related work and to compare systems among each other. We argued for example, that Pegasus and O*SQL are mainly systems of integration level II (virtual integration), however, they include some very few concepts, making them finally level III systems (real integration).

Future work will include other MDBS languages, as well as the consideration of data model heterogeneity and transaction mechanisms. Whereas we think, that transaction mechanisms are orthogonal to the presented classification, it might be interesting to investigate, what kind of data model transformation mechanisms are possible at a particular integration level.

References

[1] R. Ahmed, et. al. An overview of Pegasus. In *Proc. 3rd Int'l Workshop on Research Issues on Data Engineering (RIDE-IMS)*, Vienna, Austria, Apr. 1993. IEEE Computer Society Press.

[2] C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4), Dec. 1986.

[3] R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15(3), 1986.

[4] Ingres Corp. *INGRES/Star User's Guide, Release 6.4*, Dec. 1991.

[5] W. Kent. The breakdown of the information model in multi-database systems. *ACM SIGMOD Record*, 20(4), 1991.

[6] T. Landers and R.L. Rosenberg. An overview of multibase. In *Proc. 2nd Int'l Symp. on Distributed Data Bases*, Berlin, Germany, Sept. 1982. North-Holland.

[7] W. Litwin. O*SQL: a language for multidatabase interoperability. In *Proc. IFIP DS-5 Semantics of Interoperable Database Systems*, Lorne, Australia, Nov. 1992.

[8] R.J. Miller, Y.E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. 19th Int'l Conf. on Very Large Data Bases (VLDB)*, Dublin, Irland, Aug. 1993.

[9] A. Motro. Superviews: virtual integration of multiple databases. *IEEE Trans. on Software Engineering*, 13(7), Jul. 1987.

[10] E.J. Neuhold and M. Schrefl. Dynamic derivation of personalized views. In *Proc. 14th Int'l Conf. on Very Large Data Bases (VLDB)*, Los Angeles, California, Sept. 1988. Morgan Kaufmann.

[11] Oracle Corp. *SQL*Net TCP/IP User's Guide, Version 1.2*, Nov. 1989.

[12] M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, and M. Tresch. The COCOON object model. Technical Report 193, ETH Zürich, Dept. of Computer Science, Dec. 1992.

[13] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proc. 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Munich, Germany, Dec. 1991. Springer, LNCS 566.

[14] M.H. Scholl, H.-J. Schek, and M. Tresch. Object algebra and views for multi-objectbases. In M.T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

[15] A.P. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneuos, and autonomous databases. *ACM Computing Surveys*, 22(3), Sept. 1990.