Federation views as a basis for querying and updating database federations

H. Balsters, E.O de Brock

University of Groningen
Faculty of Management and Organization
P.O. Box 800
9700 AV Groningen, The Netherlands
{h.balsters, e.o.de.brock}@bdk.rug.nl

Abstract. This paper addresses the problem of how to query and update so-called database federations. A database federation provides for tight coupling of a collection of heterogeneous component databases into a global integrated system. This problem of querying and updating a database federation is tackled by describing a logical architecture and a general semantic framework for precise specification of such database federations, with the aim to provide a basis for implementing a federation by means of relational database views. Our approach to database federations is based on the UML/OCL data model, and aims at the integration of the underlying database schemas of the component legacy systems to a separate, newly defined integrated database schema. One of the central notions in database modelling and in constraint specifications is the notion of a database view, which closely corresponds to the notion of derived class in UML. We will employ OCL (version 2.0) and the notion of derived class as a means to treat (inter-)database constraints and database views in a federated context. Our approach to coupling component databases into a global, integrated system is based on mediation. The first objective of our paper is to demonstrate that our particular mediating system integrates component schemas without loss of constraint information. The second objective is to show that the concept of relational database view provides a sound basis for actual implementation of database federations, both for querying and updating purposes.

1. Introduction

Modern information systems are often distributed in nature. Data and services are often spread over different component systems wishing to cooperate in an integrated setting. Cooperation of component systems in one integrated information system is becoming more and more important since information is often spread over different databases in an organization (or even spread over different organizations). Such information systems involving integration of cooperating component systems are called *federated information systems*; if the component systems are all databases then we speak of a federated database system ([ShL90]). This tendency to build integrated, cooperating systems is often encountered in applications found in EAI (Enterprise Application

Integration), which typically involve several, usually autonomous, component systems (data and service repositories), with the desire to query and update information on a global, integrated level. In this paper we will address the situation where the component systems are so-called legacy systems; i.e. systems that are given beforehand and which are to interoperate in an integrated single framework in which the legacy systems are to maintain as much as possible their respective autonomy.

A major obstacle in designing interoperability of legacy systems is the heterogeneous nature of the legacy components involved. This heterogeneity is caused by the design autonomy of their owners in developing such systems. To address the problem of interoperability the term *mediation* has been defined [Wie95]. A database federation can be seen as a special kind of mediation, where all of the data sources are (legacy) databases, and the mediator offers a mapping to a (virtual) DBMS-like interface. In our paper we will consider a *tightly-coupled approach* to database mediation, in which a global integrated schema of the federation is maintained. We base our approach on the "*Closed World Assumption*" (CWA, [Rei84]), where the integrated database is to hold -in some manner- the "union" of the data in the underlying component databases, without actually migrating any data from the component databases to the integrated database. The user of the federated system will be offered the impression that he is working with a monolithic homogeneous database system, while in fact this system basically resembles an interface, mapping interactions on the federated level to actions on the existing local database components. More precisely, the federated database will consist of an integrated *database view* on top of the existing legacy database components. For an overview of work on the virtual approach to database federation, we refer to [Hull97].

We will first concentrate on problems concerning schema integration of component legacy schemas on the level of the mediator. Once we have constructed a single, global schema for the database federation, we will subsequently offer a solution to the problem of defining an actual implementation of the integrated database; our solution is based on the concept of relational database view.

Schema integration requires the definition of relationships between schema elements of component systems. Detection and definition of such relationships can be heavily complicated by so-called *semantic heterogeneity* [DKM93,Ver97,TS01]. Semantic heterogeneity refers to differences in the meaning, interpretation, or intended use of related data. In [Ver97] semantic heterogeneity was treated in the context of a special-purpose modeling language for object-oriented databases ([BBZ93, BS98]). In this paper we will focus on the UML/OCL data model to tackle the problem of integrating semantic heterogeneity. UML/OCL offers a high-level specification language and is equipped with a unique combination of high expressiveness with a large degree of precision. Also, UML is the *de facto* standard language for analysis and design in object-oriented frameworks, and is being employed more and more for analysis and design of information systems, in particular information systems based on databases and their applications. In this paper, we will assume that component databases (e.g.

relational databases, cf. [BP98]) can somehow be modelled in the UML/OCL-framework.
One of the central notions in database modelling is the notion of a *database view*, which closely corresponds to the notion of derived class in UML. We will employ OCL and the notion of derived class as a means to treat database constraints and database views in a federated context. In [Bal02] it is demonstrated that in the context of UML/OCL the notion of derived class can be given a formal basis, and that derived classes in OCL have the expressive power of the relational algebra. Hence, OCL has the explicit power to emulate basic features of the relational query language SQL. Our paper demonstrates that our particular mediating system integrates component schemas without loss of constraint information; i.e., no loss of constraint information available at the component level may take place as result of integrating on the level of the virtual federated database. We will treat integration conflicts in a tightly-coupled environment, and show how to solve them by introducing a so-called *integration isomorphism*. This isomorphism will support the CWA-principle for database federations by correctly mapping a collection of legacy databases to a virtual integrated database. Key to establishing this integration isomorphism is the construction of a so-called *homogenizing function*; the homogenizing function (cf. [BB01, Bal03]) maps schemas of component databases to the schema of the integrated database. We note that this paper basically explains our approach in terms of (illustrative) examples; in [Bal03], however, we offer a more abstract and general approach by providing a heuristics and a methodology for constructing database federations from a more or less arbitrary collection of component databases.
The last part of our paper is concerned with offering a framework for actual implementation of the integrated database using the concept of relational database view. This implementation is constructed through successive and systematic mapping of the base tables in the component databases to the (virtual) tables in the integrated database through relational database views. This mapping is based on the integration isomorphism, described above. From the user perspective, the integrated database can then be addressed as if it were a normal, monolithic relational database, which can be queried and updated in the usual fashion. Querying boils down to querying the constructed views in SQL, while updating is regulated by means of properly defined checks on the involved views and by subsequently performing updates on the base tables occurring in the component databases. We conclude our paper with a section on architecture and the role of component autonomy.

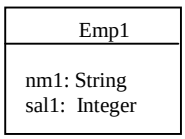
2. UML/OCL as a specification language for databases

Recently, researchers have investigated possibilities of UML as a modelling language for (relational) databases. [BP98] describes in length how this process can take place, concentrating on schema specification techniques. [DH99, DHL01] investigate further possibilities by employing OCL ([WK99, OCL2.0]) for specifying constraints and

business rules within the context of relational databases. The idea is that OCL provides expressiveness in terms of relatively abstract set definitions that should prove to be sufficient to capture the general notion of (relational) database view. In the more specific context of relational databases and OCL, [DH99] offer a framework for representing constraints within the relational data model. Current research, however, has not yet shown an effective way to deal with an important aspect of (relational) database modeling, namely modeling of so-called database views. A (database) view is a virtual table (or derived relation, in SQL), meaning that a view does not exist as a physical relation; rather a view is defined by an expression much like a query [GUW02]. Views, in turn, can be queried as if they existed physically, and in some cases, we can even modify view content. That is, a user is offered the impression that a view is some base relation inside the database, but in fact it is a derived (or virtual) relation defined in terms of the actual base relations constituting the database. View definitions are an important asset in database applications, because users are usually only interested in a part of the database, and not in the complete underlying corporate database. Hence, it is important that users have access to that part of the database considered relevant for their category of database applications. Our application area for views is focused on Federated Databases, where legacy databases are to interoperate by employing a so-called mediating system. This mediating system can be considered as an integration of a set of certain database views defined on the component legacy database systems.

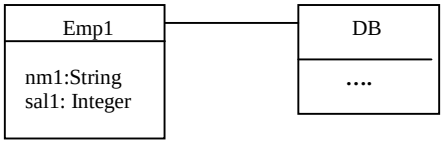
3. Basic principles: Databases and views in UML/OCL

Let's consider the case that we have a class called Emp1 with attributes nm1 and sal1, indicating the name and salary of an employee object belonging to class Emp1



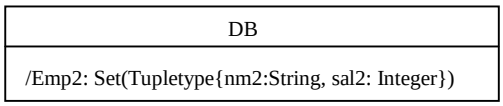
Now consider the case where we want to add a class, say Emp2, which is defined as a class whose objects are completely derivable from objects coming from class Emp1. The calculation is performed in the following manner. Assume that the attributes of Emp2 are nm2 and sal2 respectively (indicating name and salary attributes for Emp2 objects), and assume that for each object e1:Emp1 we can obtain an object e2:Emp2 by stipulating that e2.nm2=e1.nm1 and e2.sal2=(2 * e1.sal1). By definition the total set of instances of Emp2 is the set obtained from the total set of instances from Emp1 by applying the calculation rules as described above. Hence, class Emp2 is a *view* of class Emp1, in accordance with the concept of a view as known from the relational

database literature. In UML terminology [BP98], we can say that Emp2 is a *derived class*, since it is completely derivable from other already existing class elements in the model description containing model type Emp1.
We will now show how to faithfully describe Emp2 as a derived class in UML/OCL (version 2.0) in such a way that it satisfies the requirements of a (relational) view. First of all, we must satisfy the requirement that the set of instance of class Emp2 is the result of a calculation applied to the set of instances of class Emp1. The basic idea is that we introduce a class called DB that has an association to class Emp1, and that we define within the context of the database DB an attribute called Emp2. A database object will reflect the actual state of the database, and the system class DB will only consist out of one object in any of its states. Hence the variable *self* in the context of the class DB will always denote the actual state of the database that we are considering. In the context of this database class we can then define the calculation obtaining the set of instances of Emp2 by taking the set of instances of Emp1 as input.



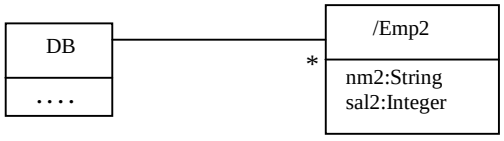

```
context  DB
def: Emp2: Set(Tupletype{nm2:String, sal2: Integer}) =
           (self.emp1 -> collect(e:Emp1 |
            Tuple{nm2=e.nm1, sal2=(2*e.sal1)}))-> asSet
```

In this way, we explicitly specify Emp2 as the result of a calculation performed on the base class Emp1. Graphically, we could represent Emp2 as follows



(1)

where the slash-prefix of Emp2 indicates that Emp2 is a derived attribute. Since in practice such a graphical representation could give rise to rather large box diagrams (due to lengthy type definitions), we will use the following (slightly abused) graphical notation (2) to indicate this derived class

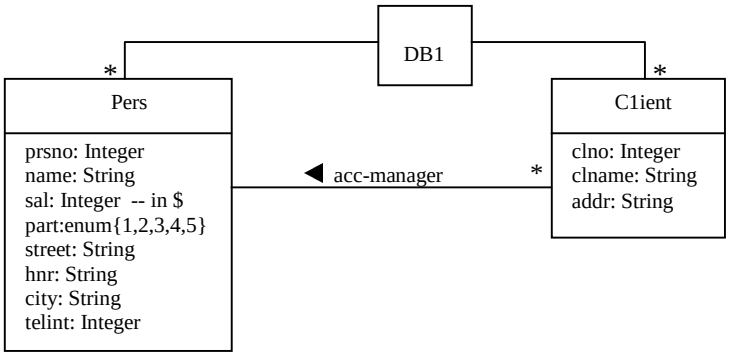


(2)

The intention is that these two graphical representations are to be considered equivalent; i.e., graphical representation (2) is offered as a diagrammatical convention with the sole purpose that it be formally equivalent (*translatable*) to graphical representation (1). Note that we have introduced a root class DB as an aid to represent the derived class \Emp2. Since in OCL, we only have the possibility to define attributes and operations within the context of a certain class, and class Emp1 is clearly not sufficient to offer the right context for the definition of such a derived construct as derived class Emp2, we had to move up one level in abstraction towards a class such as DB. A derived class then becomes a derived attribute on the level of the class DB.

4. Component frames

We can also consider a complete collection of databases by looking at so-called component frames, where each (labelled) component is an autonomous database system (typically encountered in legacy environments). As an example consider a component frame consisting of two separate component database systems: the CRM-database (DB1) and the Sales-database (DB2)

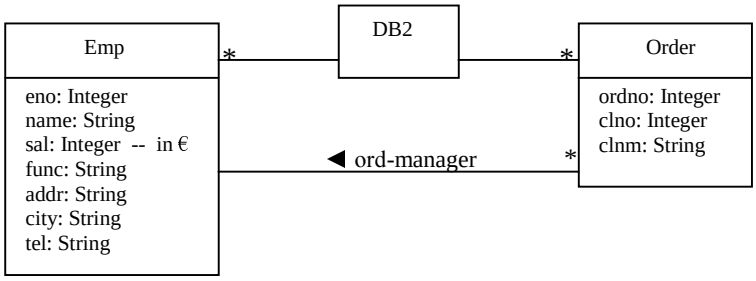


Most of the features of DB1 speak for themselves. We offer a short explanation of some of the less self-explanatory aspects: `Pers` is the class of employees responsible for management of client resource; `part` indicates that employees are allowed to work part time; `hnr` indicates house number; `telint` indicates internal telephone number; `acc-manager` indicates the employee (account manager) that is responsible for some client's account. We furthermore assume that database DB1 has the following constraints

```
context Pers inv:
Pers.allInstances --> isUnique (p: Pers | p.prsno)
sal <= 1500
telint >= 1000  and  telint <= 9999
```

```
context C1ient inv:
C1ient.allInstances --> isUnique (c: C1ient | c.clno)
```

The second database is the so-called Sales-database DB2

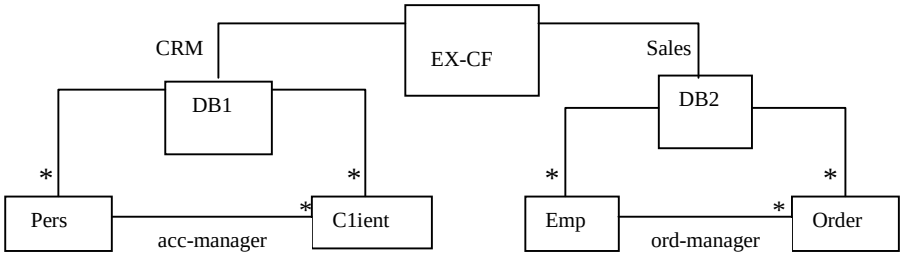


Most of the features of DB2 also speak for themselves. We offer a short explanation of some of the less self-explanatory aspects: `Emp` is the class of employees responsible for management of client order; `func` indicates that an employee has a certain function within the organization; `ord-manager` indicates the employee (account manager) that is responsible for some client's order.
We assume that this second database has the following constraints:

```
context Emp inv:
Emp.allInstances --> isUnique (p: Emp | p.eno)
sal >= 1000
tel.size <= 16

context Order inv:
Order.allInstances --> isUnique (c: Order | c.ordno)
Order.allInstances --> forall(c: Order | c.ord-manager.func =
`Sales')
```

We can now place the two databases DB1 and DB2 without confusion into one component frame EX-CF as seen in the following diagram



The two databases DB1 and DB2 are –in the case of this example- related, in the sense that an order-object residing in class Order in DB2 is associated to a certain client-object in the class C1ient in DB1. On the component frame level, we can define a auxiliary (partial) function mapping an order object in class Order to a client object class C1ient. We do this by assuming an operation in the class Order, called linkToC1ient.

```
context   Order::linkToC1ient( ): Client
post:     self.linkToClient.clno = self.clno
```

Since the attribute clno (in Client) has a unique value, the link from Order to C1ient is properly defined. (We have assumed that there always exists a corresponding clno-value in the class Client for each clno-value in the class Order. This is an example of a so-called *inter-database constraint*; we refer to Section 9 for more details on this category of constraints.)

5. Semantic heterogeneity: the integrated database DBINT

The problems we are facing when trying to integrate the data found in legacy component frames are well-known and are extensively documented (cf. [ShL90]). We will focus on one of the large categories of integration problems coined as *semantic heterogeneity* (cf. [Ver97]). Semantic heterogeneity deals with differences in intended meaning of the various database components. Integration of the source database schemas into one encompassing schema can be a tricky business due to: *renaming* (homonyms and synonyms); *data conversion* (different data types for related attributes); *default values* (adding default values for new attributes); *missing attributes* (adding new attributes in order to discriminate between certain class objects); *subclassing* (creation of a common superclass and subsequent accompanying subclasses).

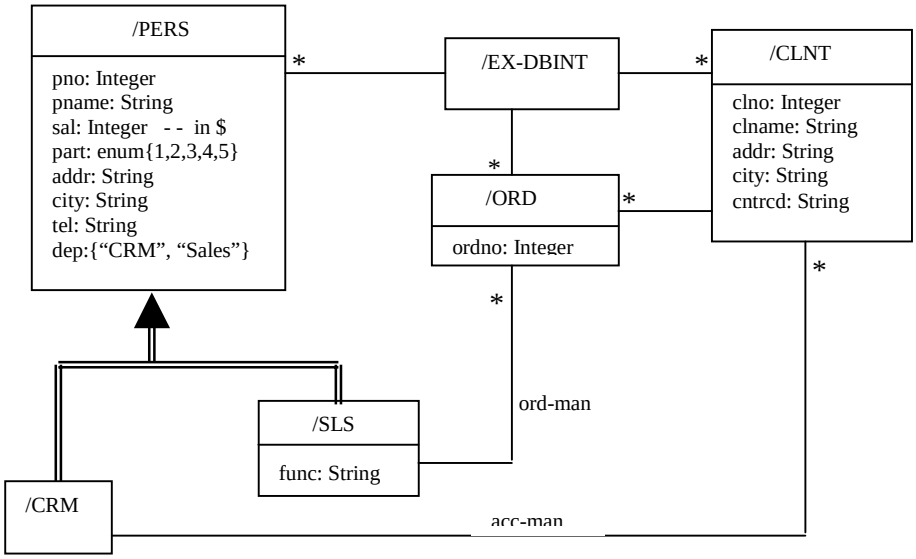
By homonyms we mean that certain names may be the same, but actually have a different meaning (different semantics). Conflicts due to homonyms are resolved by mapping two same name occurrences to different names in the integrated model. In the sequel, we will refer to this solutuion method as **hom**. Synonyms, on the contrary, refer to certain names that are different, but have the same semantics. **Synonyms** are treated analogously, by mapping two different names to one common name; this solution method is referred to by **syn**.

In the integration process, one often encounters the situation where two attributes have the same meaning, but that their domain values are differently represented. For example, the two attributes sal in the Pers and the Emp class of databases DB1 and

DB2, respectively, both indicate the salary of an employee, but in the first case the salary is represented in the currency dollars ($), while in the latter case the currency is given in euros (€). What we can do, is to convert the two currencies to a common value (e.g. $, invoking a function convertTo$). Applying a conversion function to map to some common value in the integration process, is indicated by **conv**.
Sometimes an attribute in one class is not mentioned in another class, but it could be added there by offering some suitable default value for all objects inside the second class. As an example, consider the attribute `part` in the class Pers (in DB1): it could also be added to the class Emp (in DB2) by stipulating that the default value for all objects in Emp will be 5 (indicating full-time employment). Applying this principle of adding a default value in the integration process, is indicated by **def**.
The integration of two classes often calls for the introduction of some additional attribute, necessary for discriminating between objects originally coming from these two classes. This will sometimes be necessary to be able to resolve seemingly conflicting constraints. As an example, consider the classes Pers (in DB1) and Emp (in DB2). Class Pers has as a constraint that salaries are less than 1500 (in $), while class Emp has as a constraint that salaries are at least 1000 (in €). These two constraints seemingly conflict with each other, obstructing integration of the Pers and the Emp class to a common class, say PERS. However, by adding a discriminating attribute `dep` indicating whether the object comes from the CRM or from the SLS department, one can differentiate between two kinds of employees and state the constraint on the integrated level in a suitable way. .Applying the principle of adding a discriminating attribute to differentiate between two kinds of objects inside a common class in the integration process, will be indicated by **diff**. The situation of a missing attribute mostly goes hand in hand with the introduction of appropriate subclasses. For example, introduction of the discriminating attribute `dep` (as described above), entails introduction of two subclasses, say CRM and SLS of the common superclass PERS, by listing the attributes, operations and constraints that are specific to CRM- or SLS-objects inside these two newly introduced subclasses. Applying the principle of adding new subclasses in the integration process, is indicated by **sub**.

6. The integrated database DBINT

We now offer our construction of a virtual database EX-DBINT, represented in terms of a derived class in UML/OCL. The database we describe below, intends to capture the integrated meaning of the features found in the component frame described earlier.



This database has the following constraints:

```
context PERS inv:
PERS.allInstances ->
forall(p1, p2: PERS | (p1.dep=p2.dep and p1.pno=p2.pno)  implies
p1=p2)
PERS.allInstances ->
forall(p:PERS |
(p.oclIsTypeOf(SLS) implies (p.sal >= 1000.convertTo$ and
part=5)) and (p.oclIsTypeOf(CRM) implies p.sal <= 1500 ))
tel.size <= 16

context CLNT inv:
Clnt.allInstances --> isUnique (c: CLNT | c.clno)

context ORD inv:
Order.allInstances --> isUnique (o: ORD | o.ordno)
ord-manager.func = `Sales'
```

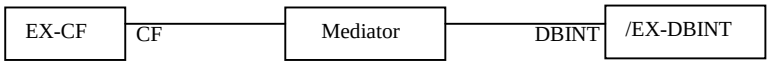
We shall now carefully analyze the specification of this (integrated) database EX-DBINT, and see if it captures the intended meaning of integrating the classes in the component frame EX-CF and resolves potential integration conflicts.

Conflict 1: Classes Emp and Pers in EX-CF have partially overlapping attributes, but Emp has no attribute `part` yet, and one still needs to discriminate between the two kinds of class objects (due to specific constraints pertaining to the classes Emp and Pers). Our solution in DBINT is based on applying **syn + def + diff + sub** (map to common class name (`PERS`); add a default value (to the attribute `part`); add an extra discriminating attribute (`dep`); introduce suitable subclasses (`CRM` and `SLS`)).
Conflict 2: Attributes `prsno` and `eno` intend to have the same meaning (a *key constraint*, entailing uniquely identifying values for employees, for Emp- and Pers-objects). Our solution in DBINT is therefore based on applying **syn** + **diff** (map to common attribute name (`pno`); introduce extra discriminating attribute (`dep`)) and enforce uniqueness of the value combination of the attributes `pno` and `dep`.
Conflict 3: Attributes `sal` (in Pers) and `sal` (in Emp) partially have the same meaning (salaries), but the currency values are different. Our solution is based on applying **conv** (convert to a common value).
Conflict 4: The attribute combination of `street` and `hnr` (in Pers) partially has the same meaning as `addr` in Emp (both indicating address values), but the domain values are differently formatted. Our solution is therefore based on applying **syn + conv** (map to common attribute name and convert to common value).
Conflict 5: Attributes `telint` (internal telephone number) and `tel` (general telephone number) partially have the same meaning, but the domain values are differently formatted. Our solution is therefore based on applying **syn + conv** (map to common attribute name and convert to common value).

7. Integrating by mediation

Our strategy to integrate a collection of legacy databases –given in some component frame CF- into an integrated database DBINT is based on two principles, being: tightly-coupled approach to database integration, followed by conformance to the Closed World Assumption of Database Integration (***CWA-INT***).
The principle of ***CWA-INT*** can informally be described as follows: *an integrated database is intended to hold exactly the "union" of the data in the source databases in the component frame CF*. Requirement ***CWA-INT*** is a direct extension of the traditional *Closed World Assumption* (CWA) found in the database literature. This assumption (CWA) reads as follows: the only possible instances of a relation are those implied by the database ([Rei84]). In this sense, a database is considered to be complete. Extending CWA to the context of database *integration*, is first discussed in [Hull97], leading to the assumption that we have coined as ***CWA-INT***. This (informal) requirement has to be further investigated for consequences when applied to querying and to updating an integrated database. In more mathematical terms, we will demand that the universe of discourse of component frame CF and the universe of discourse of the integrated database DBINT are, in a mathematical sense, *isomorphic*; only in this

way will we not lose any information when transforming the legacy components to the integrated database. We will demonstrate, in terms of constraints described in OCL, that the universe of discourse of our example component frame EX-CF and the universe of discourse of the example integrated database EX-DBINT are indeed isomorphic. We shall coin this isomorphism as the so-called ***integration isomorphism***.
We will describe a UML model containing a class, called the mediator, explicitly relating the component frame EX-CF and the virtual integrated database EX-DBINT. We will do so, by systematically exploiting various conversion functions, linking objects in the component frame EX-CF to objects in the integrated database EX-DBINT. Constructing these links is done in a very deliberate fashion, with the aim to establish an integration isomorphism between EX-CF and EX-DBINT.
In our setting, mediation is performed by introducing an explicit class Mediator, connecting EX-CF and EX-DBINT



The mediator has the task to correctly link the component frame EX-CF to the (virtual) database EX-DBINT. This is not a trivial task and involves a precise mapping of component elements to the virtual database. The mapping also has to take into account various constraint conditions which rule inside EX-CF. We do this by introducing suitable conversion operations inside the classes.

8. Mapping the component frame to the virtual integrated database

In this section we will describe how to add a method, called Hom, to the top-level class EX-CF resulting in an element (database state) of the integrated database EX-DBINT. We assume that we have the following type abbreviations at our disposal:

```
PERSTYPE = TupleType(pno: Integer, pname: String, sal: Integer,
part: enum{1,2,3,4,5}, addr: String, city: String, tel: String,
dep: enum{`CRM', `Sales'})

SLSTYPE = TupleType(pno: Integer, pname: String, sal: Integer,
part: enum{1,2,3,4,5}, addr: String, city: String, tel: String,
dep: enum{`CRM', `Sales'}, bonus: Integer, func: String)

CLNTTYPE = TupleType(clno: Integer, clname: String, addr: String,
acc-man: PERSTYPE)

ORDTYPE =  TupleType(ordno: Integer, ord-man: SLSTYPE)
```

Furthermore, we assume the existence of a conversion function convertToCLNT within the class Client (of DB1) with the following definition

```
context   Client
def: convertToCLNT( ): CLNTTYPE = Tuple{clno=self.clno,
clname=self.clname, addr=self.addr,
acc-man=self.acc-manager.convertToCRM}
```

In the Pers-class we postulate the existence of a conversion function convertToCRM

```
context   Pers
def:  convertToCRM( ): PERSTYPE = Tuple{pno=self.prsn,
pname=self.name, sal=self.sal, part=self.part,
addr= self.street ->concat((`  ') ->concat(self.hnr)),
city=self.city, tel= `+31-50-363'->concat(self.telint),
dep=`CRM'}
```

Notice that the function convertToCRM is injective! (We have assumed that the attribute hnr does not contain two consecutive spaces.) We also define two functions converting the objects in the Emp-class to corresponding objects in the SLS- and PERS-class of DBINT, by the conversion functions convertToSLS and convertToPERS within the class Emp with the following (rather trivial) definition

```
context   Emp
def:  convertToSLS( ): SLSTYPE = Tuple{pno=self.eno,
pname =self.name, sal=self.sal.convertTo$, part=self.part,
addr=self.addr, city=self.city, tel=self.tel, dep=`SLS',
bonus=self.bonus, func=self.func}

def:  convertToPERS( ): PERSTYPE = Tuple{pno=self.eno,
pname= self.name, sal=self.sal.convertTo$, part=self.part,
addr= self.addr, city=self.city, tel=self.tel, dep=self.dep}
```

A bit more difficult is the definition of a function converting the objects in the Order-class to corresponding objects in the ORD-class of DBINT. We do this by employing a conversion function convertToORD within the class Order with the following definition

```
context   Order
def:   convertToORD( ):ORD = Tuple{ordno=self.ordno,
ord-man=(self.ord-manager).convertToSLS,
clnt= (self.linkToC1ient).convertToClnt}
```

where the previously defined operation linkToC1ient provides the link to the unique Client-object associated to a given Order-object.
We are now in the position to relate the component frame CF to the integrated database, coined EX-DBINT. We shall proceed by first defining a basic type

DBINTTYPE, and showing how we can define a homogenizing function Hom inside the class EX-CF, mapping elements of the component frame to the integrated database.

```
DBINTTYPE  =  TupleType(CRM: Set(PERSTYPE), SLS: Set(SLSTYPE),
CLNT: Set(CLNTTYPE), ORD: Set(ORDTYPE), PERS: Set(PERSTYPE)}
```

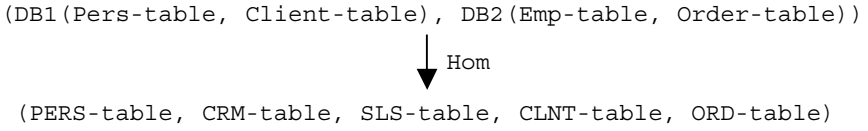
We now introduce the definition of the homogenizing function within the context of the component frame class EX-CF:

```
context    EX-CF
def:  Hom( ):DBINTTYPE = Tuple{CRM=(self.CRM.Pers.allInstances ->
collect(p| p.convertToCRM))-> asSet,
      SLS=(self.Sales.Emp.allInstances ->
collect(e| e.convertToSLS))-> asSet,
      CLNT=(self.CRM.C1ient.allInstances ->
collect(c| c.convertToCLNT))-> asSet,
      ORD= (self.Sales.Order.allInstances ->
collect(o| o.convertToORD)) -> asSet,
      PERS= (((self.CRM.Pers.allInstances ->
collect(p| p.convertToCRM))-> union(self.Sales.Emp.allInstances))
           -> collect(e | e.convertToPERS)) -> asSet}
```

With this homogenizing function we can define the missing link providing the definition of the virtual database DBINT. We do this by adding the appropriate definition to the mediator class.

```
context Mediator
def: EX-DBINT: DBINTTYPE = Tuple{CRM= (self.CF.Hom).CRM,
SLS= (self.CF.Hom).SLS, CLNT= (self.CF.Hom).CLNT,
ORD= (self.CF.Hom).ORD, PERS= (self.CF.Hom).PERS}
```

The homogenizing function Hom has the following effect: it maps a *CF-state* to a *DBINT-state*, as depicted below



It is now easily verified that the combination of the definition of the homogenizing function together with the definition of EX-DBINT offered in the Mediator class, indeed results in an integration isomorphism linking the component frame EX-CF to the integrated database EX-DBINT. Our definition of EX-DBINT also captures the desired constraints specified in Section 6. More details on the integration isomorphism can be found in [Bal03].

There is still one category of constraints that we have to deal with in order to get the picture complete, the so-called *inter-database consraints*, described in the next section.

9. Inter-database constraints

Additional information analysis might reveal the following two wishes regarding data in the component frame CF:

(1) Nobody should be registered as working for both the CRM and Sales department; i.e., these departments should have no employees in common
(2) Client numbers in the Sales database should also be present in the CRM database

On the level of DBINT, these constraints are specified as follows (taking into effect the mapping properties of the mapping Hom):

```
context  DBINT  inv:
let  X= (self.CRM.allInstances -> collect(c:CRM| c.pno))-> asSet
let  Y= (self.SLS.allInstances -> collect(s:SLS| s.pno))-> asSet
let  V= (self.CLNT.allInstances ->
         collect(c:CLNT| c.clno))-> asSet
let  W= (self.ORD.allInstances ->
         collect(o: ORD| o.clno))-> asSet
in
(X-> Intersect(Y)) -> isEmpty and
(W-> forall(w:W| V-> exists(v:V| w=v)))
```

In this way, we can easily specify constraints that are actually inter-database constraints on the componet frame level (namely between the databases DB1 and DB2) as table constraints on the level of DBINT.
Now that we have constructed our integration isomorphism, and we also have described how to deal with inter-database constraints, the road is open to offer an actual implementation of DBINT, as described in the following section.

10. How do we implement the integrated database DBINT?

Our aim in implementing DBINT –from the *user perspective*- is to be able to treat DBINT as a normal, monolithic database and, hence, query and update DBINT in the usual fashion. We will demonstrate that we can reach our aim through succesive and systematic mapping of the base tables in CF to the (virtual) tables in DBINT by exploiting the concept of relational database view. Our mapping is based on the integration isomorphism, which offers us a way to traverse in a unique manner from the integrated database to the component databases, and vice versa.

Our approach is basically as follows

• construct *SQL-views* of each of the tables described in DBINT
• assume that we –somehow- have relational representations of the base tables in the component frame CF (e.g. via *gateways*, cf. [HP97])
• querying DBINT now boils down to querying the constructed SQL-views
• initially, the views will respect the constraints present in DBINT, and in the case that we want to update DBINT, we will use a suitable *updating action* to first perform a check on the involved views and subsequently perform updates on the base tables in CF
• we can now update through the constructed views, since the homogenizing function constitutes an **isomorphism**, and each tuple in a view corresponds to *exactly one combination of tuples* in the base tables

In the following sections we will show how we realize these SQL-views, in this paper called ***federation views***, and also how to properly define a database trigger in the case that we want to perform updates on the federation.

Constructing federation views

We will now show how to construct SQL-views of the tables in DBINT. First we will construct a view of the PERS-table

```
CREATE VIEW PERS(pno, pname, sal, part, addr, zip, city, tel,
cntrcd, dep) AS
(SELECT t.prsno, t.name, t.sal, t.part, t.street&`'&t.hnr,
Num2Chr(t.zip.num)&`'&t.zip.letcom, t.city, `+31-50-363' &
Num2Chr(t.telint), `NL', `CRM'
FROM CF@CRM.Pers t)
UNION
(SELECT t.eno, t.name, convertTo$(t.sal), 5, t.addr, t.zip,
t.city, t.tel, t.cntrcd, `SLS'
FROM CF@Sales.Emp t);
```

Note that we have left open how to describe, in terms of SQL, the actual implementation of the function `convertTo$`. In any case, we have to achieve that this function turns out to be injective. (Should this not be possible in a direct manner, then we could resort to adding an extra attribute, e.g. using a separate currency attribute indicating which currency symbol is actually used.)

SQL-views of the tables `SLS`, `CRM`, `CLNT`, and `ORD` are more or less straightforward:

```
CREATE VIEW SLS(pno, bonus, func) AS
SELECT t.eno, t.bonus, t.func
FROM CF@Sales.Emp t;


CREATE VIEW CRM(pno) AS
SELECT t.pno
FROM PERS t
WHERE t.dep=`CRM';


CREATE VIEW CLNT(clno, clname, addr, zipcity, cntrcd, acc-man-no)
AS
SELECT t.clno, t.clname, t.addr, t.zipcity, t.cntrcd, t.acc-
manager
FROM CF@CRM.Client t;


CREATE VIEW ORD(ordno, ord-man-no, clno) AS
SELECT t.ordno, t.ord-manager, t.clno
FROM CF@Sales.Client t;
```

Note that we have used standard relational representations for OO-concepts (cf. [BP98]) in traversing from UML-representations of the virtual table to the associated tables in SQL. Note also that these view definitions directly reflect the definition of the integration isomorphism described above.
Each of the views described above is an example of what we shall call a *federation view*; i.e., we wish to conceive of the database federation as a collection of database views, where each database view reflects a virtual table defined in the integrated database DBINT. Manpulating the federation (i.e., querying and updating), now means manipulating the collection of federation views.
Querying can now be done directly on these federation views defined, but updating is, however, a different matter altogether! The reason is that updating has to take into account the various conditions described in the constraints on the level of DBINT.
The next section deals with representation of constraints, and how to suitably define actions to deal with updates.

11. Updating federation views

Before we can perform an update on one of the federation views, we first have to see if certain constraints are satisfied. These constraints can be split into two categories: those that are essentially local to the databases in the component frame CF, and those that are essentially global and pertain to the federation. Since we do not want to get involved into unnecessary ACID-violations due to a non-commiting update on one of the local databases, we must also take local database integrity into account before we decide to actually try to perform an update on the base tables. We will deal with matters

pertaining to concurrency control in a later section. First we treat the representation of constraints within the collection of federation views.

The constraints

Consider the constraint on the view PERS, pertaining to the non-overlap between views SLS and CRM

```
CREATE VIEW NOCOMMONCRMSLS AS
(SELECT pno FROM SLS) INTERSECT (SELECT pno FROM CRM);

C1(PERS):   (SELECT COUNT(*) FROM NOCOMMONCRMSLS) = 0
```

Alternatively, this constraint can be specified by

```
(SELECT COUNT(*) FROM PERS)=
(SELECT COUNT (DISTINCT pno) FROM PERS)
```

On view ORD we have the following constraint pertaining to referential integrity between the ORD- and the CLNT class

```
C2(ORD):  (SELECT COUNT(*)
           FROM ORD t
           WHERE NOT (t.clno IN (SELECT clno FROM CLNT)) = 0
```

How to update

Eventually, we will offer a means to specify an action that performs an update on the views PERS, CRM, SLS, CLNT, and ORD. We shall first, however, set out the boundaries within which we will conduct our investigations. In this paper we will concentrate on a simplified situation concerning transactions on DBINT, by adopting the following assumption

The global user is the only user of the federation (including the databases in the component frame) during execution of the global transaction

In this case, we abstract from problems due to concurrency control and concentrate on the situation that we only have one user performing an update on the database DBINT. Showing how to correctly perform an update on DBINT in this simplified case, is a first step that has to be solved before dealing with the more complex case with problems due to concurrency control. In this paper we will refrain from treating the more general case including multi-user updates, as this is a matter of ongoing research.

We shall start by considering an insert of a tuple t in view PERS. In order to perform this update, we commence with a series of checks:

1. Check if t satisfies the gobal constraints of DBINT: `C1(PERS union {t}).`

This constraint is coined here as `Global(t)`

2a. If t is a CRM-tuple, check and see if t satisfies the tuple- and table constraints of CRM (e.g. , check to see if attribute pno remains to be a key) :

```
t.sal<=1500 and t.tel>=1000 and t.tel<=9999 and
     t.pno NOT IN (SELECT u.pno FROM CRM u)
```

This constraint is coined here as `Local-CRM(t)`

2b. If t is a SLS-tuple, check and see if t satisfies the tuple- and table constraints of SLS :

```
t.sal>=convertTo$(1000)  and  t.bonus>0  and
   t.pno NOT IN (SELECT u.pno FROM SLS u)
```

This constraint is coined here as `Local-SLS(t)`

3. If t satisfies the constraints mentioned in 1-2a (2b) then we can insert `CF-Format(t)` into either `CF@CRM.Pers` or `CF@Sales.Emp`

Here, `CF-Format(t)` denotes the *inverse construction* of tuple `t` offering a format for `t` suitable for insertion in either `CF@CRM.Pers` or `CF@Sales.Emp`.
`CF-Format` is defined by considering the following four mappings from the underlying tuple types in the component frame to a corresponding tuple in DBINT

- `Pers` is mapped to `CRM` via the (injective!) function `convertToCRM`
- `CRM@Client` is mapped to `CLNT` via the (injective) function `convertToCLNT`
- `Emp` is mapped to `SLS` via the (injective) function `convertToSLS`
- `Sales@Client` is mapped to `ORD` via the (injective) function `convertToORD`

Since all these four function have inverses, it is clear that an expression `CF-Format(t)` is always uniquely defined for any tuple t in PERS!

A suitable action in the case of an insert in the CRM-view could now look like this:

```
   WHEN Global(t) and Local-CRM(t)
INSERT INTO CF@CRM.Pers(CF-Format(t))
```

We note that a conditon is first checked (in the WHEN-clause) and in the case that the condition yields to `true`, the action

```
INSERT INTO CF@CRM.Pers(CF-Format(t))
```

is performed. Should the condition evaluate to `false`, then no action is performed at all.
A similar action could also be specified in the case that tuple t is to be inserted in the SLS-view.
In our treatment of updates, we have assumed that we can express all relevant constraint checks pertaining to tuple t on the level of the *federation*! This assumption is realistic, since all local constraints (i.e. expressed in component frame CF) have all been taken into account in the initial construction of DBINT. This makes dealing with checks on federation updates easier to handle, since constraint checking can then be completely delegated from the level of the component frame CF to the level of DBINT.

12. Architecture of federated databases based on mediation

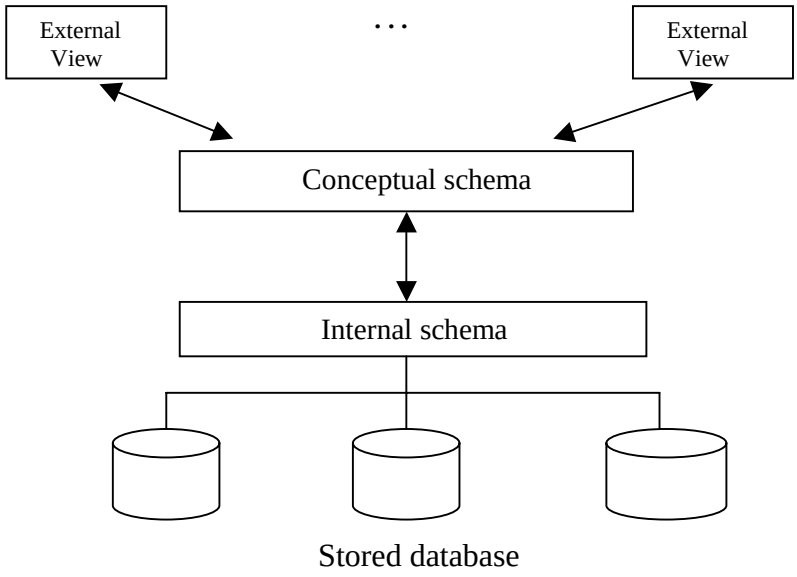
In this section we will have a closer look at our particular choice of architecture for federated databases. In particular, we are interested in answering the following two questions:

• How does this architecture compare to classic (i.e., monolithic) database architecture?
• What impact does this architecture have on the issue of site autonomy?

Traditionally, a monolithic database system is based on what is called the **three-schema architecture** (also known as the ANSI/SPARC architecture), which was proposed to separate user applications, the conceptual schema of the database, and the physical database (cf. [EN00]). In this architecture, schemas can be defined at three levels:

1. The internal level has an **internal schema**, which describes the physical storage structure of the database
2. The conceptual level has a **conceptual schema**, which describes the complete database for the whole community of users. This schema abstracts from physical storage structures, and concentrates on entities, types, relationships, constraints, and operations
3. The external or view level includes a number of **external schemas** or **user views**. Each external schema describes that part of the conceptual schema of

the database that is relevant to a particular group of users, and hides other parts that are not relevant to that particular group



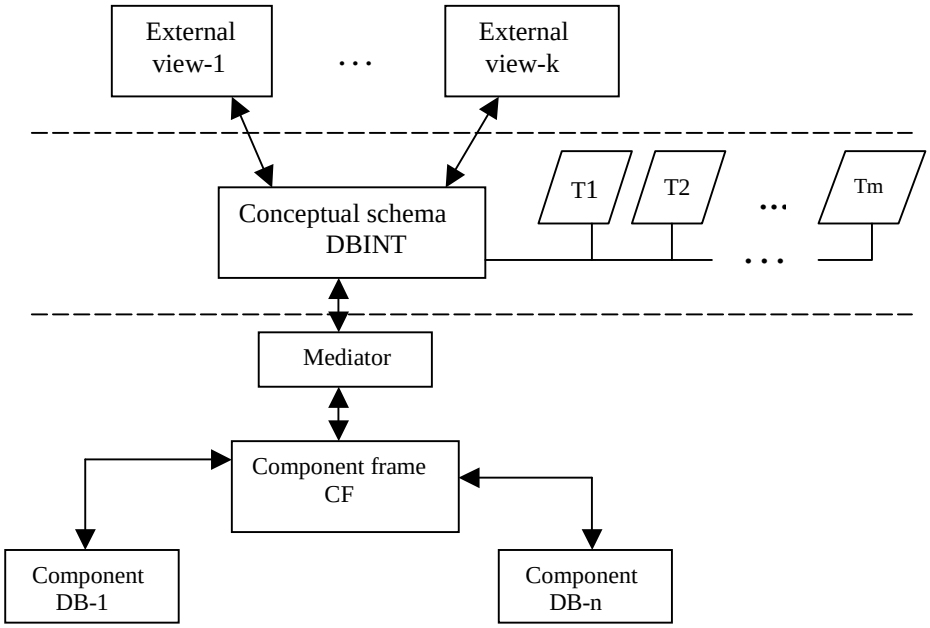
The processes of transforming requests and results between the levels are called **mappings**.

This architecture has the advantage to support the so-called **data-independence property**, meaning that one can change the conceptual schema without having to change the external schema (*logical data independence*), and also that one can change the internal schema without having to change the conceptual schema (*physical data independence*).

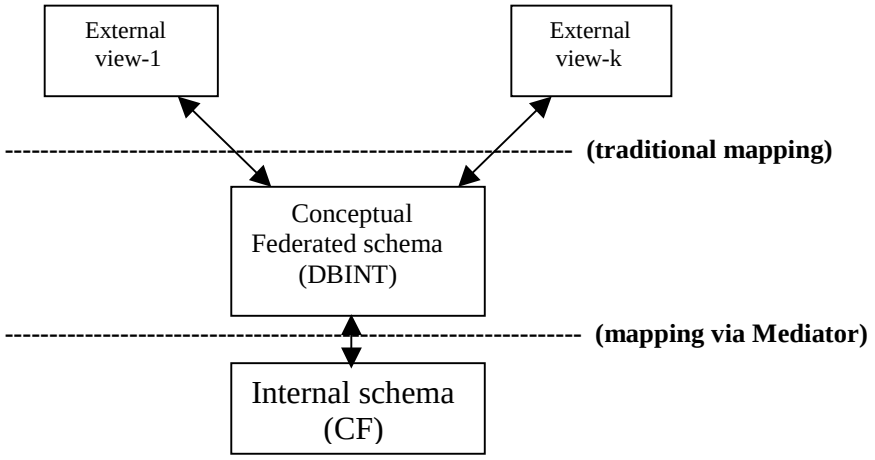
In our setting, we deal with a collection of component databases inside some component frame, with the aim to integrate these component databases, with a federated database as result. As described in Section 7, integration is based on the principle of the tightly-coupled approach in combination with the principle of the Closed World Assumption of Database Integration (CWA-INT). In this section we will demonstrate how to achieve an architecture for a federated database, based on these two principles.

We will assume that each of these component databases internally abide to the three-schema architecture as described above. We are now faced with the problem of what the architecture of the *federated database* looks like. Actually, the solution is quite straightforward. The idea is that the integrated database DBINT contains the

conceptual schema of the federation, consisting of a collection of *federation views*, and that user groups of the federation define their own user views (with their own separate external schemas) on top of DBINT. We can depict this architecture as follows



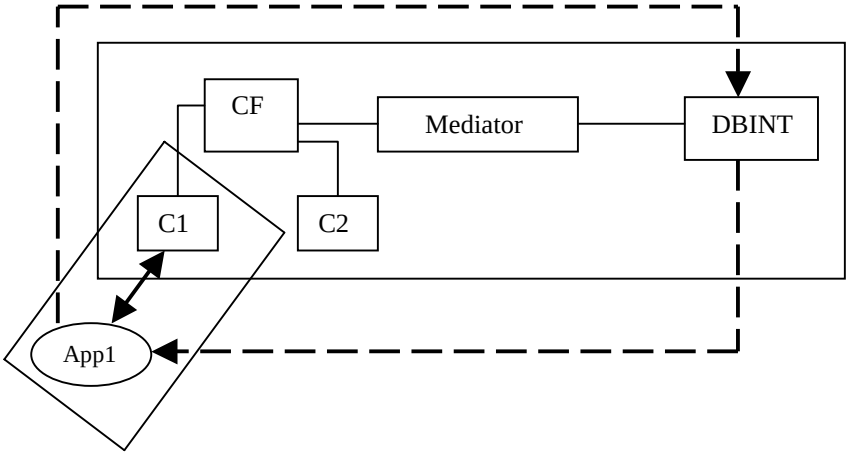
where *n* component databases (each abiding internally to their own 3-level architecture) are integrated (via CF and the Mediator), resulting in the database schema of DBINT (representing the conceptual schema of the *database federation*), containing m tables T1, …, Tm (all of which are defined as *federation views*), and where subsequently a number of *k* external views are defined on top of the (conceptual) schema of DBINT. If we succeed in offering a mapping constituting an *integration isomorphism* from the component frame CF to the integrated database DBINT, then we shall also have succeeded in realizing a database federation abiding to the Closed World Assumption CWA-INT, as explained earlier on. This will be our eventual goal of integration.
In this perspective, the architecture of a federated database is basically still much along the lines of a traditional three-level architecture (user views on top of a conceptual schema of a federation, and the eventual internal schema realized via the mediator as a combination of internal schemas of component databases inside a component frame). We therefore call this architecture a "**three-level federation architecture**", which can be concisely depicted as follows



Analogous to the original three-level architecture, this three-level federation architecture also supports the principles of both logical- and physical data independence. The only difference is that the mapping between the conceptual level and internal level is defined within the context of the database federation, which now is defined via the mediator and the component frame.

Component autonomy

We now proceed with a discussion on so-called *component autonomy* in database federations. In federated database literature it is often claimed that the component databases should maintain their respective autonomy as much as possible. In practice this makes sense, because a database federation, as we have seen, is actually no more than a database view on a component frame; i.e. the component databases remain intact, and the federated database is no more than a calculation resulting in a virtual integrated database on the global level. Updating the federated database then boils down to updating associated federation views (cf. the previous section). Using federation views, we have taken into account that by allowing a database to become a member of the federation (i.e. the database becomes a component database in a component frame), such an update might also become subject to certain inter-database constraints. This means that an autonomous update on a local component database might violate an inter-database constraint, thus eventually rendering it as an incorrect update. Hence, local updates –in a federated setting- are in principle also updates on the complete federation!
The situation described above can depicted in the following diagram



In this diagram, App1 is an old application running on database C1. We would ideally like App1 to keep on running on C1, as if it had no knowledge of the fact that from some moment in time, C1 has become a member of the federation. We could realize this by taking the following measure. We construct a new application App1' equal to application App1, except that we first perform a check to see if the global constraints of the federation are satisfied. That is, App1' is defined by

```
App1'= (Global-constraints  →  App1)
```

stating that App1 is performed only in the case that the global constraints of the federation are satisfied. This has as an effect, that App1 will not run in the case of a violation of an inter-database constraint. Should the inter-databae constraints be satisfied then App1 can run on component database C1 as it normally does. (It could still possibly not run due to violation of a local constraint belonging to C1, but that would also have been the case if C1 had not become a member of the federation.)
We note that it will often be necessary to have knowledge of the external effect of App1 in order to determine the exact specification of the condition `Global-constraints` (e.g., in the case of updates resulting from invocation of App1).

13. Future research

Our paper offers a first step to actual implementation of a database federation. We have demonstrated that a suitable integration isomorphism can provide for the sound definition of a database federation as a collection of so-called federation views. There is, however, still much work to be done. Much of this work pertains to the *database*

functionality of the integrated database DBINT. We have, for example, abstracted from problems dealing with concurrency control in the case of updates on DBINT. More complex manipulation of data on the level of DBINT (e.g. selections and joins) have also not yet been treated. Another matter is that we have assumed that we can, somehow, provide for relational representations (for example via suitable interfaces or gateway constructions) of the component databases. We are addressing these issues as part of our ongoing research on sound implementations of database federations.

References

[AB01] Akehurst, D.H., Bordbar, B.; On Querying UML data models with OCL; «UML» 2001 4th Int. Conf., LNCS 2185, Springer, 2001

[Bal02] Balsters, H. ; Derived classes as a basis for views in UML/OCL data models; SOM Report 02A47, University of Groningen, 2002

[BB01] Balsters, H., de Brock, E.O.; Towards a general semantic framework for design of federated database systems ; SOM Report 01A26, University of Groningen, 2001

[Bal03] Balsters, H.; Object-oriented modeling and design of database federations; SOM Report 03A18, University of Groningen, 2003

[BBZ93] Balsters, H., de By, R.A., Zicari, R.; Sets and constraints in an object-oriented data model; Proc. 7th ECOOP, LNCS 707, Springer 1993.

[BP98] Blaha, M., Premerlani, W.; Object-oriented modeling and design for database applications; Prentice Hall, 1998

[BS98] Balsters, H., Spelt, D.; ``Automatic verification of transactions on an object-oriented database", 6th Int. Workshop on Database Programming Languages, LNCS 1369, Springer, 1998.

[DH99] Demuth, B., Hussmann, H.; Using UML/OCL constraints for relational database design; «UML»'99: 2nd Int. Conf., LNCS 1723, Springer, 1999

[DHL01] Demuth, B., Hussmann, H., Loecher, S.; OCL as a specification language for business rules in database applications; «UML» 2001, 4th Int. Conf., LNCS 2185, Springer, 2001

[DKM93] Drew, P., King, P.R., McLeod, D., Rusinkievicz, M., Silberschatz, A.; Workshop on semantic heterogeneity and interoperation in multidatabase systems; SIGMOD RECORD 22, 1993

[EN00] Elmasri, R., Navathe, S.B.; Fundamentals of database systems; Addison Wesley, 2000

[GR97] Gogolla, M., Richters, M.; On constraints and queries in UML; Proceedings UML'97 Workshop "The Unified Modeling Language – Techniques and Applications", 1997

[GUW02] Garcia-Molina, H., Ullman, J.D., Widom, J.; Database systems PrenticeHall, 2002

[HP97] Hewlett Packard (White paper); Database gateway use in heterogeneous

environments; 1997

[Hull97] Hull, R.; Managing Semantic Heterogeneity in Databases; ACM PODS'97, ACM Press 1997.

[MC99] Mandel, L., Cengarle, M.V.; On the expressive power of OCL; Formal Methods '99 – LNCS 1708, Springer, 1999

[OCL2.0] Response to the UML 2.0 OCL RfP, Revised Submission, Version 1.6, January 6, 2003

[Rei 84] Reiter, R.; Towards a logical reconstruction of relational database theory. In: Brodie, M.L., Mylopoulos, J., Schmidt, J.W.; On conceptual modeling; Springer Verlag, 1984

[ShL90] Sheth, A.P., Larson, J.A.; Federated database systems for managing distributed and heterogeneous and autonomous databases; ACM Computing Surveys 22, 1990

[TS01] Türker, C., Saake, G.; Global extensional assertions and local integrity constraints in federated schemata; Information Systems, Vol. 25, No.8, 2001

[Ver97] M. Vermeer; Semantic interoperability for legacy databases. Ph.D.-thesis, University of Twente, 1997.

[Wie95] Wiederhold, G.; Value-added mediation in large-scale information systems IFIP Data Semantics (DS-6), 1995

[WK99] Warmer, J.B., Kleppe, A.G.; The object constraint language; Addison Wesley, 1999