An Approach to Schema Integration Based on Transformations and Behaviour*

Christiaan Thieme and Arno Siebes

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{ct,arno}@cwi.nl

Abstract

This article presents an approach to schema integration that combines structural aspects and behavioural aspects. The novelty of the approach is that it uses behavioural information to guide both schema restructuring and schema merging. Schema restructuring is based on schema transformations and schema merging is based on join operators.

1 Introduction

Schema integration is an important and non-trivial task in database design. It occurs when a number of different user views, developed for a new database system, or a number of existing database schemas must be integrated into a global, unified schema. As schema integration is a difficult task, methods to support the designer with this task are essential. In [6], a framework for comparing integration methods is given. The framework identifies four steps. In the first step, the preintegration step, an integration strategy is chosen and additional information on the schemas is gathered. Subsequently, the schemas are analysed and compared to find similarities/conflicts among the schemas. In the conforming step, the conflicts found in the comparison step have to be resolved. Finally, in the last step, the schemas are merged by superimposition and the resulting schema is analysed and restructured if necessary.

For our purpose, the main characteristic of an integration method is: which similarities/conflicts are detected and how are conflicts resolved? A number of integration methods use assertions among different component schemas to compare attributes and entity types. In [17], interschema assertions, names, and types are used to compare object types. In [15], schemas are merged using schema operators and assertions among entity types and attributes in different schemas. And in [13], attribute assertions (e.g., key/non-key and lower/upper bounds) are used to compare attributes and entity types. However, the assertions must be supplied by the designer and the resolution of conflicts depends heavily on the common sense of the designer. Other methods use schema transformations to resolve structural conflicts. In [9], structural transformations are defined to integrate compatible structures. In [16], a number of

*This research is partly funded by the Dutch Organisation for Scientific Research through NFI-grant NF74.

schema transformations (e.g., join and meet) are proposed to restructure schemas. And in [5], transformations between attributes, entities and relationships are used to resolve type conflicts. However, only the last one gives a heuristic (viz., concept likeness/unlikeness) for applying the transformations. Finally, a number of recent methods use more specific information on semantical properties of attributes and entity types to detect similarities and conflicts. In [18, 23], attribute assertions are used to define relationships between an attribute on one hand and a semantic point or a set of concepts on the other hand. Again, the assertions must be supplied by the designer. In [19], a database metadictionary is used to define a semantic domain for each attribute. And in [10], a terminological knowledge base containing information on negative and positive associations between terms and information on specialisation of terms is used to compare entity types.

This article presents a new approach to schema integration, based on schema transformations and the approach taken in [20, 21], where classes are compared by structure and by behaviour. The approach consist of two steps. First, component schemas are restructured using schema transformations, and syntactical properties of methods are used to guide the restructuring process. Subsequently, the component schemas are merged using join operators, and semantical properties of methods are used to guide the merging process. More details on the approach are given in [22]. There is, as far as the authors know, no other approach that uses methods to compare attributes. For sake of completeness, it should be mentioned that there is an approach to schema evolution that analyses methods ([8]), not to compare attributes, but to solve non-legitimate overriding of methods.

The outline of this article is as follows. In the next section, we give a brief overview and formalisation of our data model. In Section 3, we define a number of well-known type transformations and extend them to be applicable to recursive types as well. Furthermore, we show how these type transformations induce schema transformations. In Section 4, we show how methods can be used to guide schema restructuring and give a heuristic algorithm to restructure and merge schemas. In the last section, we summarise and give some directions for further research.

2 Database schemas

In this section, we introduce a subset of the database schemas found in object-oriented database languages such as Galileo [2], Goblin [12], O_2 [14], and TM/FM [4].

Informally, an object-oriented database schema is a a class hierarchy, i.e., a set of classes related by a subclass relation. A class has a name, a set of superclasses, a set of attributes, a set of constraints, and a set of update and query methods.

Definition 1 (Class hierarchies). First, five disjoint sets are postulated: a set CN of class names, a set AN of attribute names, a set MN of method names, a set L of labels, and a set *Cons* of basic constants (i.e., 'integer', 'rational', and 'string' constants). The sets are generated by the nonterminals CN, AN, MN, L, and Cons, respectively. Class hierarchies are the sentences of the following BNF-grammar, where the plus sign ($^+$) denotes a finite, nonempty repetition, square brackets ([ ]) denote an option, and the vertical bar (|) denotes a choice:

Hierarchy	::=	Class$^+$
Class	::=	'**Class**' CN ['**Isa**' CN$^+$] ['**Attributes**' Att$^+$] ['**Constraints**' Key$^+$] ['**Methods**' Meth$^+$] '**Endclass**'
Att	::=	AN ':' Type
Type	::=	BasicType | SetType | RecordType | CN
BasicType	::=	'integer' | 'rational' | 'string'
SetType	::=	'{' Type '}'
RecordType	::=	'<' FieldList '>'
FieldList	::=	Field | Field ',' FieldList
Field	::=	L ':' Type
Key	::=	**key** KeyAtt$^+$
KeyAtt	::=	AN | KeyAtt '.' L
Meth	::=	MN '(' [ParList] ') =' AsnList | MN '(' [ParList] '→' Result ') =' AsnList |
ParList	::=	Par | Par ',' ParList
Par	::=	L ':' BasicType
Result	::=	L ':' Type
AsnList	::=	Assign | Assign ';' AsnList
Assign	::=	Dest ':=' Source | '**insert**(' Source ',' Dest ')'
Dest	::=	L | AN | L '.' Dest | AN '.' Dest
Source	::=	'**self**' | Term | Term '+' Source | Term '−' Source | Term '×' Source | Term '÷' Source | '**new** (' CN ')' | Dest '.' MN '(' ActParList ')'
Term	::=	Dest | Cons
ActParList	::=	Term | Term ',' ActParList

□

A class hierarchy is well-defined if it satisfies four conditions. The first condition is that the **Isa** relation is acyclic, and classes have a unique name and only refer to classes in the class hierarchy. The second is that attributes have a unique name within their class and are well-typed. The third is that keys must be well-defined. The fourth is that methods have a unique name within their class and are well-typed.

2.1 Underlying types

Informally, the set of all attributes of a class consists of both the new and inherited attributes.

Definition 2 (Attributes). Let H be a class hierarchy satisfying the first condition for well-defined class hierarchies. We abbreviate every class in H to a 5-tuple (c, S, A, K, M), where c is the name of the class, S is the set of (names of) superclasses, A is the set of new attributes, K is the set of new keys, and M is the set of new methods. Now let $C = (c, S, A, K, M)$ be an abbreviated class in H. The name of C is denoted by $name(C)$ and the set of all attributes of C, denoted by $atts(C)$, is defined as:

$$atts(C) = A \cup \{a : T \mid inherits(a) \wedge \\ T = \sqcap\{T' \mid inherits(a, T')\} \wedge \forall a' : T' \in A[a \neq a']\}$$

where

$$inherits(a, T') = \exists C' \in H\ [name(C') \in S \wedge a : T' \in atts(C')],$$
$$inherits(a) = \exists C' \in H\ \exists a' : T' \in atts(C')\ [name(C') \in S \wedge a = a'],$$

and $\sqcap\{T_1, \cdots, T_n\}$ is the meet of a set of types [7]. Since we require that the **Isa** relation is acyclic, *atts* is well-defined. □

Every class in a class hierarchy has an underlying type, which describes the structure of the class, i.e., the structure of the objects in its extensions (cf. TM/FM [4]). The underlying type of a class is an aggregation of its attributes, where recursive types [3] are used to cope with attributes that refer to classes.

Definition 3 (Underlying types). First, postulate a new type 'oid', whose extension is an enumerable set of object identifiers. Let H be a class hierarchy satisfying the first condition, C be a class in H, and c be the name of C. The underlying type of class C, denoted by $type(C)$, is defined as:

$$type(C) = \tau(c, \emptyset)$$

where

$$\tau(d, \eta) = \mu\ t_d\ .\ < \mathrm{id} : \mathrm{oid}, a_1 : \tau(T_1, \eta \cup \{d\}), \cdots, a_k : \tau(T_k, \eta \cup \{d\}) > \\ \text{if } d \notin \eta \text{ and } \exists D \in H[name(D) = d \wedge atts(D) = \{a_1 : T_1, \cdots, a_k : T_k\}],$$
$$\tau(d, \eta) = t_d \text{ if } d \in \eta,$$
$$\tau(B, \eta) = B \text{ if } B \in \{\mathrm{integer}, \mathrm{rational}, \mathrm{string}\},$$
$$\tau(\{U\}, \eta) = \{\tau(U, \eta)\},$$
$$\tau(< l_1 : U_1, \cdots, l_n : U_n >, \eta) = < l_1 : \tau(U_1, \eta), \cdots, l_n : \tau(U_n, \eta) >.$$

The set η contains the names of the classes for which a (recursive) type is being constructed as part of the construction of the underlying type of class C. If η contains d, then $\tau(d, \eta) = t_d$ indicates a repetition of the recursive type. □

Note that the underlying type of a class depends on the hierarchy.

3 Schema transformations

In this section, we give an overview of type transformations and show how type transformations induce schema transformations.

The basic type transformations we have chosen (viz., renaming, aggregation, and objectification) are variants of type transformations in [1]).

Definition 4 (Basic type transformations). Let L' be the union of L and AN and *Types* be the set of types introduced in Definition 2. Renaming is defined as a function of type $L' \rightarrow L' \rightarrow Types \rightarrow Types$:

$rename(l')(l)(B) = B$ if $B \in \{\text{oid}, \text{integer}, \text{rational}, \text{string}\}$
$rename(l')(l)(\{\tau\}) = \{\tau\}$
$rename(l')(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >) = < l_1[l' \setminus l] : \tau_1, \cdots, l_n[l' \setminus l] : \tau_n >$,
$rename(l')(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) = \mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >$
if $l' = \text{id}$,
$rename(l')(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) = \mu t. < l_1[l' \setminus l] : \tau_1, \cdots, l_n[l' \setminus l] : \tau_n >$
if $l' \neq \text{id}$.

Note that we do not allow renaming of id-fields.
Aggregation is defined as a function of type $\wp(L') \rightarrow L' \rightarrow Types \rightarrow Types$:

$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(B) = B$ if $B \in \{\text{oid}, \text{integer}, \text{rational}, \text{string}\}$
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\{\tau\}) = \{\tau\}$
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >) = < l_1 : \tau_1, \cdots, l_n : \tau_n >$
if $\{l_i, l_{i+1}, \cdots, l_j\} \not\subseteq \{l_1, \cdots, l_n\}$,
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >) =$
$< l_1 : \tau_1, \cdots, l :< l_i : \tau_i, \cdots, l_j : \tau_j >, \cdots, l_n : \tau_n >$
if $\{l_i, l_{i+1}, \cdots, l_j\} \subseteq \{l_1, \cdots, l_n\}$,
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) =$
$\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >$
if $\{l_i, l_{i+1}, \cdots, l_j\} \not\subseteq \{l_1, \cdots, l_n\}$,
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) =$
$\mu t. < l_1 : \tau_1, \cdots, l :< l_i : \tau_i, \cdots, l_j : \tau_j >, \cdots, l_n : \tau_n >$
if $\text{id} \notin \{l_i, l_{i+1}, \cdots, l_j\}$ and $\{l_i, l_{i+1}, \cdots, l_j\} \subseteq \{l_1, \cdots, l_n\}$,
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) =$
$\mu s. < l_1 : \tau_1[t \setminus s], \cdots, l : \mu t. < l_i : \tau_i[t \setminus s], \cdots, l_j : \tau_j[t \setminus s] >, \cdots, l_n : \tau_n[t \setminus s] >$
if $\text{id} \in \{l_i, l_{i+1}, \cdots, l_j\}$ and $\{l_i, l_{i+1}, \cdots, l_j\} \subseteq \{l_1, \cdots, l_n\}$.

Objectification is defined as a function of type $Types \rightarrow Types$:

$objectify(B) = B$ if $B \in \{\text{oid}, \text{integer}, \text{rational}, \text{string}\}$
$objectify(\{\tau\}) = \{\tau\}$
$objectify(< l_1 : \tau_1, \cdots, l_n : \tau_n >) = \mu s. < \text{id} : \text{oid}, l_1 : \tau_1, \cdots, l_n : \tau_n >$,
$objectify(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) = \mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >$
if $\exists i \in \{1, \cdots, n\}[l_i = \text{id}]$,
$objectify(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) = \mu t. < \text{id} : \text{oid}, l_1 : \tau_1, \cdots, l_n : \tau_n >$
if $\forall i \in \{1, \cdots, n\}[l_i \neq \text{id}]$.

□

Complex type transformations are obtained by combining basic type transformations.

Example 1. Type $\sigma = < l_1 : \mu s. < \text{id} : \text{oid}, l : \tau_1, l_2 : \tau_2 >, l_3 : \tau_3 >$ can be obtained from type $\sigma_1 = < l_1 : \tau_1, l_2 : \tau_2, l_3 : \tau_3 >$ as follows:

$\sigma_2 = rename(l_1)(l)(\sigma_1) = < l : \tau_1, l_2 : \tau_2, l_3 : \tau_3 >$
$\sigma_3 = aggregate(\{l, l_2\})(l_1)(\sigma_2) = < l_1 : < l : \tau_1, l_2 : \tau_2 >, l_3 : \tau_3 >$
$\sigma_4 = < l_1 : objectify(< l : \tau_1, l_2 : \tau_2 >), l_3 : \tau_3 > = \sigma$.

□

Both the transformation for lexical attributes and the transformation for unstable subtypes from [11] can be obtained by composing one aggregate and one objectify operation.

Example 2. The following class hierarchy introduces a class Person, a class Employee, which inherits from class Person, and a class Company:

Class Person
Attributes
name : string
street : string
house : integer
city : string
Endclass

Class Employee **Isa** Person
Attributes
employer : Company
salary : integer
Endclass
Class Company
Attributes
name : string
Endclass.

The underlying type of class Person is:

$\mu\, t_P$. <id:oid, name:string, street:string, house:integer, city:string>.

The underlying type of class Person can be transformed into (by applying *aggregate* ({street,house,city}) (address)):

$\mu\, t_P$. <id:oid, name:string, address:<street:string, house:integer, city:string>>,

which can be transformed into (by applying *objectify* to the type of address):

$\mu\, t_P$. <id:oid, name:string,
address:$\mu\, t_X$. <id: oid, street:string, house:integer, city:string>>.

The composite transformation is a variant of the transformation for lexical attributes from [11]. We can redefine class Person as a class (named Person1) that refers to a new class (named X).

Class Person1
Attributes
name : string
address : X
Endclass

Class X
Attributes
street : string
house : integer
city : string
Endclass.

The underlying type of class Employee is:

$\mu\, t_E$. <id:oid, name:string, street:string, house:integer, city:string,
employer:τ_C, salary:integer>,

where τ_C is the underlying type of class Company. The underlying type of class Employee can be transformed into (by applying *aggregate* ({id,name,street,house,city}) (employee)):

$\mu\, t_Y$. <employee:$\mu\, t_E$. <id:oid, name:string, street:string, house:integer,
city:string>, employer:τ_C, salary:integer>,

which can be transformed into (by applying *objectify*):

$\mu\, t_Y$. <id:oid, employee:$\mu\, t_E$. <id:oid, name:string, street:string, house:integer,
city:string>, employer:τ_C, salary:integer>.

The composite transformation is a variant of the transformation for unstable subtypes from [11]. We can redefine class Employee as a 'relation' (named Y) that refers to a new class (named Employee1):

Class Y
Attributes
employee : Employee1
employer : Company
salary : integer
Endclass

Class Employee1
Attributes
name : string
street : string
house : integer
city : string
Endclass.

Note that, in the original situation, an employee (an object in class Employee), does have a unique employer, whereas, in the resulting situation, an employee (an object in class Employee1) does not. Therefore, we define a key for class Y:

Class Y1
Attributes
employee : Employee1
employer : Company
salary : integer
Constraints
key employee
Endclass.

□

4 Application of schema transformations

In the previous section, we defined type transformations and showed how they induce schema transformations. In this section, we show how behaviour of methods can be used to choose among a set of schema transformations.
A class can be transformed in several ways, using different factors and different transformations.

Example 3. Let class Employee be the following class:

Class Employee
Attributes
name : string
dob : Date
street : string
house : integer
city : string
employer : Company
Methods
move (s:string,h:integer,c:string) =
street := s; house := h; city := c
Endclass

and class Address be a factor of Employee:

Class Address
Attributes
street : string
house : integer
city : string
Methods
move (s:string,h:integer,c:string) =
street := s; house := h; city := c
Endclass.

One option to transform class Employee is to redefine Employee as a subclass of Address (factorisation by specialisation):

Class Employee1 **Isa** Address
Attributes
name : string
dob : Date
employer: Company
Endclass.

Another option is to redefine Employee as a class referring to Address (factorisation by delegation):

Class Employee2
Attributes
name : string
dob : Date
address: Address2
employer: Company
Methods
move (s:string,h:integer,c:string) =
address := address.new_address(s,h,c)
Endclass
Class Address2
Attributes
street : string
house : integer
city : string
Methods
new_address (s:string,h:integer,c:string → l:Address2) =
l := **new**(Address2) ; l.street := s ; l.house := h ; l.city := c
Endclass.

Note that, as an employee is not an address in the real world, it is unlikely that the first option is the right choice. The second option, where employee refers to an address (as one of its attributes) is a more reasonable choice. Now, let class Person be a factor of class Employee2:

Class Person
Attributes

```
    name : string
    dob : Date
    address : Address2
Methods
    move (s:string,h:integer,c:string) =
        address := address.new_address(s,h,c)
Endclass.
```

One option to transform class Employee2 is to redefine Employee2 as a subclass of Person (factorisation by specialisation):

```
Class Employee3 Isa Person
Attributes
    employer : Company
Endclass.
```

Another option is to redefine Employee2 as a class referring to Person (factorisation by delegation):

```
Class Employee4
Attributes
    person : Person1
    employer : Company
Methods
    move (s:string,h:integer,c:string) =
        person := person.new_person(s,h,c)
Endclass
Class Person1
Attributes
    name : string
    dob : Date
    address : Address2
Methods
    new_person (s:string,h:integer,c:string → l:Person1)
        l := new(Person1) ; l.name := name ;
        l.dob := dob ; l.address := l.address.new_address(s,h,c)
Endclass.
```

Since the objects in class Employee2 become the objects in class Employee4, we redefine method 'move' to be applicable to objects in class Employee4. Yet another option is to redefine class Employee2 as a relation involving class Person:

```
Class Employment
Attributes
    employee : Person
    employer : Company
Constraints
    key employee
Endclass.
```

Since the objects in class Employee2 become the objects in class Person, we do not redefine method 'move', because it is already applicable to objects in class Person. Note that, as an employee is a person in the real world, it is likely that options one and three are more reasonable than option two, where an employer refers to a person (as one of its attributes). □

As we have seen, a class can be transformed in several ways, using different factors and different transformations, e.g., factorisation by specialisation, factorisation by delegation, or redefinition as a relation. But how do we choose factors and how do we choose between specialisation, delegation and redefinition as a relation? For that purpose, we introduce evidence ratios for relatedness. Weak relatedness for a set of attributes says whether the attributes are mutually related (according to the methods). Strong relatedness for a set of attributes says whether the attributes are mutually related, but not to attributes outside the set (according to the methods). Isolation for a set of attributes says whether the attributes are not related to attributes outside the set (according to the methods).

Definition 5 (Relatedness ratios). Let H be a well-defined class hierarchy, C be a class in H, c be the name of C, and M be the set of all methods of C. Furthermore, for $meth \in M$, let $atts(meth)$ consist of the names of attributes of C that occur in $meth$. Weak relatedness of a set of attributes $A \subseteq \{a \mid a : T \in atts(C)\}$ is defined as:

$$weakrel(c, A) = \frac{|\{meth \in M \mid atts(meth) \supseteq A\}|}{|\{meth \in M \mid atts(meth) \cap A \neq \emptyset\}|}.$$

Strong relatedness of a set of attributes A is defined as:

$$strongrel(c, A) = \frac{|\{meth \in M \mid atts(meth) = A\}|}{|\{meth \in M \mid atts(meth) \cap A \neq \emptyset\}|}.$$

Isolation of a set of attributes $A \subseteq \{a \mid a : T \in atts(C)\}$ is defined as:

$$isolation(c, A) = \frac{|\{meth \in M \mid \emptyset \neq atts(meth) \subseteq A\}|}{|\{meth \in M \mid atts(meth) \cap A \neq \emptyset\}|}.$$

If $\{meth \in M \mid atts(meth) \cap A \neq \emptyset\}$ is empty, then $weakrel(c, A)$ and $strongrel(c, A)$ are defined to be 0, and $isolation(c, A)$ is defined to be 1.

For a set of attributes with strong relatedness ratio 1 and any method, either all attributes occur in the method and all attributes that occur in the method are in the set, or no attribute in the set occurs in the method. In that case, the attributes are strongly related. For a set of attributes with weak relatedness ratio 0, there is no method in which all attributes occur and, hence, the attributes are not (mutually) related. And for a set of attributes with isolation ratio 1 and any method, either all attributes that occur in the method are attributes in the set or no attribute that occurs in the method is an attribute in the set. In that case, the attributes are only related within the set. □

Weak and strong relatedness can help to choose a factor. If the strong relatedness ratio of a set of attributes is high, then it is reasonable to believe that they belong together and, hence, to factorise. On the other hand, if the weak relatedness ratio is low, then it reasonable to believe that they do not belong together and, hence, not to factorise.

Example 4. Consider class Employee of Example 3. The weak and strong relatedness ratios for {street, house, city} and {name, dob} are given by:

strongrel(Employee, {street, house, city}) = 1
weakrel(Employee, {street, house, city}) = 1
strongrel(Employee, {name, dob}) = 0
weakrel(Employee, {name, dob}) = 0.

As we can see, street, house, and city are strongly related, whereas name and dob are not related.

Now, consider class Employee2 of Example 3. The weak and strong relatedness ratios for {name, dob, address} and {name, dob, employer} are given by:

strongrel(Employee2, {name, dob, address}) = 0
weakrel(Employee2, {name, dob, address}) = 0
strongrel(Employee2, {name, dob, employer}) = 0
weakrel(Employee2, {name, dob, employer}) = 0.

As we can see, in both cases the attributes are not related. □

Isolation can help to choose between specialisation and redefinition as a relation. If the isolation ratio is less than one, then specialisation is possible, but redefinition as a relation is not, since, in that case, we have to add a method to the relation that updates another relation or class.

Example 5. Consider class Employee2 of Example 3. The isolation evidence ratio for name, dob, address is given by:

isolation(Employee2, {name, dob, address}) = 1.

Redefinition as a relation results in a relation (Employment) that represents a simple association between a person and a company. Now, if we add a method to class Employee2 that updates attribute address and attribute employer, then we will have to add a method to Employment that creates a new person and updates attribute employee and attribute employer. Since this method is not a simple insert or update operation on Employment, Employment is no longer a relation. □

So, how do we choose factors and transformations? Factors are chosen by comparing weak evidence ratios. If the weak evidence ratio of a set of attributes is greater than some threshold, there is reason to assume that the attributes can be used as a factor. If not, there is no reason. Transformations are chosen by comparing strong evidence ratios and isolation ratios. In case the strong evidence ratio is greater than some threshold, delegation is a reasonable option, because the attributes are strongly related within the set and weakly related with other attributes. In case the isolation ratio is less than one, then specialisation is possible, but redefinition as a relation is not. Otherwise, specialisation or redefinition as a relation are both possible. It should be mentioned that, in the context of schema integration, schema transformations must be applied carefully and only if necessary. In particular, this is true for factorisation by specialisation, since a lot of new classes will be generated by this type of transformation.

The considerations for choosing factors and transformations can be used in a heuristic algorithm to support schema integration. First, the attributes of every class are partitioned in such a way that the isolation ratio of every element in the partition is one, and every class is factorised by delegation if desirable. Subsequently, for every

pair of promising classes, a set of possible superclasses is computed, and both classes are factorised by specialisation or redefined as a relation if desirable.

Algorithm 1. The following algorithm is a heuristic for integrating two database schemas (resp., DBS1 and DBS2), given thresholds for strong relatedness and weak relatedness (resp., TSR and TWR):

```
integrate(DBS1,DBS2,TSR,TWR) =
   for every class C in DBS1 or DBS2
   do   for every element A in partition(C)
        do   if strongrel(c,A) ≥ TSR and 1 < |A| < |atts(C)|
             then create class C1 as the class containing A
                  and the methods that refer to A;
                  factorise C by delegation using C1;
                  mark C and C1
             elif weakrel(c,A) ≥ TWR
             then mark C
             fi
        od
   od
   for every marked C1 in DBS1
   do   for every marked C2 in DBS2
        do   if there is a superclass C of a class in joins(C1,C2) that can be used
             as a factor according to the designer
             then transform(C1,C2,C)
             else  for every key a.p in keys(C1) such that a:D in atts(C1)
                   for some class D
                   do define class D1 as obtained from C1 by applying
                        the inverse of redefinition as a relation;
                        if there is a superclass C of a class in joins(D1,C2) that
                        can be used as a factor according to the designer
                        then transform(D1,C2,C)
                        fi
                   od
             fi
        od
   od;
transform(C1,C2,C) =
   begin let φ1 be an injection from atts(C) to atts(C1) induced by C1 ⪯ C;
        let φ2 be an injection from atts(C) to atts(C2) induced by C2 ⪯ C;
        define A1 as the attribute names in the range of φ1;
        define A2 as the attribute names in the range of φ1;
        if isolation(name(C1),A1) < 1 or isolation(name(C2),A2) < 1
        then factorise C1 and C2 by specialisation using C
        elif 1 < |A1| < |atts(C1)| and 1 < |A2| < |atts(C2)|
        then factorise C1 and C2 or redefine C1 and C2 as relations
             according to the choice of the designer
        else factorise C1 and C2 by specialisation using C
```

fi
end;

where partition(C) is constructed as follows:

graph(C) has a node for every attribute name in atts(C)
graph(C) has an edge between two nodes if there is a method in the set of
all methods of C in which both attribute names occur
partition(C) consists of sets of attribute names,
one set for every connected subgraph of graph(C):
two attribute names are in the same set if their nodes are connected
two attribute names are in different sets if their nodes are not connected,

and joins(D1,D2) (i.e., the set of common superclasses of D1 and D2) and $\preceq$ (i.e., the subclass relation) are as defined in [21]. □

Note that the algorithm interacts with the designer. It should be mentioned again that the algorithm is a heuristic and should therefore be used in close interaction with the designer. The heuristic can be improved by combining the different thresholds and refining the different actions. This is the subject of future research.

5 Conclusion

In this article, we presented a new approach to schema integration based on transformations and behaviour. First, we formalised schemas using underlying types and underlying constraints. Next, we presented a number of type transformations on underlying types and used them to transform schemas. Finally, we gave a heuristic algorithm for integrating schemas. The algorithm uses schema transformations to restructure schemas and join operators to merge them and behavioural information to guide restructuring and merging. Advantages of this approach are: 1) structural aspects are integrated in a guided fashion and 2) both structural and behavioural aspects are integrated.

Further research includes 1) extension of the data model, 2) extension of the schema transformations, and 3) extension and refinement of the heuristic algorithm.

Bibliography

[1] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62:3–38, 1988.

[2] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Trans. on Database Systems*, 10(2):230–260, 1985.

[3] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. Int. Symp. on Principles of Programming Languages*, pages 104–118, 1991.

[4] P. Apers, H. Balsters, R. de By, and C. de Vreeze. Inheritance in an object-oriented data model. Memoranda Informatica 90-77, University of Twente, Enschede, The Netherlands, 1990.

[5] C. Batini and M. Lenzerini. A methodology for data schema integration in the ER model. *IEEE Transactions on Software Engineering*, pages 650–664, November 1984.

[6] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[7] L. Cardelli. A semantics of multiple inheritance. In *Proc. Int. Symp. on Semantics of Datatypes, LNCS 173*, pages 51–67. Springer-Verlag, Berlin, 1984.

[8] E. Casais. An incremental class reorganization approach. In *European Conf. on Object-Oriented Programming*, pages 114–132, 1992.

[9] R. Elmasri and G. Wiederhold. Data model integration using the structural model. In *Proc. Int. Conf. on Management of Data*, pages 191–202, 1979.

[10] P. Fankhauser, M. Kracker, and E. Neuhold. Semantic vs. structural resemblance of classes. *ACM SIGMOD Record*, 20(4):59–63, 1991.

[11] P. Johannesson. Schema transformations as an aid in view integration. In *Proc. Int. Conf. on Advanced Information Systems Engineering, LNCS 685*, pages 71–92. Springer-Verlag, Berlin, 1993.

[12] M. Kersten. Goblin: a DBPL designed for advanced database applications. In *Proc. Int. Conf. on Database and Expert Systems Applications*, pages 345–349. Springer-Verlag, Wien, 1991.

[13] J. Larson, S. Navathe, and R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, 1989.

[14] C. Lécluse and P. Richard. The O_2 database programming language. In *Proc. Int. Conf. on Very Large Databases*, pages 411–422. Morgan Kaufmann, Palo Alto, CA, 1989.

[15] M. Mannino, S. Navathe, and W. Effelsberg. A rule based approach for merging generalisation hierarchies. *Information Systems*, 13(3):257–272, 1988.

[16] A. Motro and P. Buneman. Constructing superviews. In *Proc. Int. Conf. on Management of Data*, pages 56–64, 1981.

[17] S. Navathe and S.Gadgil. A methodology for view integration in logical data base design. In *Proc. Int. Conf. on Very Large Databases*, pages 142–155, 1982.

[18] A. Sheth and S. Gala. Attribute relationships: an impediment in automating schema integration. In *Proc. Workshop on Heterogeneous Database Systems*, 1989.

[19] M. Siegel and S. Madnick. A metadata approach to resolving semantic conflicts. In *Proc. International Conference on Very Large Databases*, pages 133–145, 1991.

[20] C. Thieme and A. Siebes. Schema integration in object-oriented databases. In *Proc. Int. Conf. on Advanced Information Systems Engineering, LNCS 685*, pages 54–70. Springer-Verlag, Berlin, 1993.

[21] C. Thieme and A. Siebes. Schema refinement and schema integration in object-oriented databases. In *Proc. Computing Science in The Netherlands, ISBN 90 6196 430 X*, pages 343–354. Stichting Mathematisch Centrum, 1993.

[22] C. Thieme and A. Siebes. An approach to schema integration based on transformations and behaviour. Report CS-R9403, CWI, Amsterdam, The Netherlands, 1994 (available by anonymous ftp from ftp.cwi.nl).

[23] C. Yu, W. Sun, S. Dao, and D. Keirsey. Determining relationships among attributes for interoperability of multi-database systems. In *Proc. Int. Workshop on Interoperability in Multidatabase Systems*, pages 251–257, 1991.