Schema Integration in Object-Oriented Databases*

Christiaan Thieme and Arno Siebes

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
{ct,arno}@cwi.nl

Abstract

This paper presents a formal approach to support schema integration in object-oriented databases. The basis of the approach is a subclass order, which is defined in terms of a weak subtype relation on underlying types of classes and a subfunction relation on functional forms of methods. The subclass order induces an equivalence relation and a join operator, which are used to identify and factorise class hierarchies, leading to a natural framework for integration of class hierarchies. The novelty of this paper is that both attributes and methods are used to compare classes, and that behaviour of methods is used to compare attributes, resulting in a more semantic approach towards schema integration in object-oriented databases.

1 Introduction

Database design is a complex, iterative process consisting of several activities, including conceptual design and implementation design [16]. Conceptual design concerns itself with the description of diverse users' information requirements and the integration of these requirements into a DBMS-independent database schema. Implementation design uses the results of the conceptual design phase and the processing requirements as input to produce a DBMS-processible database schema. Due to its complexity, database design is an error-prone process. Therefore, it has to be structured by a design methodology [15], which includes guidelines, techniques, methods, and tools to support the activities of the designer.

This paper addresses the problem of identifying and factorising classes in class hierarchies, which form the static part of object-oriented database schemas [2, 14, 3, 12]. A solution to this problem can be used to support integration of different user views in the conceptual design phase of an object-oriented design methodology [11], schema normalisation in the implementation design phase, or schema integration in general, e.g., in multidatabase systems.

An overview of methods for schema integration in relational and semantic databases can be found in [5]. These methods intend to integrate entities and relationships that

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

represent the same concept in the application domain. First, naming conflicts, such as homonyms and synonyms, and structural conflicts such as type inconsistencies, integrity constraint conflicts, redundancy conflicts, and differences in abstraction levels are investigated. Subsequently, the conflicts are resolved by renaming, type transformations, restriction, redundancy elimination, and aggregation. Some of the methods create generalisation hierarchies to combine entities. Finally, entities and relationships are merged.

Normalisation of class hierarchies is the subject of [6], in which Bergstein and Lieberherr give an algorithm for the construction of class hierarchies from examples and the optimisation of the resulting class hierarchies by reducing the number of attributes and subclass relationships. However, normalisation of classes is restricted in two ways. Firstly, only attributes are considered, methods are ignored. Secondly, attributes are compared by name and type only (i.e., syntactic), not by meaning (i.e., semantic). In [9], Fankhauser, Kracker, and Neuhold present an approach to determine the semantic similarity of classes using probabilistic knowledge on terminological relationships between classes.

This paper presents a formal approach to integrate class hierarchies on the basis of syntactic and semantic similarity of classes. First, classes are described in terms of types and functions, based on Cardelli's work on subtyping [7]. Subsequently, a synthetic subclass order is introduced, which induces an equivalence relation on classes. The subclass order requires attributes to have equivalent types and to play the same roles in the methods, and methods to be extensionally equal. This leads to a natural framework for integration of class hierarchies, where classes are identified using the equivalence relation and factorised using a join operator w.r.t. the subclass order.

2 Class Hierarchies

In this section, we introduce class hierarchies, similar to class hierarchies in Galileo [2], Goblin [12], O_2 [14], and TM/FM [3], and show how to flatten class hierarchies.

Informally, a class hierarchy is a set of classes. A class has a name, a set of superclasses, a set of attributes, and a set of methods. An attribute has a name and a type, which can be a basic, set, or record type, or a class. Hence, classes can be recursive. An update method has a name, a list of parameters and a body which consists of simple assignments. The formal syntax definition of class hierarchies can be found in Appendix A.

A class hierarchy is well-defined if it satisfies two constraints. The first constraint is that classes have a unique name and only refer to classes in the class hierarchy. The second constraint is that attributes and methods (inherited attributes and methods included) have a unique name within their class and methods are well-typed. The formal definition of well-typed methods can be found in Appendix B.

Example 1. The following well-defined class hierarchy introduces a class 'Person', which is recursive, a class 'Employee', which is a subclass of 'Person', and a class 'Date':

Class Person
Attributes
name : string

dob : Date
mother : Person
Methods
change (s:string) = name := s
Endclass
Class Employee **Isa** Person
Attributes
salary : int
company : string
Methods
increase (i:int) = salary := salary+i
Endclass
Class Date
Attributes
day : int
month : int
year : int
Endclass.

□

The attributes and methods of class 'Person' are also attributes and methods of class 'Employee'. This leads to the notion of flattened class hierarchies.

Let H be a class hierarchy and C be a class in H. The flattened form of C, denoted by $\mathit{flat}(C)$, is obtained by accumulating the attributes and methods of all superclasses of C.

Example 2. Let C_e be class 'Employee' of Example 1. The flattened form of C_e is given by:

Class Employee
Attributes
name : string
dob : Date
mother : Person
salary : int
company : string
Methods
change (s:string) = name := s
increase (i:int) = salary := salary+i
Endclass

or, more concisely, by:

(Employee,
{name:string, dob:Date, mother:Person, salary:int, company:string},
{change (s:string) = name := s, increase (i:int) = salary := salary+i}).

□

The flattened class hierarchy corresponding to H, denoted by $\mathit{flat}(H)$, is obtained by replacing every class by its flattened form:

$$\mathit{flat}(H) = \{\mathit{flat}(C) \mid C \in H\}.$$

Note that the subclass (**Isa**) relation is not explicitly preserved by flattening. In Section 4, we will define a synthetic subclass relation on flattened classes.

3 Types and Functions

In this section, we describe classes in terms of types and functions, similar to TM/FM [4, 8], which is based on Cardelli's work on subtyping [7]. First, we introduce the underlying type of a class, define structural equality for underlying types, and briefly mention an order on underlying types. Subsequently, we introduce the functional form of a method and define extensional equality for functional forms. Underlying types and functional forms will be used to define class equivalence.

3.1 Underlying Types

Every class in a well-defined class hierarchy corresponds to an underlying type and a set of functional forms, one for each of its methods. The underlying type describes the structure of the class (i.e., the structure of the objects in its extensions). Let H be a well-defined class hierarchy and C be a class in $\mathit{flat}(H)$. Furthermore, let c be the name and $\{a_1 : T_1, \cdots, a_k : T_k\}$ be the attributes of class C. The underlying type of C, denoted by $\mathit{type}(C)$, is defined as:

$$\mathit{type}(C) = < a_1 : \mathit{type}(T_1), \cdots, a_k : \mathit{type}(T_k) >,$$

where

$$\begin{array}{ll} \mathit{type}(d) = \uparrow \mathit{type}(D) & \text{if } D = (d, A, M) \in \mathit{flat}(H), \\ \mathit{type}(T) = T & \text{if } T \in \{\text{bool}, \text{int}, \text{string}\}, \\ \mathit{type}(\{U\}) = \{\mathit{type}(U)\}, & \\ \mathit{type}(< U_1 : T_1, \cdots, l_n : U_n >) = < l_1 : \mathit{type}(U_1), \cdots, l_n : \mathit{type}(U_n) >. & \end{array}$$

That is, the underlying type of a class is an aggregation of its attributes, where pointer types are used to cope with attributes that refer to classes. Note that, if a class is recursive, then its underlying type is recursive [13].

Example 3. Let C_p be class 'Person' of Example 1. The underlying type of $\mathit{flat}(C_p)$ is given by:

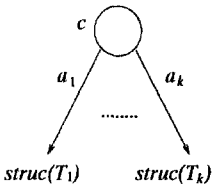
$$\tau_p = <\text{name:string, dob:}\uparrow<\text{day:int,month:int,year:int}>\text{, mother:}\uparrow \tau_p >,$$

which is a recursive type. □

A natural notion of type equivalence is structural equality [1]. Two types are structurally equal if they are either the same basic type or are formed by applying the same constructor to structurally equal types. Algorithms for testing structural equality of recursive types can, e.g., be found in [1] and [13]. In [13], (infinite) trees are used to represent (recursive) types and structural equality of types is defined in terms of tree

equality. In Section 5, class equivalence will be defined in terms of structural equality of classes and method equivalence. Two classes are structurally equal if their underlying types are structurally equal. Hence, the algorithm from [13] can be used to determine structural equality of classes, using trees to represent their underlying types.

Class equivalence will be defined using a weaker form of structural equality of classes, where an attribute can be mapped to an attribute with a different name. For that purpose, the tree representing the underlying type of a class is adapted slightly. The adapted tree is obtained by removing pointers and labeling class nodes by the name of the corresponding class. Let H be a well-defined class hierarchy and C be a class in $\mathit{flat}(H)$. Furthermore, let c be the name and $\{a_1 : T_1, \cdots, a_k : T_k\}$ be the attributes of class C. The tree representing the structure of class C, denoted by $\mathit{struc}(C)$, is defined as:

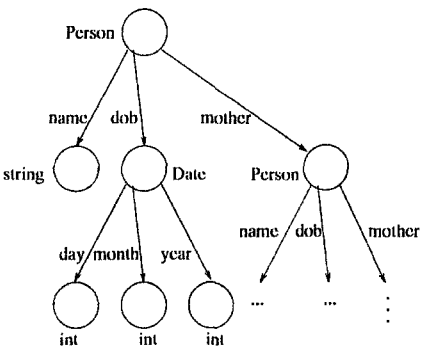


where

$\mathit{struc}(d) = \mathit{struc}(D)$ if $D = (d, A, M) \in \mathit{flat}(H)$,
$\mathit{struc}(T)$ has only one node, labeled T if $T \in \{\text{bool}, \text{int}, \text{string}\}$,
$\mathit{struc}(\{U\})$ consists of a root, labeled by $\{\}$, a subtree $\mathit{struc}(U)$,
and an unlabeled arrow from the root labeled by $\{\}$
to the root of $\mathit{struc}(U)$
$\mathit{struc}(< l_1 : U_1, \cdots, l_n : U_n >)$ consists of a root, labeled by $<>$,
subtrees $\mathit{struc}(U_1), \cdots, \mathit{struc}(U_n)$, and arrows, labeled l_i,
one for each $i \in \{1, \cdots, n\}$, from the root labeled by $<>$
to the root of $\mathit{struc}(U_i)$.

Note that the tree representing the structure of a class depends on the hierarchy. Furthermore, observe that the tree representing the structure of a recursive class is infinite.

Example 4. Let C_p be class 'Person' of Example 1. The tree representing the structure of class $\mathit{flat}(C_p)$ is given by the following infinite tree:



□

In [7], a natural notion of subtyping is defined for record types: τ_1 is a subtype of type τ_2 if τ_1 has at least the fields of τ_2. For example, <name:string,dob:Date> $\leq$ <name:string>, where $\leq$ denotes the subtype relation. This notion of subtyping can be extended to underlying types of classes:

$$type(C_1) \leq type(C_2) \Leftrightarrow struc^*(C_2) \text{ is a subtree of } struc^*(C_1),$$
$$\text{such that the root of the subtree is also the root of } struc^*(C_1).$$

where $struc^*(C_i)$ is $struc(C_i)$ with class names removed. The subtype relation can be used to define structural equality of types, and, hence, structural equality of classes:

$$C_1 =_{struc} C_2 \Leftrightarrow type(C_1) =_{struc} type(C_2)$$
$$\Leftrightarrow type(C_1) \leq type(C_2) \wedge type(C_2) \leq type(C_1).$$

Although structural equality is an equivalence relation on classes, it is too coarse, because attributes are considered only. In order to define a finer equivalence relation on classes, methods have to be considered as well.

3.2 Functional Forms

Every well-typed method has a functional form. The functional form of a method in a class describes the way in which objects in the extensions of the class are manipulated by the method. Let H be a well-defined class hierarchy and C be a class in $flat(H)$. Furthermore, let $m(P) = E$ be a method in class C. The functional form of $m(P) = E$ in class C, denoted by $func(C, m)$, is defined as:

$$func(C, m) = \lambda self : type(C) \lambda P. < a_1 = e_1, \cdots, a_k = e_k >,$$

where $\{a_1 : T_1, \cdots, a_k : T_k\}$ are the attributes of C and expression e_i is the accumulated effect of the assignments in E on attribute a_i. The type of $body(C, m)$ is $type(C)$, as should be the case, since $m(P) = E$ is an update method. The formal definition of functional forms can be found in Appendix C.

Example 5. Let C_p be class 'Person' and C_e be class 'Employee' of Example 1. The functional forms of method 'change' in class $flat(C_p)$ and method 'change' in class $flat(C_e)$ are given by:

$func(flat(C_p)$,change$) = \lambda self : \tau_p \lambda$s : string.
<name=s, dob=*self*.dob,mother=*self*.mother>,
$func(flat(C_e)$,change$) = \lambda self : \tau_e \lambda$s : string.
<name=s, dob=*self*.dob, mother=*self*.mother,
salary=*self*.salary, company=*self*.company>,

where τ_p is the underlying type of class $flat(C_p)$ and τ_e is the underlying type of class $flat(C_e)$. □

A natural notion of function equivalence is (extensional) equality. Two functions are (extensionally) equal if they map the same combination of input values to the same output value.

Example 6. The following functions are extensionally equal:

$$\lambda self : \tau_p \lambda \mathrm{i} : \text{int}.\ (2 \times \mathrm{i} + self.\text{dob.year}) =$$
$$\lambda self : \tau_p \lambda \mathrm{i} : \text{int}.\ (\mathrm{i} + \mathrm{i} + self.\text{dob.year}),$$

where τ_p is the underlying type of Example 3. □

Two functional forms are extensionally equal if they are pairwise equal for every attribute:

$$\lambda self : \tau \lambda P. < a_1 = e_1, \cdots, a_n = e_n > =$$
$$\lambda self : \tau \lambda P'. < a_1 = e'_1, \cdots, a_n = e'_n > \Leftrightarrow$$
$$\forall i \in \{1, \cdots, n\}\ [\lambda self : \tau \lambda P.e_i = \lambda self : \tau \lambda P'.e'_i].$$

For our language, allowing disjunction and conjunction for booleans, addition and multiplication for integers, concatenation for strings, and insertion for sets, extensional equality of functional forms is decidable [17].

4 Comparison of Classes

In this section, we introduce a synthetic subclass order to compare classes on the basis of syntactic and semantic similarity. The subclass order is defined in terms of a weak subtype relation on underlying types of classes (using graph homomorphisms between adapted trees) and a subfunction relation on functional forms of methods (using extensional equality for functional forms). In the following section, a join operator w.r.t. the subclass order will be defined to factorise classes.

In order to define the subclass order, a number of properties are introduced for graph homomorphisms. A graph homomorphism φ from graph G_1 to graph G_2 is said to preserve labels if:

1. $(n \in \mathit{nodes}(G_1) \wedge \mathit{label}(n) = l) \Rightarrow \mathit{label}(\varphi(n)) = l$
2. $(p \in \mathit{arrows}(G_1) \wedge \mathit{label}(p) = l) \Rightarrow \mathit{label}(\varphi(p)) = l,$

where $\mathit{nodes}(G_1)$ and $\mathit{arrows}(G_1)$ denote the set of nodes of G_1 and the set of arrows of G_1, respectively, and $\mathit{label}(q)$ denotes the label of node or arrow q. And, for this paper, a graph homomorphism φ from tree G_1 to tree G_2 is a tree homomorphism if it maps the root of G_1 to the root of G_2. For example, the identity graph homomorphism from a tree onto itself is a tree homomorphism.

Now let H be a well-defined class hierarchy. Let C_1 and C_2 be classes in $\mathit{flat}(H)$. The following two properties will be used to relate an attribute $a_1 : T_1$ in C_1 to an attribute $A_2 : T_2$ in C_2, such that $\mathit{type}(T_2)$ is a weak subtype of $\mathit{type}(T_1)$. A graph homomorphism φ from $\mathit{struc}(C_1)$ to $\mathit{struc}(C_2)$ is faithful with respect to classes if it maps classes to classes, i.e., if for every node n in $\mathit{struc}(C_1)$, the following holds:

$$\mathit{label}(n) \in CN \Rightarrow \mathit{label}(\varphi(n)) \in CN.$$

Example 8. Let C_p be class 'Person' and C_e be class 'Employee' of Example 1. Any graph homomorphism from $\mathit{struc}(\mathit{flat}(C_p))$ to $\mathit{struc}(\mathit{flat}(C_e))$ that maps nodes labeled 'Person' to nodes labeled 'Employee' or 'Person' and nodes labeled 'Date' to nodes labeled 'Date' is faithful w.r.t. classes. □

A graph homomorphism φ from $struc(C_1)$ to $struc(C_2)$ is faithful with respect to attributes if it maps attributes in one class to attributes in another class consistently, i.e., if for every node n_1 in $struc(C_1)$ with outgoing arrow p_1 and every node n_2 in $struc(C_1)$ with outgoing arrow p_2, the following holds:

$$(label(n_1) = label(n_2) \in CN \wedge label(p_1) = label(p_2) \wedge \\ label(\varphi(n_1)) = label(\varphi(n_2))) \Rightarrow label(\varphi(p_1)) = label(\varphi(p_2)).$$

Example 9. Let C_p be class 'Person' and C_e be class 'Employee' of Example 1. The tree homomorphism φ_2 from $struc(flat(C_p))$ to $struc(flat(C_e))$ that preserves labels, except the label of the root, is faithful w.r.t. attributes. If all arrows labeled 'day' are mapped to arrows labeled 'month', then the graph homomorphism is still faithful w.r.t. attributes. However, if only one of the arrows labeled 'day' is mapped to an arrow labeled 'month', then the graph homomorphism is not faithful any more. □

The following property will be used to relate a method $m_1(P_1) = E_1$ in C_1 to a method $m_2(P_2) = E_2$ in C_2, such that $func(C_2, m_2)$ is a subfunction of $func(C_1, m_1)$.

Let φ be an injective graph homomorphism from $struc(C_1)$ to $struc(C_2)$ that preserves labels, except class names and attribute names, and is faithful w.r.t. to classes and faithful w.r.t. attributes. If such a graph homomorphism exists, then we say that $type(C_2)$ is a weak subtype of $type(C_1)$.

Let n be a node in $struc(C_1)$, such that $label(n)$ is the name of class C and $label(\varphi(n))$ is the name of class $C' = (c', A', M')$ in $flat(H)$. For every method $m(P) = E$ in C, define $func(C, m)[\varphi]$ to be the functional form of its imaginary counterpart in C':

$$func(C, m)[\varphi] = func(C'_m, m),$$

where

$$C'_m = (c'_m, A', \{m(P) = E[\varphi]\}),$$

where c'_m is a unique class name depending on c' and m, and $E[\varphi]$ is obtained from E as follows: if $r_1 \cdots r_p$ is a path in $struc(C_1)$ starting at node n, then every occurrence of term $label(r_1).\cdots.label(r_p)$ in E is replaced by $label(\varphi(r_1)).\cdots.label(\varphi(r_p))$.

If $m(P) = E$ is a method in C, $m'(P') = E'$ is a method in C', and $func(C, m)[\varphi] = func(C', m')$, then we say that $func(C', m')$ is a subfunction of $func(C, m)$, because $type(C')$ is a weak subtype of $type(C)$ and $body(C', m')$ manipulates objects in the same way as $body(C, m)$.

Now associate with every node n in $struc(C_1)$, such that $label(n)$ is the name of class $C = (c, A, M)$ in $flat(H)$, the set of functional forms corresponding to C:

$$funcs(n) = \{func(C, m) \mid m(P) = E \in M\}.$$

Graph homomorphism φ is faithful with respect to methods if it maps methods in one class to methods in another class consistently, i.e., if for every node n in $struc(C_1)$, such that $label(n) \in CN$, the following holds:

$$\forall f_1 \in funcs(n)\ \exists f_2 \in funcs(\varphi(n)) : f_1[\varphi] = f_2.$$

Example 10. Let C_p be class 'Person', C_e be class 'Employee', and C_d be class 'Date' of Example 1. The tree homomorphism from $struc(flat(C_p))$ to $struc(flat(C_e))$ that preserves labels, except the label of the root, is faithful w.r.t. methods, because class C_d has no methods and class C_p has only one method ('change'), whose imaginary counterpart in C_e corresponds to method 'change' in C_e. □

Finally, we define a synthetic subclass relation on flattened classes as follows: C_1 is a subclass of C_2, denoted by $C_1 \preceq C_2$, if every attribute in C_2 corresponds to a unique attribute in C_1 whose type is a weak subtype and every method in C_2 corresponds to a method in C_1 whose functional form is a subfunction, such that, whenever an attribute plays a role in a method, the corresponding attribute plays the same role in the corresponding method, i.e., if there is an injective tree homomorphism from $struc(C_2)$ to $struc(C_1)$ that

1.a. preserves labels of nodes, except class names
  b. is faithful with respect to classes
2.a. preserves labels of arrows, except attribute names
  b. is faithful with respect to attributes
3. is faithful with respect to methods.

In [17], it is proven that the subclass relation is reflexive and transitive. Of course, other subclass relations could have been chosen. The motivation for choosing this subclass order is that classes should not be compared by the name and the type of their attributes (i.e., syntactic) only, but also by the meaning of their attributes (i.e., semantic). The chosen subclass order compares classes by the following characteristics: the structure of the objects in their extensions (requirement 1 and 2) and the way these objects are manipulated (requirement 3). These characteristics can be regarded as abstract semantics for classes, where classes are semantically equal if the objects in their extensions have the same structure and are manipulated in the same way. Since abstract semantics are used to compare classes, rather than real world semantics, the subclass order is called synthetic.

Example 11. Let C_p be class 'Person' and C_e be class 'Employee' of Example 1. Then $flat(C_e)$ is a subclass of $flat(C_p)$. □

Example 12. The following well-defined class hierarchy is a part of the definition of a drawing tool:

```
Class Square
Attributes
    x_left_up:int
    y_left_up:int
    width:int
Methods
    set (x:int, y:int) =
        x_left_up := x; y_left_up := y
    translate (delta_x:int, delta_y:int) =
        x_left_up := x_left_up + delta_x;
        y_left_up := y_left_up + delta_y
```

Endclass
Class Rectangle
Attributes
x_left_up:int
y_left_up:int
width_x:int
width_y:int
Methods
set (x:int, y:int) =
x_left_up := x; y_left_up := y
translate (delta_x:int, delta_y:int) =
x_left_up := x_left_up + delta_x;
y_left_up := y_left_up + delta_y
rotate = y_left_up := y_left_up + width_x;
width_x := width_y - width_x;
width_y := width_y - width_x;
width_x := width_x + width_y
Endclass.

The designer has chosen to model squares by the coordinates of the leftupper corner and the width, and rectangles by the coordinates of the leftupper corner and the width in both directions. According to the subclass order on flattened classes: $\mathit{flat}(C_r) \preceq \mathit{flat}(C_s)$. This does not mean that every rectangle is a square. It only means that every description of a rectangle, as given by the designer, can be regarded as a description of a square, viz., by neglecting the width in one of the two directions. □

5 Integration of Class Hierarchies

In this section, we describe integration of class hierarchies. First, we define class equivalence using the subclass order and factorisation of classes using a join operator w.r.t. the subclass order. Subsequently, we define a normalisation procedure, which identifies and factorises the classes in a class hierarchy until all classes have been factorised.

5.1 Factorisation of Classes

Let $\mathcal{H}$ be the set of well-defined class hierarchies and

$$\mathcal{FH} = \{\mathit{flat}(H) \mid H \in \mathcal{H}\}$$

be the set of well-defined flattened class hierarchies. Let H be a well-defined class hierarchy and F be its flattened form. For every class $D = (d, A, M)$ in F, let $\mathit{sups}(D)$ be the set of syntactic superclasses of D:

$$\mathit{sups}(D) = \{(d_{(A',M')}, A', M') \mid A' \subseteq A \wedge M' \subseteq M \wedge F \cup \{(d_{(A',M')}, A', M')\} \in \mathcal{FH}\}.$$

where $d_{(A,M)} = d$ and every other $d_{(A',M')}$ is a unique class name depending on A' and M'. The set of classes that can be constructed from F, denoted by $\mathcal{S}_F$, consists of the syntactic superclasses of all classes in F:

$$\mathcal{S}_F = \{C \in sups(D) \mid D \in F\}.$$

And, for every class $D = (d, A, M)$ in $\mathcal{S}_F - F$, let $sups(D)$ be the set of syntactic superclasses of D:

$$sups(D) = \{(d', A', M') \in \mathcal{S}_F \mid A' \subseteq A \wedge M' \subseteq M\}.$$

The subclass order $\preceq$ on $\mathcal{S}_F$, as defined in Section 4, induces an equivalence relation $\cong$ on $\mathcal{S}_F$, as follows:

$$C_1 \cong C_2 \Leftrightarrow C_1 \preceq C_2 \wedge C_2 \preceq C_1.$$

If $C \in \mathcal{S}_F$ is a class, then $[C] = \{C' \in \mathcal{S}_F \mid C' \cong C\}$ represents all classes which are equivalent to C. The universe of classes modulo equivalence is given by:

$$\hat{\mathcal{S}}_F = \{[C] \mid C \in \mathcal{S}_F\}.$$

The subclass order on $\mathcal{S}_F$ induces a subclass relation on $\hat{\mathcal{S}}_F$:

$$[C_1] \preceq [C_2] \Leftrightarrow C_1 \preceq C_2$$

It follows that the subclass relation on $\hat{\mathcal{S}}_F$ is a partial order, because the subclass order on $\mathcal{S}_F$ is reflexive and transitive.

Factorisation of classes is defined in terms of a join operator w.r.t. the subclass order. For every pair of classes in $\hat{\mathcal{S}}_F$, the join operator defines the set of least common superclasses. Let C_1 and C_2 be a pair of classes in $\mathcal{S}_F$. The set

$$sups(C_1, C_2) = \{D_1 \in sups(C_1) \mid \exists D_2 \in sups(C_2) : D_1 \cong D_2\}$$

can, up to equivalence, be regarded as the set of common superclasses of C_1 and C_2. Define a binary operator $\sqcup_F$ on $\hat{\mathcal{S}}_F$ as follows:

$$[C_1] \sqcup_F [C_2] = \{[C] \mid C \in sups(C_1, C_2) \mid \forall D \in sups(C_1, C_2) : D \preceq C \Rightarrow C \cong D\}.$$

Then $[C_1] \sqcup_F [C_2]$ is the set of least common superclasses of $[C_1]$ and $[C_2]$. Hence, $\sqcup_F$ can be regarded as a kind of join operator.

Let D_1 and D_2 be classes in H. Every element $[C]$ of the set of least common superclasses of $[flat(D_1)]$ and $[flat(D_2)]$ gives, up to equivalence, one possibility to factorise D_1 and D_2: choose a member (d, A, M) of $[C]$, define a new class $D = (d, \emptyset, A, M)$, and redefine D_1 and D_2 to be subclasses of D. (It can be necessary to replace attributes in D_1 and D_2 (and, hence, rename attributes in methods in other classes that refer to D_1 and D_2) and rename methods in D_1 and D_2 in order to make D a syntactic superclass of both $flat(D_1)$ and $flat(D_2)$).

Example 13. Let H_d be the class hierarchy of Example 12. Furthermore, let C_s be class 'Square' and C_r be class 'Rectangle' in H_d. The set of least common superclasses of C_s and C_r is given by:

$$[flat(C_s)] \sqcup_{flat(H_d)} [flat(C_r)] = \{[flat(C_s)]\}.$$

Hence, there is, up to equivalence, only one least common superclass, and, up to equivalence, only one possibility to factorise C_r.

Now, let the designer choose a class from $[flat(C_s)]$. The class chosen is equivalent to C_s. If C_s is chosen, then there are still several (equivalent) possibilities to factorise C_r, because there are several possible homomorphisms from $struc(flat(C_s))$ to $struc(flat(C_r))$. The first possibility to factorise C_r is:

Class Rectangle **Isa** Square
Attributes
width_y:int
Methods
rotate = y_left_up := y_left_up + width;
width := width_y - width;
width_y := width_y - width;
width := width + width_y
Endclass.

Note that attribute 'width_x' has been renamed to 'width'. The second (equivalent) possibility to factorise C_r is:

Class Rectangle **Isa** Square
Attributes
width_x:int
Methods
rotate = y_left_up := y_left_up + width_x;
width_x := width - width_x;
width := width - width_x;
width_x := width_x + width
Endclass.

Of course, the designer could choose not to factorise, or to factorise only partially. For example, the designer could decide to define a new class with attributes 'x_left_up' an 'y_left_up' and methods 'set' and 'translate', and redefine classes 'Square' and 'Rectangle' to be subclasses of the new class. Partial factorisation is described in [17]. □

5.2 Normalisation of Class Hierarchies

In this subsection, we define a normalisation procedure, which identifies and factorises the classes in a class hierarchy until all classes have been factorised. First, we introduce a subhierarchy relation on flattened class hierarchies. Let $\mathcal{FH}_F$ be the universe of class hierarchies which consist of classes in F and superclasses of classes in F:

$$\mathcal{FH}_F = \{F_0 \in \mathcal{FH} \mid F_0 \subseteq \mathcal{S}_F\}.$$

For every class hierarchy $F_0 \in \mathcal{FH}_F$, let $\hat{F}_0 = \{[C] \mid C \in F_0\}$ be the class hierarchy modulo equivalence. The universe of class hierarchies modulo equivalence (w.r.t. F) denoted by $\widehat{\mathcal{FH}}_F$, is defined as:

$$\widehat{\mathcal{FH}}_F = \{\hat{F}_0 \mid F_0 \in \mathcal{FH}_F\}.$$

The subhierarchy relation on class hierarchies modulo equivalence, denoted by $\preceq$, is defined as a superset order on the classes in the class hierarchies:

$$\hat{F}_1 \preceq \hat{F}_2 \Leftrightarrow \hat{F}_1 \supseteq \hat{F}_2.$$

Note that the subhierarchy relation resembles the subclass relation: subhierarchy $\hat{F}_1$ is a refinement of superhierarchy $\hat{F}_2$. Since $\widehat{\mathcal{FH}}_F$ is finite and meets (unions) exist w.r.t. $\preceq$, $(\widehat{\mathcal{FH}}_F, \preceq)$ is a complete lattice [10].

Normalisation of class hierarchies is defined using factorisation of classes. A class hierarchy $\hat{F}_0 \in \widehat{\mathcal{FH}_F}$ is in normal form if it is closed under joins:

$$[C_1] \in \hat{F}' \wedge [C_2] \in \hat{F}' \Rightarrow [C_1] \sqcup_F [C_2] \subseteq \hat{F}'.$$

Note that we use $\subseteq$, because $[C_1] \sqcup_F [C_2]$ yields a set. So, normal forms are the fixpoints in $\widehat{\mathcal{FH}_F}$ of $\varphi(X) = X \cup \bigcup\{Y_1 \sqcup_F Y_2 \mid Y_1, Y_2 \in X\}$.

The normal form of $\hat{F}$, denoted by $nf(\hat{F})$, is the greatest subhierarchy of $\hat{F}$ that is in normal form, i.e., the greatest fixpoint in $(\widehat{\mathcal{FH}_F}, \preceq)$ of the function φ_F, defined as:

$$\varphi_F(X) = \hat{F} \cup X \cup \bigcup\{Y_1 \sqcup_F Y_2 \mid Y_1, Y_2 \in X\}.$$

In [17], it is proven that the normal form of $\hat{F}$ is characterised by:

$$nf(\hat{F}) = \bigcup\{\varphi_F^i(\emptyset) \mid i \in I\!N\}.$$

It follows that the normal form of a class hierarchy can be computed by a simple fixpoint iteration, starting from the empty set, using the join operator w.r.t. the subclass order to factorise classes. This iteration procedure to normalise flattened class hierarchy $\hat{F}$ gives a way to normalise non-flattened class hierarchy H, as follows. Let N be the smallest natural number such that $\varphi_F^N(\emptyset) = nf(\hat{F})$. Compute a sequence $H_0, \cdots, H_{N+1}$ of non-flattened class hierarchies, where $H_0 = \emptyset$ and $H_1 = H$, as follows:

initialise H_{i+1} by H_i
for every pair of classes D_1 and D_2 in $H_i \setminus H_{i-1}$
  let the designer choose a class $[C]$ from $[\mathit{flat}(D_1)] \sqcup_F [\mathit{flat}(D_2)]$
  let the designer choose a member (d, A, M) of $[C]$
  add $D = (d, \emptyset, A, M)$ to H
  redefine D_1 and D_2 to be subclasses of D.

The resulting non-flattened class hierarchy H_{N+1} is characterised by:

$$nf(\hat{F}) \preceq \widehat{\mathit{flat}(H_{N+1})} \preceq \hat{F},$$

up to renaming of attributes and methods.

Example 14. Let H_d be the class hierarchy of Example 12. Then $\widehat{\mathit{flat}(H_d)}$ is already in normal form, and we can normalise H_d in one step obtaining the class hierarchy consisting of class 'Square' of Example 12 and of class 'Rectangle' of Example 13. □

The normalisation procedure leads to a natural framework for integration of two class hierarchies H_1 and H_2: define H_0 to be the union of H_1 and H_2 and normalise H_0. If class $D_1 \in H_1$ and $D_2 \in H_2$ have the same name, then it is necessary to choose a new name for one of the classes, e.g., for D_1, and to replace all occurrences of the old name in H_1 by the new name.

Example 15. Let H_d be the class hierarchy of Example 12 and H_p be the class hierarchy consisting of the following class:

Class Point
Attributes
  x_co:int

```
    y_co:int
Methods
    reset (x:int, y:int) =
        x_co := x; y_co := y
    translate (delta_x:int, delta_y:int) =
        x_co := x_co + delta_x;
        y_co := y_co + delta_y
Endclass.
```

If we integrate H_d and H_p, we obtain something like the class hierarchy consisting of class 'Point', class 'Rectangle' of Example 13, and the following class:

```
Class Square Isa Point
Attributes
    width:int
Endclass.
```

A more natural solution would be to aggregate the coordinates of the left upper corner in class 'Square' and class 'Rectangle' of Example 12 into a new attribute 'left_up:Point'. This is the subject of future research. □

6 Conclusion

This paper develops a formal approach to integrate class hierarchies on the basis of syntactic and semantic similarity of classes, based on a synthetic subclass order, which induces an equivalence relation on classes. The result is a natural framework for integration of class hierarchies, where classes are identified using the equivalence relation and factorised using a join operator w.r.t. the subclass order.

In contrast with existing literature [5, 6, 9], both attributes and methods are used to compare classes and behaviour of methods is used to compare attributes. The benefit is a more semantic approach to support view integration and schema normalisation in object-oriented databases, and schema integration in general, e.g., in multidatabase systems.

Future research will include more sophisticated subclass orders and join operators to cope with extensions of the datamodel (e.g., variant types, attribute specialisation, and retrieval methods) and aggregation of attributes.

Acknowledgement

Special thanks to Carel van den Berg and Leonie van der Voort for their constructive and stimulating criticism while this research was carried out. Also many thanks to Martin Kersten for his useful comments on earlier versions of this paper.

7 Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[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] H. Balsters, R. de By, and R. Zicari. Sets and constraints in an object-oriented data model. Technical report INF90-75, University of Twente, Enschede, The Netherlands, 1990.

[4] H. Balsters and C. de Vreeze. A formal theory of sets in object-oriented contexts. Technical report INF90-74, University of Twente, Enschede, The Netherlands, 1990.

[5] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, pages 323–364, 1987.

[6] P. Bergstein and K. Lieberherr. Incremental class dictionary learning and optimization. In *Proc. European Conf. on Object-Oriented Programming, LNCS 512*, pages 377–395. Springer-Verlag, Berlin, 1991.

[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] C. de Vreeze. Formalization of inheritance of methods in an object-oriented data model. Technical report INF90-76, University of Twente, Enschede, The Netherlands, 1990.

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

[10] G. Graetzer. *General Lattice Theory.* Academic Press, New York, NY, 1978.

[11] S. Hong, G. van den Goor, and S. Brinkkemper. A comparison of object-oriented analysis and design methodologies. In *Proc. Computing Science in the Netherlands*, pages 120–131. Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1992.

[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] C. Koster. On infinite modes. *ACM SIGPLAN Notices*, 4(3):109–112, 1969.

[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] T. Olle, J. Hagelstein, I. MacDonald, C. Rolland, H. Sol, F. van Assche, and A. Verrijn Stuart (Eds.). *Information Systems Methodologies - A Framework for Understanding.* Addison-Wesley, Reading, MA, 1988.

[16] T. Teorey and J. Fry. *Design of Database Structures.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[17] C. Thieme and A. Siebes. Schema integration in object-oriented databases. Report CS-R93xx, CWI, Amsterdam, The Netherlands, 1993.

A Syntax Definition of Class Hierarchies

This appendix gives the syntax definition of class hierarchies. For that purpose, 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 constants of type 'bool', 'int', and 'string'. These 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$^+$] [**'Methods'** Meth$^+$] **'Endclass'**
Att	::=	AN ':' Type
Type	::=	BasType | '{' Type '}' | '<' FieldList '>' | CN
BasType	::=	'bool' | 'int' | 'string'
FieldList	::=	Field | Field ',' FieldList
Field	::=	L ':' Type
Meth	::=	MN ['(' ParList ')'] '=' AsnList
ParList	::=	Par | Par ',' ParList
Par	::=	L ':' BasType
AsnList	::=	Assign | Assign ';' AsnList
Assign	::=	AN ':=' Source | **'insert('** Source ',' AN ')'**
Source	::=	Term | Term '+' Source | Term '×' Source | Term '∨' Source | Term '∧' Source
Term	::=	Cons | Sel
Sel	::=	L | AN | L '.' Sel | AN '.' Sel

An assignment of the form **'insert**(e, V)' should be read as '$V := V \cup \{e\}$'.

B Definition of Well-typed Methods

This appendix defines well-typed methods. For that purpose, the definition of *struc* Subsection 3.1 is pre-supposed. Let H be a class hierarchy, such that classes have a unique name, classes only refer to classes which belong to the class hierarchy, and attributes and methods have a unique name within their class. Furthermore, let C be a class in H and $m(P) = E$ be a method in C. The source $t_1 \circ_1 \cdots \circ_n t_{n+1}$ $(n \geq 0)$ of an assignment in E is well-typed if:

1. every term t_i is a constant in *Cons*, or a parameter in P, or (the labeling of) a path in $struc(flat(C))$, starting at the root
2. one of the following holds:
   - $n = 0$, i.e., there are no operators, or
   - every term t_i is of type bool and every operator $\circ_j$ is $\vee$ (disjunction) or $\wedge$ (conjunction), or

- every term t_i is of type int and every operator $\circ_j$ is $+$ (addition) or $\times$ (multiplication), or
- every term t_i is of type string and every operator $\circ_j$ is $+$ (concatenation).

An assignment in E is well-typed if its source an destination are well-typed and their types are equal:

1. if the assignment is of the form $a := s$, then the assignment is well-defined if:
   (a) $a : T$ is an attribute in C for some T,
   (b) s is well-typed and has type $type(T)$;
2. if the assignment is of the form $\mathbf{insert}(s, a)$, then the assignment is well-defined if:
   (a) $a : \{T\}$ is an attribute in C for some T,
   (b) s is well-typed and has type $type(T)$.

Finally, method $m(P) = E$ is well-typed if every assignment in E is well-typed.

C Definition of Functional Forms

This appendix defines functional forms. For that purpose, let H be a well-defined class hierarchy. Furthermore, let C be a class in $flat(H)$ and $m(P) = E$ be a method in class C. If $\{a_1 : T_1, \cdots, a_k : T_k\}$ are the attributes of class C, then the functional form of $m(P) = E$ in C, denoted by $func(C, m)$ is defined as:

$$func(C, m) = \lambda self : type(C) \lambda P. body(C, m),$$

where

$$body(C, m) = eval(E)(< a_1 = self.a_1, \cdots, a_k = self.a_k >),$$

where

$$\begin{aligned}
&eval(L_1; L_2)(\sigma) = eval(L_2)(eval(L_1)(\sigma)),\\
&eval(a_i := s)(< a_1 = e_1, \cdots, a_k = e_k >) =\\
&\quad < a_1 = e_1, \cdots, a_{i-1} = e_{i-1}, a_i = s[a_1 \backslash e_1, \cdots, a_k \backslash e_k],\\
&\qquad\qquad a_{i+1} = e_{i+1}, \cdots, a_k = e_k >,\\
&eval(\mathbf{insert}(s, a_i))(< a_1 = e_1, \cdots, a_k = e_k >) =\\
&\quad < a_1 = e_1, \cdots, a_{i-1} = e_{i-1}, a_i = e_i \cup \{s[a_1 \backslash e_1, \cdots, a_k \backslash e_k]\},\\
&\qquad\qquad a_{i+1} = e_{i+1}, \cdots, a_k = e_k >.
\end{aligned}$$

The expression $s[a \backslash e]$ is the expression that results when every term a in s is replaced by e and every term $a.r_1.\cdots.r_n$ in s is replaced by $e.r_1.\cdots.r_n$.