Schema integration in object-oriented databases

C.J.E Thieme, A.P.J.M. Siebes

Computer Science/Department of Algorithmics and Architecture

CS-R9320 1993

Schema Integration in Object-Oriented Databases

Christiaan Thieme and Arno Siebes
{ct,arno}@cwi.nl

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

This report presents a formal approach to support schema integration in object-oriented databases. The basis of the approach is a synthetic 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. Classes are identified using an equivalence relation induced by the subclass order and factorised using a join operator w.r.t. the subclass order. Class hierarchies are integrated by repeatedly identifying and factorising classes. The novelty of this report is that both attributes and methods are used to compare classes and behaviour of methods is used to compare attributes.

1991 CR Categories: D.1.5: [Software] Object-oriented programming, D.2.2: [Software] Tools and techniques, H.2.1: [Database management] Logical design.
Keywords and Phrases: Database design, Object-oriented databases, Schema integration.
Note: This research is partly funded by the Dutch Organisation for Scientific Research through NFI-grant NF74.

Contents

1 Introduction **2**

2 Class hierarchies **2**

3 Types and functions **7**
3.1 Underlying types 7
3.2 Functional forms 11

4 Comparison of classes **14**

5 Integration of class hierarchies **17**
5.1 Factorisation of classes 18
5.2 Normalisation of class hierarchies 21

6 Conclusion **27**

7 Bibliography **27**

A Decidability of extensional equality **28**

1 Introduction

Database design is a complex, iterative process consisting of several activities, including conceptual design and implementation design [18]. 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 [16], which includes guidelines, techniques, methods, and tools to support the activities of the designer.

This report addresses the problem of identifying and factorising classes in class hierarchies, which constitute the database schemas in object-oriented data models [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 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, e.g., 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, optimisation of class hierarchies 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 report 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 compares classes by the structure of the objects in their extensions and the way these objects are manipulated. Finally, a natural framework for integration of class hierarchies is described, 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]. First, we give a BNF-grammar that generates class hierarchies as defined by the database designer. Subsequently, we define a number of constraints which have to be imposed on class hierarchies to make them well-defined. Finally, we introduce flattened class hierarchies.

In order to define class hierarchies, 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 'int' and 'string'. Furthermore, a lexicographic order $<$ is postulated on the set of attribute names.

Definition 1. Class hierarchies are the sentences of the following BNF-grammar, where the nonterminals CN, AN, MN, L, and Cons generate the elements of CN, AN, MN, L, and *Cons*, respectively:

```
Hierarchy   ::=   Class+
Class       ::=   'Class' CN [ 'Isa' CN+ ]
                  [ 'Attributes' Att+ ]
                  [ 'Methods' Meth+ ]
                  'Endclass'
Att         ::=   AN ':' Type
Type        ::=   BasType | '{' Type '}' | '<' FieldList '>' | CN
BasType     ::=   '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        ::=   Cons | Sel
Sel         ::=   L | AN | L '.' Sel | AN '.' Sel
```

The plus sign ($^+$) denotes a finite, nonempty, repetition; square brackets ([ ]) denote an option; and the vertical bar (|) denotes a choice. □

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

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

```
Class Person
Attributes
      name : string
      dob : Date
      mother : Person
Methods
      change (s:string) = name := s
Endclass
```

Class Employee **Isa** Person
Attributes
  company : string
Methods
  increase (i:int) = salary := salary+i
Endclass.

□

However, the class hierarchy of Example 1 is not well-defined, because attribute 'dob:Date' refers to a class ('Date') which does not belong to the class hierarchy, and method 'increase' assigns to an attribute ('salary:T') which does not belong to the attributes of class 'Employee'.

A class hierarchy is well-defined if it satisfies two constraints. The first constraint (completeness) is that classes have a unique name and only refer to classes in the class hierarchy. The second constraint (soundness) is that attributes and methods (inherited attributes and methods included) have a unique name within their class and are well-typed. To formalise this, let us abbreviate a class

Class c **Isa** $d_1 \cdots d_n$
Attributes
  $a_1 : T_1$
  $\vdots$
  $a_k : T_k$
Methods
  $m_1(P_1) = E_1$
  $\vdots$
  $m_l(P_l) = E_l$
Endclass

to $(c, \{d_1, \cdots, d_n\}, \{a_1 : T_1, \cdots, a_k : T_k\}, \{m_1(P_1) = E_1, \cdots, m_l(P_l) = E_l\})$.

That is, a little of the syntactic sugar of the class is removed.

Example 2. The classes of Example 1 can be abbreviated to:

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

□

Furthermore, let the abbreviation of a class hierarchy be the set of abbreviations of the classes in the hierarchy. For example, the class hierarchy of Example 1 can be abbreviated to the set $\{C_p, C_e\}$, consisting of the classes of Example 2. In the sequel, sugared classes will mainly be used in examples, and abbreviated classes mainly in definitions.

To formalise the completeness constraint, let $name(C)$ be the projection of the name component of an abbreviated class C and $refs(T)$ be the set of classes a type T refers to:

$$\begin{array}{ll}
\mathit{name}(c,S,A,M)=c, & \\
\mathit{refs}(c)=c & \text{if } c\in CN,\\
\mathit{refs}(B)=\emptyset & \text{if } B\in\{\text{int},\text{string}\},\\
\mathit{refs}(\{U\})=\mathit{refs}(U), & \\
\mathit{refs}(<l_1:U_1,\cdots,l_n:U_n>)=\mathit{refs}(U_1)\cup\cdots\cup\mathit{refs}(U_n). &
\end{array}$$

For example, *refs*(<e:Employee,d:Date>) = $\{\text{Employee},\text{Date}\}$. Let H be an abbreviated class hierarchy.

Definition 2. The completeness constraint for H, denoted by $\mathit{Complete}(H)$, is the conjunction of the following three items:

1. $((c,S_1,A_1,M_1)\in H \wedge (c,S_2,A_2,M_2)\in H) \Rightarrow (S_1=S_2\wedge A_1=A_2\wedge M_1=M_2)$
2. $(c,S,A,M)\in H \Rightarrow \forall c'\in S\ \exists C'\in H\ [\mathit{name}(C')=c']$
3. $(c,S,A,M)\in H \Rightarrow \forall a:T\in A\ \forall c'\in\mathit{refs}(T)\ \exists C'\in H\ [\mathit{name}(C')=c']$.

□

The constraint says that every class has a unique name, every class only refers to superclasses which belong to H, and every attribute only refers to classes which belong to H.

Cycles in the subclass (**Isa**) relation are allowed, because there is no reason to prohibit cycles: if C is a subclass of D and D is a subclass of C, then C and D are necessarily equal.

Example 3. Class hierarchy $\{C_p, C_e\}$, consisting of the classes of Example 2, satisfies the first and the second item, but not the third, because attribute 'dob:Date' refers to a class which does not belong to the class hierarchy. However, class hierarchy $\{C_p, C_e, C_d\}$, where C_d is (Date, $\emptyset$, {day:int, month:int, year:int}, $\emptyset$), satisfies all items and, hence, the completeness constraint. □

To formalise the soundness constraint, let H be a class hierarchy that satisfies the completeness constraint. For every class $C=(c,S,A,M)$ in H, let $\mathit{atts}(C)$ be the set of all attributes of C and $\mathit{meths}(C)$ be the set of all methods of C:

$$\begin{array}{l}
\mathit{atts}(C)=A\cup\{a:T\mid \exists C'\in H\ [\mathit{name}(C')\in S\wedge a:T\in\mathit{atts}(C')]\},\\
\mathit{meths}(C)=M\cup\{m(P)=E\mid \exists C'\in H\ [\mathit{name}(C')\in S\wedge m(P)=E\in\mathit{meths}(C')]\}.
\end{array}$$

Note that, if S is empty, then the right-hand side of the union is empty. Otherwise, if S is not empty, the right-hand side is not empty and depends on the superclasses of C in H. Furthermore, define *WType* to be the set of well-defined types.

Definition 3. The set *WType* is the smallest set, such that:

1. if $c\in CN$ is a class name, then $c\in \mathit{WType}$
2. if $B\in\{\text{int},\text{string}\}$ is a basic type, then $B\in\mathit{WType}$
3. if $U\in\mathit{WType}$ is a well-defined type, then $\{U\}\in\mathit{WType}$
4. if $\{U_1,\cdots,U_n\}\subseteq\mathit{WType}$ is a set of well-defined types and $\{l_1,\cdots,l_n\}\subseteq L$ is a set of n distinct labels, then $<l_1:U_1\cdots,l_n:U_n>\ \in\mathit{WType}$.

□

And define $\mathit{WTyMeth}_C$ to be the set of well-typed methods in class C, where a method is well-typed if for every assignment in its body, the type of the source and destination are equal. The formal definition of well-typed methods is given in Section 3.

Definition 4. The soundness constraint for H, denoted by $\mathit{Sound}(H)$, is the conjunction of the following four items:

1. $(C \in H \wedge a : T_1 \in \mathit{atts}(C) \wedge a : T_2 \in \mathit{atts}(C)) \Rightarrow T_1 = T_2$
2. $(C \in H \wedge a : T \in \mathit{atts}(C)) \Rightarrow T \in \mathit{WType}$
3. $(C \in H \wedge m(P_1) = E_1 \in \mathit{meths}(C) \wedge m(P_2) = E_2 \in \mathit{meths}(C)) \Rightarrow (P_1 = P_2 \wedge E_1 = E_2)$
4. $(C \in H \wedge m(P) = E \in \mathit{meths}(C)) \Rightarrow m(P) = E \in \mathit{WTyMeth}_C$.

□

The constraint says that every attribute has a unique name within its class, every attribute has a well-defined type, every method has a unique name within its class, and every method is well-typed. Hence, overriding of attributes and methods is not allowed.

Example 4. Class hierarchy $\{C_p, C_e, C_d\}$ of Example 3 satisfies the completeness constraint and the first, second, and third item of the soundness constraint, but not the fourth item, because the type of destination 'salary' cannot be determined (because 'salary:T' does not belong to the attributes of C_e). However, class hierarchy $\{C_p, C_e', C_d\}$, where

C_e' = (Employee, {Person}, {company:string, salary:int},
{increase (i:int) = salary := salary+i}),

satisfies the completeness constraint and the soundness constraint, and, hence, is well-defined.
□

Finally, we introduce flattened class hierarchies to simplify the framework for integration of class hierarchies. Let H be a class hierarchy that satisfies the completeness constraint. The flattened form of a class in H is obtained by removing superclasses and accumulating attributes and methods.

Definition 5. Let C be a class in H. The flattened form of C, denoted by $\mathit{flat}(C)$, is defined as:

$$\mathit{flat}(C) = (\mathit{name}(C), \mathit{atts}(C), \mathit{meths}(C)).$$

□

Example 5. Let $\{C_p, C_e', C_d\}$ be the class hierarchy of Example 4. The flattened form of class C_e' is given by:

$\mathit{flat}(C_e')$ = (Employee,
{name:string, dob:Date, mother:Person, company:string, salary:int},
{change (s:string) = name := s, increase (i:int) = salary := salary+i }).

□

The flattened class hierarchy corresponding to H is obtained by replacing every class by its flattened form.

Definition 6. The flattened class hierarchy corresponding to H, denoted by $\mathit{flat}(H)$, is defined as:

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

□

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 a subclass order on flattened classes.

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.

Let H be a class hierarchy that satisfies the completeness constraint. The underlying type of a class in $\mathit{flat}(H)$ is an aggregation of its attributes, where pointer types are used to cope with attributes that refer to classes.

Definition 7. Let $C = (c, A, M)$ be a class in $\mathit{flat}(H)$ and $A = \{a_1 : T_1, \cdots, a_k : T_k\}$ be the attributes of C, such that $a_i < a_{i+1}$ for $i \in \{1, \cdots, k-1\}$. 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

$$\begin{array}{ll} \mathit{type}(c') = \uparrow \mathit{type}(C') & \text{if } C' = (c', A', M') \in \mathit{flat}(H), \\ \mathit{type}(B) = B & \text{if } B \in \{\text{int}, \text{string}\}, \\ \mathit{type}(\{U\}) = \{\mathit{type}(U)\}, & \\ \mathit{type}(< l_1 : U_1, \cdots, l_n : U_n >) = < l_1 : \mathit{type}(U_1), \cdots, l_n : \mathit{type}(U_n) >. & \end{array}$$

□

The underlying type of a class describes the structure of the class, i.e., the structure of the objects in its extensions. Note that the underlying type of a class depends on the hierarchy. Furthermore, observe that, if a class is recursive, then its underlying type is recursive [13].

Example 6. Let $\{C_p, C_e, C_d\}$ be the class hierarchy of Example 3. The underlying type τ_p of class *flat*(C_p) is given by:

$$\tau_p = \text{<name:string, dob:}\uparrow \tau_d\text{, mother:}\uparrow \tau_p \text{>},$$

where

$$\tau_d = \text{<day:int, month:int, year:int>}.$$

Underlying type τ_p is recursive, because τ_p appears on the right-hand side of its definition. □

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 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:

$$\tau_1 =_{struc} \tau_2 \Leftrightarrow \mathit{tree}(\tau_1) = \mathit{tree}(\tau_2),$$

where $\mathit{tree}(\tau)$ is the tree representing type τ. That is, structural equality of types is defined as equality of their trees.

We define structural equality of classes as structural equality of their underlying types.

Definition 8. Let C_1 and C_2 be classes in *flat*(H). Structural equality of classes, denoted by $=_{struc}$, is defined as:

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

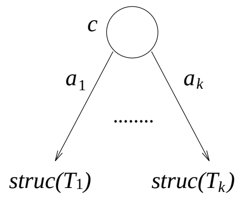
□

Lemma 1. Structural equality of classes is an equivalence relation.
Proof. The lemma follows from the fact that structural equality of types is an equivalence relation. □

The algorithm from [13] can also 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 labelling class nodes by the name of the corresponding class.

Definition 9. Let $C = (c, A, M)$ be a class in *flat*(H) and $A = \{a_1 : T_1, \cdots, a_k : T_k\}$ be the attributes of C, such that $a_i < a_{i+1}$ for $i \in \{1, \cdots, k-1\}$. The tree representing the structure of class C, denoted by *struc*(C), is defined as:



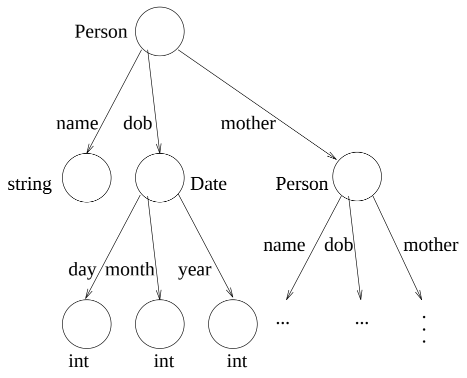
where

$\mathit{struc}(c') = \mathit{struc}(C')$ if $C' = (c', A', M') \in \mathit{flat}(H)$,
$\mathit{struc}(B)$ has one node, labeled B if $B \in \{\text{int}, \text{string}\}$,
$\mathit{struc}(\{U\})$ 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 7. Let $\{C_p, C_e, C_d\}$ be the class hierarchy of Example 3. The structure of class $\mathit{flat}(C_p)$ is represented by the following infinite tree ($\mathit{struc}(\mathit{flat}(C_p))$):



□

Adapted trees will be used to define a subclass order on flattened classes in Section 4 and class equivalence in Section 5.

Structural equality of classes can be defined in terms of equality of adapted trees, where two trees are equal if their roots are equal (i.e., have the same label) and their children (including the arrows pointing to them) are pairwise equal.

Lemma 2. Let C_1 and C_2 be classes in $\mathit{flat}(H)$. Let $\mathit{struc}^*(C_i)$ be $\mathit{struc}(C_i)$ with class names removed ($i \in \{1, 2\}$). Then:

$$\mathit{type}(C_1) =_{\mathit{struc}} \mathit{type}(C_2) \Leftrightarrow \mathit{struc}^*(C_1) = \mathit{struc}^*(C_2).$$

Proof. We have that $\mathit{type}(C_1) =_{\mathit{struc}} \mathit{type}(C_2) \Leftrightarrow \mathit{tree}(\mathit{type}(C_1)) = \mathit{tree}(\mathit{type}(C_2))$. Since pointers in $\mathit{tree}(\mathit{type}(C_i))$ only occur in combination with a class and $\mathit{struc}^*(C_i)$ is obtained from $\mathit{tree}(\mathit{type}(C_i))$ by removing pointers, we also have that $\mathit{tree}(\mathit{type}(C_1)) = \mathit{tree}(\mathit{type}(C_2))$ $\Leftrightarrow \mathit{struc}^*(C_1) = \mathit{struc}^*(C_2)$. □

In [7], a natural notion of subtyping is defined for record types: τ_1 is a subtype of 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.

Definition 10. Let C_1 and C_2 be classes in $\mathit{flat}(H)$. Let $\mathit{struc}^*(C_i)$ be $\mathit{struc}(C_i)$ with class names removed ($i \in \{1, 2\}$). The subtype relation, denoted by $\leq$, is defined as:

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

□

Lemma 3. The subtype relation on underlying types is a partial order w.r.t. structural equality.
Proof. The lemma follows from the fact that the subtree relation is a partial order (if the root of a subtree must also be the root of its supertree) and Lemma 2. □

Lemma 4. Let H be a well-defined class hierarchy and $D' = (d', \{d\}, A', M')$ be a subclass of $D = (d, \emptyset, A, M)$ in H. Then:

$$\mathit{type}(\mathit{flat}(D')) \leq \mathit{type}(\mathit{flat}(D)).$$

Proof. The lemma follows from Definition 10, 5, and 9, and the fact that $\mathit{atts}(D) \subseteq \mathit{atts}(D')$.
□

The lemma states that the subtype relation on underlying types of classes respects the subclass (**Isa**) relation.

Example 8. Let $\{C_p, C_e, C_d\}$ be the class hierarchy of Example 3. Then:

$$\mathit{type}(\mathit{flat}(C_e)) \leq \mathit{type}(\mathit{flat}(C_p)).$$

□

Lemma 5. Let C_1 and C_2 be classes in $\mathit{flat}(H)$. Then:

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

Proof. The lemma follows from Definition 8 and Lemma 3. □

The lemma states that structural equality of classes can also be defined in terms of the subtype relation on underlying types.

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

In this subsection, we introduce the functional form of a method and define extensional equality for functional forms. Before defining functional forms, we first give a characterisation of well-typed methods.

Definition 11. Let H be a class hierarchy that satisfies the completeness constraint and part of the soundness constraint: attributes and methods have a unique name within their class and attributes have a well-defined type. Furthermore, let D be a class in H and $m(P) = E \in \mathit{meths}(D)$ be one of the methods of D. The type of a term t in an assignment in E is defined as follows:

1. if t is a constant in Cons of type B, then t has type B
2. if $t : B$ is a parameter in P, then t has type B
3. if t is (the labeling of) a path in $\mathit{struc}(\mathit{flat}(D))$, starting at the root, then there is a tree $\mathit{struc}(T)$ at the end of the path, such that $T \in \mathit{WType}$, and the type of t is $\mathit{type}(T)$
4. otherwise, the type of t is undefined.

The type of a source $s \equiv t_1 \circ_1 \cdots \circ_n t_{n+1}$ $(n \geq 0)$ of an assignment in E is defined as:

1. if $n = 0$, i.e., there are no operators, then the type of s is the type of t_1
2. if every term t_i is of type int and every operator $\circ_j$ is $+$ (addition) or $\times$ (multiplication), then s has type int
3. if every term t_i is of type string and every operator $\circ_j$ is $+$ (concatenation), then s has type string
4. otherwise, the type of s is undefined.

An assignment in E is well-typed if the type of the source and destination are equal:

1. if the assignment is of the form $a := s$, then the assignment is well-typed if:
   (a) $a : T \in \mathit{atts}(D)$ for some $T \in \mathit{WType}$,
   (b) the type of s is a subtype of $\mathit{type}(T)$;
2. if the assignment is of the form $\mathbf{insert}(s, a)$, then the assignment is well-typed if:
   (a) $a : \{T\} \in \mathit{atts}(D)$ for some $T \in \mathit{WType}$,
   (b) the type of s is a subtype of $\mathit{type}(T)$.

Otherwise, an assignment is not well-typed. Finally, method $m(P) = E$ is well-typed in class D if every assignment in E is well-typed. □

Example 9. Let $\{C_p, C_e, C_d\}$ be the class hierarchy of Example 3. Then method 'change' is well-typed in C_e, but method 'increase' is not well-typed in C_e, because there is no attribute 'salary:T' in $\mathit{atts}(C_e)$. □

Lemma 6. Let H be a class hierarchy as in Definition 11. Let $D' = (d', \{d\}, A', M')$ be a subclass of $D = (d, \emptyset, A, M)$ in H and $m(P) = E$ be a well-typed method in class D. Then:

$m(P) = E$ is a well-typed method in class D'.

Proof. Let $m(P) = E$ be a well-typed method in class D. If a term t in an assignment in E is (the labeling of) a path in $struc(flat(D))$ and the type of t in $struc(flat(D))$ is $type(T)$ for some $T \in WType$, then t is also a path in $struc(flat(D'))$ and the type of t in $struc(flat(D'))$ is $type(T)$ as well, because of Definition 5 and 9, and the fact that $atts(D) \subseteq atts(D')$. Let $a := s$ be an assignment in E. Then:

1. if $a : T \in atts(D)$, then $a : T \in atts(D')$, because $atts(D) \subseteq atts(D')$
2. if the type of s, according to D, is a subtype of $type(T)$, then the type of s, according to D', is a subtype of $type(T)$ as well.

It follows that assignment $a := s$ is well-typed in D', because $a := s$ is well-typed in D. A similar argument holds for assignments of the form **insert**(s, a). Hence, $m(P) = E$ is a well-typed method in class D'. □

Every well-typed method corresponds to a functional form.

Let H be a well-defined class hierarchy. The functional form of a method in a class in $flat(H)$ is a function whose body is an accumulation of the assignments in the method body.

Definition 12. Let $C = (c, A, M)$ be a class in $flat(H)$ and $A = \{a_1 : T_1, \cdots, a_k : T_k\}$ be the attributes of C, such that $a_i < a_{i+1}$ for $i \in \{1, \cdots, k-1\}$. Let $m(P) = E \in M$ be one of the methods of C. 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

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

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 $e.r_1. \cdots .r_n$. The type of $body(C, m)$ is $type(C)$, as should be the case, since $m(P) = E$ is an update method. □

The set of functional forms corresponding to the methods of a class describe the way in which the objects in the extensions of the class are manipulated.

Example 10. Let $\{C_p, C'_e, C_d\}$ be the class hierarchy of Example 4. The functional forms of method 'change' in class $\mathit{flat}(C_p)$ and method 'change' in class $\mathit{flat}(C'_e)$ are given by:

$\mathit{func}(\mathit{flat}(C_p),\text{change}) = \lambda \mathit{self} : \tau_p \lambda \text{s} : \text{string}.$
$\quad$ <name=s, dob=*self*.dob, mother=*self*.mother>,
$\mathit{func}(\mathit{flat}(C'_e),\text{change}) = \lambda \mathit{self} : \tau'_e \lambda \text{s} : \text{string}.$
$\quad$ <name=s, dob=*self*.dob, mother=*self*.mother,
$\quad\;$ company=*self*.company, salary=*self*.salary>,

where τ_p is the underlying type of class $\mathit{flat}(C_p)$ and τ'_e is the underlying type of class $\mathit{flat}(C'_e)$.
□

Functional forms will be used to define a subclass order on flattened classes in Section 4 and class equivalence in Section 5.

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:

$$f_1 = f_2 \Leftrightarrow \forall \vec{x}\, [f_1(\vec{x}) = f_2(\vec{x})].$$

Example 11. The following functions are extensionally equal:

$$\lambda x_1 : \text{int}\ \lambda x_2 : \text{int}\ .\ 2 \times x_1 - x_2 \quad = \quad \lambda x_1 : \text{int}\ \lambda x_2 : \text{int}\ .\ x_1 - x_2 + x_1.$$

□

In order to cope with permutations of parameters, extensional equality is adapted slightly for functional forms of methods.

Definition 13. Let C_1 and C_2 be classes in $\mathit{flat}(H)$, such that $\mathit{type}(C_1)$ and $\mathit{type}(C_2)$ are structurally equal. Let $m_1(P_1) = E_1$ be a method in C_1 and $m_2(P_2) = E_2$ be a method in C_2. Extensional equality of functional forms, denoted by $=_{ext}$, is defined as follows:

$$\mathit{func}(C_1, m_1) =_{ext} \mathit{func}(C_2, m_2)$$

if there exists a permutation P'_2 of P_2, such that:

$$\forall i \in \{1, \cdots, k\}\ [\lambda \mathit{self} : \mathit{type}(C_1) \lambda P_1.e_i = \lambda \mathit{self} : \mathit{type}(C_2) \lambda P'_2.e'_i],$$

where

$$\mathit{func}(C_1, m_1) = \lambda \mathit{self} : \mathit{type}(C_1) \lambda P_1. < a_1 = e_1, \cdots, a_k = e_k >$$
$$\mathit{func}(C_2, m_2) = \lambda \mathit{self} : \mathit{type}(C_2) \lambda P_2. < a_1 = e'_1, \cdots, a_k = e'_k >.$$

□

For our language, allowing addition and multiplication for integers, concatenation for strings, and insertion for sets, extensional equality of functional forms is decidable; this is proven in Appendix A.

Theorem 1. Extensional equality is decidable for:

$$\{\mathit{func}(\mathit{flat}(D), m) \mid \exists H \in \mathcal{H}\ [\mathit{Complete}(H) \wedge \mathit{Sound}(H) \wedge D \in H \wedge m(P) = E \in \mathit{meths}(D)]\},$$

where $\mathcal{H}$ is the set of (abbreviations of) class hierarchies generated by the BNF-grammar. □

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 for graph homomorphisms are introduced. 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. $(r \in \mathit{arrows}(G_1) \wedge \mathit{label}(r) = l) \Rightarrow \mathit{label}(\varphi(r)) = l$,

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 report, 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.

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 between trees representing the structure of classes is faithful w.r.t. classes if it maps classes to classes.

Definition 14. Let φ be a graph homomorphism from $\mathit{struc}(C_1)$ to $\mathit{struc}(C_2)$. Then φ is faithful with respect to classes if for every node n in $\mathit{struc}(C_1)$:

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

□

Example 12. Let $\{C_p, C'_e, C_d\}$ be the class hierarchy of Example 4. 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 between trees representing the structure of classes is faithful w.r.t. attributes if it maps attributes in one class to attributes in another class consistently.

Definition 15. Let φ be a graph homomorphism from $\mathit{struc}(C_1)$ to $\mathit{struc}(C_2)$ Then φ is faithful with respect to attributes if for every node n_1 in $\mathit{struc}(C_1)$ with outgoing arrow r_1 and every node n_2 in $\mathit{struc}(C_1)$ with outgoing arrow r_2, the following holds:

$$(\mathit{label}(n_1) = \mathit{label}(n_2) \in CN \wedge \mathit{label}(r_1) = \mathit{label}(r_2) \wedge \mathit{label}(\varphi(n_1)) = \mathit{label}(\varphi(n_2))) \Rightarrow$$
$$\mathit{label}(\varphi(r_1)) = \mathit{label}(\varphi(r_2)).$$

□

Example 13. Let $\{C_p, C'_e, C_d\}$ be the class hierarchy of Example 4. 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. 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 $\mathit{func}(C_2, m_2)$ is a subfunction of $\mathit{func}(C_1, m_1)$. A graph homomorphism between trees representing the structure of classes is faithful w.r.t. methods if it maps methods in one class to methods in another class consistently.

Definition 16. Let φ be an injective graph homomorphism from $\mathit{struc}(C_1)$ to $\mathit{struc}(C_2)$ that preserves labels, except class names and attribute names, and is faithful w.r.t. to classes and w.r.t. attributes. If such a graph homomorphism exists, then we say that $\mathit{type}(C_2)$ is a weak subtype of $\mathit{type}(C_1)$.

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

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

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 $\mathit{struc}(C_1)$ starting at node n, then every occurrence of term $\mathit{label}(r_1). \cdots .\mathit{label}(r_p)$ in E is replaced by $\mathit{label}(\varphi(r_1)). \cdots .\mathit{label}(\varphi(r_p))$.

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

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

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

Graph homomorphism φ is faithful with respect to methods if for every node n in $\mathit{struc}(C_1)$, such that $\mathit{label}(n) \in CN$, the following holds:

$$\forall f_1 \in \mathit{funcs}(n)\ \exists f_2 \in \mathit{funcs}(\varphi(n)) : f_1[\varphi] =_{ext} f_2.$$

□

Example 14. Let $\{C_p, C'_e, C_d\}$ be the class hierarchy of Example 4. 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 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.

Definition 17. Let H be a well-defined class hierarchy. Let C_1 and C_2 be classes in $\mathit{flat}(H)$. Then C_1 is a subclass of C_2, denoted by $C_1 \preceq C_2$, if there is an injective tree homomorphism from $\mathit{struc}(C_2)$ to $\mathit{struc}(C_1)$ 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.

□

Theorem 2. The subclass relation is reflexive and transitive.
Proof. To proof reflexivity, let C be a class. The identity homomorphism from $\mathit{struc}(C)$ to $\mathit{struc}(C)$ is injective and satisfies the requirements of Definition 17. Hence, $C \preceq C$.

To proof transitivity, let C_1, C_2, and C_3 be classes, such that $C_1 \preceq C_2$ and $C_2 \preceq C_3$. Then there exist tree homomorphisms φ from $\mathit{struc}(C_2)$ to $\mathit{struc}(C_1)$ and ψ from $\mathit{struc}(C_3)$ to $\mathit{struc}(C_2)$ that are injective and satisfy the requirements of Definition 17. But then $\varphi \circ \psi$ is a tree homomorphism from $\mathit{struc}(C_3)$ to $\mathit{struc}(C_1)$ that is injective, because φ and ψ are, and satisfies the requirements of Definition 17, because φ and ψ do (function composition preserves the requirements). Hence, $C_1 \preceq C_3$. □

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.

Lemma 7. Let H be a well-defined class hierarchy and $D' = (d', \{d\}, A', M')$ be a subclass of $D = (d, \emptyset, A, M)$ in H. Then:

$$\mathit{flat}(D') \preceq \mathit{flat}(D).$$

Proof. The lemma follows from Definition 17, 5, and 9, the fact that $\mathit{atts}(D) \subseteq \mathit{atts}(D')$, and the fact that $\mathit{meths}(D) \subseteq \mathit{meths}(D')$. □

The lemma states that the subclass (**Isa**) relation is implicitly preserved by flattening.

Example 15. Let $\{C_p, C_e', C_d\}$ be the class hierarchy of Example 4. Then $\mathit{flat}(C_e')$ is a subclass of $\mathit{flat}(C_p)$. □

Example 16. 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 describe squares by the coordinates of the left upper corner and the width, and rectangles by the coordinates of the left upper corner and the width in both directions. According to the subclass order on flattened classes: $\mathit{flat}(C_r) \preceq \mathit{flat}(C_s)$, where C_r is (the abbreviation of) class 'Rectangle' and C_s is (the abbreviation of) class 'Square'. 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 in terms of 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 (abbreviations of) class hierarchies generated by the BNF-grammar of Section 2.

Definition 18. The set of well-defined flattened class hierarchies, denoted by $\mathcal{FH}$, is defined as:

$$\mathcal{FH} = \{\mathit{flat}(H) \mid H \in \mathcal{H} \wedge \mathit{Complete}(H) \wedge \mathit{Sound}(H)\}.$$

□

For the sequel, let H be a well-defined class hierarchy and F be its flattened form. Define $\mathcal{S}_F$ to be the set of classes that can be constructed from the classes in F.

Definition 19. 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'. Note that $\mathit{sups}(D)$ is finite for every $D \in F$. 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:

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

Note that $\mathcal{S}_F$ is finite, because F is finite and $\mathit{sups}(D)$ is finite for every $D \in F$. Finally, for every class $D = (d, A, M)$ in $\mathcal{S}_F - F$, let $\mathit{sups}(D)$ be the set of syntactic superclasses of D:

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

Note that $\mathit{sups}(D)$ is also finite for every $D \in \mathcal{S}_F - F$. □

Lemma 8. Let D be a class in $\mathcal{S}_F$. Then:

$$\forall C \in \mathit{sups}(D) : D \preceq C.$$

Proof. The lemma follows from Definition 19, 17 and 9. □

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

Definition 20. Define a relation $\cong$ on $\mathcal{S}_F$ as follows:

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

□

Lemma 9. Relation $\cong$ on $\mathcal{S}_F$ is an equivalence relation.
Proof. The lemma follows from Theorem 2. □

Lemma 10. Let C_1 and C_2 be classes in $\mathcal{S}_F$. If $C_1 \cong C_2$, then:

$$\forall D_1 \in \mathit{sups}(C_1)\ \exists D_2 \in \mathit{sups}(C_2) : D_1 \cong D_2.$$

Proof. If $C_1 \cong C_2$, then there is a bijective tree homomorphism φ from $\mathit{struc}(C_1)$ to $\mathit{struc}(C_2)$ that satisfies the requirements of Definition 17. Let $D_1 \in \mathit{sups}(C_1)$ be a syntactic superclass of C_1. Then $\mathit{struc}(D_1)$ is a subtree of $\mathit{struc}(C_1)$, except for the label of the root, and the image of the subtree under φ is a subtree of $\mathit{struc}(C_2)$ corresponding to a class $D_2 \in \mathit{sups}(C_2)$. Since φ is bijective and satisfies the requirements of Definition 17, we have that $D_1 \cong D_2$. □

Lemma 11. Let C_1, C_2, D_1, and D_2 be classes in $\mathcal{S}_F$. If $C_1 \cong D_1$ and $C_2 \cong D_2$, then:

$$C_1 \preceq C_2 \Leftrightarrow D_1 \preceq D_2.$$

Proof. The lemma follows from Definition 20 and Theorem 2. □

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:

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

Note that $\hat{\mathcal{S}}_F$ is finite, because $\mathcal{S}_F$ is finite. The subclass order on $\mathcal{S}_F$ induces a subclass order on $\hat{\mathcal{S}}_F$. According to Lemma 11, the induced subclass order is well-defined.

Definition 21. Define a subclass relation $\preceq$ on $\hat{\mathcal{S}}_F$ as follows:

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

□

Lemma 12. The subclass relation on $\hat{\mathcal{S}}_F$ is a partial order.
Proof. The lemma follows from Theorem 2 and Definition 20. □

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.

Definition 22. Let C_1 and C_2 be a pair of classes in $\mathcal{S}_F$. Define $\mathit{sups}(C_1, C_2)$ to be the set of superclasses of C_1 which are equivalent to superclasses of C_2:

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

□

Lemma 13. Let C_1 and C_2 be a pair of classes in $\mathcal{S}_F$. Then:

1. $\forall D \in \mathit{sups}(C_1, C_2)\ \exists D' \in \mathit{sups}(C_2, C_1) : D \cong D'$
2. $\forall D \in \mathit{sups}(C_2, C_1)\ \exists D' \in \mathit{sups}(C_1, C_2) : D \cong D'$.

Proof. The lemma follows immediately from Definition 22. □

The lemma states that both $\mathit{sups}(C_1, C_2)$ and $\mathit{sups}(C_2, C_1)$ can, up to equivalence, be regarded as the set of common superclasses of C_1 and C_2.

Now, we define a join operator. According to Definition 22, Lemma 10 and 11, the join operator is well-defined.

Definition 23. Define a binary operator $\sqcup_F$ on $\hat{\mathcal{S}}_F$ as follows:

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

□

Theorem 3. Let $[C_1]$ and $[C_2]$ be classes in $\hat{\mathcal{S}}_F$. Then:

1. $[C_1] \sqcup_F [C_2]$ is nonempty
2. $[C_1] \sqcup_F [C_2]$ contains the least elements of $\{[C] \mid C \in \mathit{sups}(C_1, C_2)\}$ w.r.t. $\preceq$.

Proof. The first item follows from the fact that $\mathit{sups}(C_1, C_2)$ contains at least one element, viz., $(d, \emptyset, \emptyset)$ for some $d \in CN$.

To proof the second item, let V be $\{C \mid C \in \mathit{sups}(C_1, C_2) \wedge \forall D \in \mathit{sups}(C_1, C_2)[D \preceq C \Rightarrow C \cong D]\}$. Furthermore, let $[D_1]$ be a class in $\{[C] \mid C \in \mathit{sups}(C_1, C_2)\}$ and $[D_2]$ be a class in $\{[C] \mid C \in V\}$. Suppose $[D_1] \preceq [D_2]$. From Lemma 11 it follows that there are classes $D_1' \in \mathit{sups}(C_1, C_2)$ and $D_2' \in V$, such that $D_1' \cong D_1$, $D_2' \cong D_2$, and $D_1' \preceq D_2'$. Since $D_2' \in V$, we have that $D_2' \cong D_1'$ and, hence, $[D_1] = [D_1'] = [D_2'] = [D_2]$. □

The theorem states that, given a pair of classes $[C_1]$ and $[C_2]$, operator $\sqcup_F$ defines the set of least common superclasses of $[C_1]$ and $[C_2]$. Hence, $\sqcup_F$ can be regarded as a kind of join operator. This join operator can be used to factorise classes, as follows.

Let D_1 and D_2 be classes in H. Every element $[C]$ of the set of least common superclasses of $[\mathit{flat}(D_1)]$ and $[\mathit{flat}(D_2)]$ gives, up to equivalence, one possibility to factorise D_1 and D_2: choose a member (d, A, M) of $[C]$, add $D = (d, \emptyset, A, M)$ to H, and redefine D_1 and D_2 to be subclasses of D. (In order to make $\mathit{flat}(D)$ a syntactic superclass of both $\mathit{flat}(D_1)$ and $\mathit{flat}(D_2)$, we have to replace attributes and methods in $\mathit{flat}(D_1)$ and $\mathit{flat}(D_2)$ by equivalent attributes and methods of class $\mathit{flat}(D)$, as follows. Since $C \cong C_1 = (c_1, A_1, M_1)$ for some $C_1 \in \mathit{sups}(\mathit{flat}(D_1))$ and $C \cong C_2 = (c_2, A_2, M_2)$ for some $C_2 \in \mathit{sups}(\mathit{flat}(D_2))$, there are bijections $\varphi_1 : \mathit{struc}(C_1) \to \mathit{struc}(\mathit{flat}(D))$ and $\varphi_2 : \mathit{struc}(C_2) \to \mathit{struc}(\mathit{flat}(D))$ that satisfy the requirements of Definition 17. We have to replace every attribute $a : T$ in $\mathit{flat}(D_i)$, such that $a : T \in A_i$, by the corresponding attribute of $\mathit{flat}(D)$ according to φ_i. Furthermore, we have to replace every method $m(P) = E$ in $\mathit{flat}(D_i)$, such that $m(P) = E \in M_i$, by the corresponding method $m'(P') = E'$ of $\mathit{flat}(D)$ according to φ_i, i.e., $\mathit{func}(C_i, m)[\varphi_i] =_{ext} \mathit{func}(\mathit{flat}(D), m')$.)

Example 17. Let H_d be the class hierarchy of Example 16. 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:

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

Hence, there is, up to equivalence, only one possibility to factorise C_r.

Now, let the designer choose a class from $[\mathit{flat}(C_s)]$. 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 $\mathit{struc}(\mathit{flat}(C_s))$ to $\mathit{struc}(\mathit{flat}(C_r))$. 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. In the following subsection, we will briefly describe partial factorisation. □

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.

Definition 24. The universe of class hierarchies modulo equivalence (w.r.t. F), denoted by $\widehat{\mathcal{FH}}_F$, is defined as:

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

Note that $\widehat{\mathcal{FH}}_F \subseteq \wp(\hat{\mathcal{S}}_F)$ is finite, because $\hat{\mathcal{S}}_F$ is finite. □

There is a natural order on class hierarchies modulo equivalence.

Definition 25. Define a subhierarchy relation $\preceq$ on $\widehat{\mathcal{FH}}_F$ as follows:

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

□

Note that the subhierarchy relation resembles the subclass relation: subhierarchy $\hat{F}_1$ is a refinement of superhierarchy $\hat{F}_2$.

Lemma 14. $(\widehat{\mathcal{FH}}_F, \preceq)$ is a complete lattice.
Proof. The subhierarchy relation on $\widehat{\mathcal{FH}}_F$ is a partial order, because $\supseteq$ is.

To proof that greatest lower bounds exist, let $\hat{F}_1$ and $\hat{F}_2$ be class hierarchies in $\widehat{\mathcal{FH}}_F$. Define MEET to be $F_1 \cup F_2$. Then MEET $\subseteq \mathcal{S}_F$ and MEET $\in \mathcal{FH}$, because classes still have a unique name (class names are unique within $\mathcal{S}_F$) and only refer to classes in MEET, attributes and methods still have a unique name within their class, and attributes and methods are still well-typed. It follows that MEET $\in \mathcal{FH}_F$. Hence, $\hat{F}_1 \cup \hat{F}_2 = \{[C] \mid C \in \text{MEET}\}$ is the greatest lower bound (meet) of $\hat{F}_1$ and $\hat{F}_2$ in $\widehat{\mathcal{FH}}_F$ w.r.t. $\supseteq$.

Since $\widehat{\mathcal{FH}}_F$ is finite and greatest lower bounds exist, least upper bounds exist as well [10]. Hence, $(\widehat{\mathcal{FH}}_F, \preceq)$ is a lattice. The lattice is complete, because $\widehat{\mathcal{FH}}_F$ is finite. □

Normalisation of class hierarchies is defined using the equivalence relation to identify classes and the join operator to factorise classes.

A class hierarchy modulo equivalence is in normal form if it has been factorised completely.

Definition 26. Let $\hat{F}_0 \in \widehat{\mathcal{FH}}_F$ be a class hierarchy modulo equivalence. Class hierarchy $\hat{F}_0$ is in normal form if it is closed under joins:

$$[C_1] \in \hat{F}_0 \wedge [C_2] \in \hat{F}_0 \;\Rightarrow\; [C_1] \sqcup_F [C_2] \subseteq \hat{F}_0.$$

Note that we use $\subseteq$, because $[C_1] \sqcup_F [C_2]$ yields a set. □

Hence, normal forms are the fixpoints in $\widehat{\mathcal{FH}}_F$ of $\varphi(X) = X \cup \bigcup\{Y_1 \sqcup_F Y_2 \mid Y_1, Y_2 \in X\}$, where $\bigcup\{Z_1, \cdots, Z_n\} = Z_1 \cup \cdots \cup Z_n$. The normal form of $\hat{F}$ is the greatest subhierarchy of $\hat{F}$ that is in normal form.

Definition 27. The normal form of $\hat{F}$, denoted by $nf(\hat{F})$, is the greatest fixpoint in $(\widehat{\mathcal{FH}}_F, \preceq)$ of the function $\varphi_F : \widehat{\mathcal{FH}}_F \to \widehat{\mathcal{FH}}_F$, defined as:

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

□

Lemma 15. The normal form of $\hat{F}$ is characterised by:

$$nf(\hat{F}) = \bigcup\{X \in \widehat{\mathcal{FH}}_F \mid X \subseteq \varphi_F(X)\}.$$

Proof. The lemma follows from [17], Lemma 14, and the fact that φ_F is monotone w.r.t. $\preceq$. □

There is another, stronger, characterisation of the normal form.

Theorem 4. The normal form of $\hat{F}$ is characterised by:

$$nf(\hat{F}) = \varphi_F^N(\emptyset) \quad \text{for some } N \in \mathbb{N}.$$

Proof. First, we will proof that there is an $N \in I\!N$, such that $\varphi_F^N(\emptyset)$ is a fixpoint of φ_F, and second, that this fixpoint is the greatest fixpoint of φ_F in $(\widehat{\mathcal{FH}}_F, \preceq)$.

Let $\#S$ be the number of elements of $\hat{\mathcal{S}}_F$. Then every element of $\widehat{\mathcal{FH}}_F$ has at most $\#S$ elements. From the definition of φ_F, it follows that $\varphi_F^i(\emptyset) \subseteq \varphi_F(\varphi_F^i(\emptyset))$ for every $i \in \{1, \cdots, \#S\}$. Now suppose that $\varphi_F^i(\emptyset) \neq \varphi_F(\varphi_F^i(\emptyset))$ for every $i \in \{1, \cdots, \#S\}$. Then it follows by induction that $\varphi_F^{\#S+1}(\emptyset)$ has at least $\#S + 1$ elements ($\varphi_F(\emptyset)$ is nonempty). This is in contradiction with the fact that every element of $\widehat{\mathcal{FH}}_F$ has at most $\#S$ elements. Hence, there must be some $i \in \{1, \cdots, \#S\}$, such that $\varphi_F^i(\emptyset) = \varphi_F(\varphi_F^i(\emptyset))$. Let N be the smallest natural number, such that this equality holds. Then $\varphi_F^N(\emptyset)$ is a fixpoint of φ_F.

Suppose that $\hat{G}$ is also a fixpoint of φ_F: $\varphi_F(\hat{G}) = \hat{G}$. From the fact that $\emptyset \subseteq \hat{G}$ and the fact that φ_F is monotone, it follows by induction that $\varphi_F^i(\emptyset) \subseteq \hat{G}$ for every $i \in \{1, \cdots, N\}$. In particular, $\hat{G} \supseteq \varphi_F^N(\emptyset)$. Hence, $\varphi_F^N(\emptyset)$ is the greatest fixpoint of φ_F in $(\widehat{\mathcal{FH}}_F, \preceq)$. □

It follows that the normal form of a class hierarchy modulo equivalence 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 fixpoint iteration can be used to normalise non-flattened class hierarchy H.

Algorithm 1. Let N be the smallest natural number such that $\mathit{nf}(\hat{F}) = \varphi_F^N(\emptyset)$. Compute a sequence of class hierarchies $H_0, \cdots, H_{N+1}$, a subclass relation R, and a normalised class hierarchy H_ω, as follows:

```
initialise $H_0$ by $\emptyset$
initialise $H_1$ by $H$
for $i$ from 1 to $N$
do initialise $H_{i+1}$ by $H_i$
   for every pair of classes $D_1$ and $D_2$ in $H_i - H_{i-1}$
   do 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 $H_{i+1}$
      add $(D_1, D)$ and $(D_2, D)$ to $R$
   od
od
initialise $H_\omega$ by $H_{N+1}$
for every pair of classes $(D_1, D_2)$ in $R$
do redefine $D_1$ to be a subclass of $D_2$ in $H_\omega$
od.
```

□

In the last step, we have to replace attributes and methods in class D_1 by equivalent attributes and methods of class D_2 (and, hence, rename attributes in methods in other classes that refer to D_1).

Theorem 5. The normalised class hierarchy of Algorithm 1, i.e., H_ω, is characterised by:

1. $\mathit{flat}(H_\omega)$ equals $\mathit{flat}(H_{N+1})$, modulo attribute and method equivalence
2. $\mathit{nf}(\hat{F}) \preceq \mathit{flat}(\widehat{H_{N+1}}) \preceq \hat{F}$.

Proof. The first item follows from the fact that $\mathit{flat}((d, S, A \cup A', M \cup M')) = \mathit{flat}((d, S \cup \{d'\}, A, M))$ if d' is the name of a class D' with attributes A' and methods M'.

The second inequality of the second item follows from the fact that $\hat{F} \subseteq \widehat{\mathit{flat}(H_1)}$ and the fact that $\widehat{\mathit{flat}(H_i)} \subseteq \widehat{\mathit{flat}(H_{i+1})}$ for every $i \in \{1, \cdots, N\}$. Hence, $\widehat{\mathit{flat}(H_{N+1})} \supseteq \hat{F}$.

The first inequality of the second item follows from the fact that $\widehat{\mathit{flat}(H_1)} \subseteq \hat{F} = \varphi_F(\emptyset)$ and the fact that, for every $i \in \{1, \cdots, N\}$:

$$\begin{aligned}
&\widehat{\mathit{flat}(H_{i+1})} \subseteq \\
&\widehat{\mathit{flat}(H_i)} \cup \bigcup\{[\mathit{flat}(D_1)] \sqcup_F [\mathit{flat}(D_2)] \mid D_1, D_2 \in H_i - H_{i-1}\} \subseteq \\
&\widehat{\mathit{flat}(H_i)} \cup \bigcup\{[C_1] \sqcup_F [C_2] \mid [C_1], [C_2] \in \widehat{\mathit{flat}(H_i)}\} \subseteq \\
&\varphi_F^i(\emptyset) \cup \bigcup\{Y_1 \sqcup_F Y_2 \mid Y_1, Y_2 \in \varphi_F^i(\emptyset)\} \subseteq \\
&\varphi_F(\varphi_F^i(\emptyset)),
\end{aligned}$$

where $\widehat{\mathit{flat}(H_i)} \subseteq \varphi_F^i(\emptyset)$ is used as induction hypothesis. That is, $\widehat{\mathit{flat}(H_i)} \subseteq \varphi_F^i(\emptyset)$ for every $i \in \{1, \cdots, N+1\}$. Hence, $\mathit{nf}(\hat{F}) = \varphi_F^{N+1}(\emptyset) \supseteq \widehat{\mathit{flat}(H_{N+1})}$. □

Example 18. Let H_d be the class hierarchy of Example 16. 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 16 and one of the classes 'Rectangle' of Example 17. □

Algorithm 1 can be adapted to allow for partial factorisation of classes by replacing the last two lines before before the first 'od' in Algorithm 1 by:

```
let the designer choose a class $(d', A', M')$ from $\mathit{sups}((d, A, M))$
if the designer agrees to factorise $D_1$ and $D_2$ by $D' = (d', \emptyset, A', M')$
then add $D'$ to $H_{i+1}$
     add $(D_1, D')$ and $(D_2, D')$ to $R$
fi.
```

If we normalise H according to the adapted algorithm, then, in fact, we normalise $H' = H \cup P$ according to the old algorithm with the possibility to skip the last two lines before the first 'od', where P is the set of partial factorisations:

$$\begin{aligned}
P = \{D \mid \exists i \in \{1, \cdots, N\}[D \in H_{i+1} \wedge \\
\forall D_1, D_2 \in H_i - H_{i-1} : [\mathit{flat}(D)] \notin [\mathit{flat}(D_1)] \sqcup_F [\mathit{flat}(D_2)]]\}.
\end{aligned}$$

The normalised class hierarchy of the adapted algorithm, i.e., H_ω, is characterised by:

1. $\mathit{flat}(H_\omega)$ equals $\mathit{flat}(H_{N+1})$, modulo attribute and method equivalence
2. $\mathit{nf}(\widehat{\mathit{flat}(H')}) \preceq \widehat{\mathit{flat}(H_{N+1})} \preceq \hat{F}$.

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 19. Let H_d be the class hierarchy of Example 16 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 according to Algorithm 1, we obtain something like the class hierarchy consisting of class ‘Point’ and the following classes:

```
Class Square Isa Point
Attributes
    width:int
Endclass
Class Rectangle Isa Square
Attributes
    width_y:int
Methods
    rotate = y_co := y_co + width;
        width := width_y − width;
        width_y := width_y − width;
        width := width + width_y
Endclass.
```

Another, more natural, solution would be to aggregate the coordinates of the left upper corner in class ‘Square’ and class ‘Rectangle’ of Example 16 into a new attribute ‘left_up:Point’. This is the subject of future research. □

Example 20. Let H_d be the class hierarchy of Example 16 and H_x be the class hierarchy consisting of the following class:

```
Class X
Attributes
    $a_1$:int
    $a_2$:int
    $a_3$:int
    $a_4$:int
Methods
    $m_1$($p_1$:int, $p_2$:int, $p_3$:int, $p_4$:int) =
        $a_1 := p_1$; $a_2 := p_2$; $a_3 := p_3$; $a_4 := p_4$
    $m_2$ ($p_1$:int, $p_2$:int) =
        $a_1 := a_1 + p_1$; $a_2 := a_2 + p_2$
    $m_3$ ($p_1$:int) =
        $a_3 := a_3 \times p_1$; $a_4 := a_4 \times p_1$
        $m_4$ =
            $a_2 := a_2 + a_3$; $a_3 := a_4 - a_3$;
            $a_4 := a_4 - a_3$; $a_3 := a_3 + a_4$
    Endclass.
```

If we integrate H_d and H_x according to Algorithm 1, we obtain something like the following class hierarchy:

```
    Class Fig1
    Attributes
        x_left_up:int
        y_left_up:int
        width:int
    Methods
        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 Square Isa Fig1
    Methods
        set (x:int, y:int) =
            x_left_up := x; y_left_up := y
    Endclass
    Class Fig2 Isa Fig1
    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
    Class Rectangle Isa Square Fig2
    Endclass
    Class X Isa Fig2
    Methods
        $m_1$($p_1$:int, $p_2$:int, $p_3$:int, $p_4$:int) =
            x_left_up := $p_1$; y_left_up := $p_2$;
            width := $p_3$; width_y := $p_4$
        $m_3$ ($p_1$:int) =
            width := width × $p_1$; width_y := width_y × $p_1$
    Endclass.
```

□

The existing methods for schema integration in relational and semantic databases can also be used to integrate object-oriented database schemas, as follows: compare classes by their attributes only, checking for homonyms, synonyms, type inconsistencies, integrity constraint conflicts, redundancy conflicts, and differences in abstraction level.

Consider, e.g., class 'Rectangle' and class 'X' of Example 20 and ignore their methods. Checking for synonyms can help to factorise 'Rectangle' and 'X', because all attributes have a different name, but checking for structural conflicts will not help, because all attributes have the same simple type. But deciding whether two different names (from different contexts) are synonyms is a difficult problem. Therefore, the use of existing approaches is limited in this case. Since existing approaches do not consider methods and our approach is based on behaviour of methods, our approach can be used as an extension to existing ones.

6 Conclusion

This report develops a formal approach to integrate class hierarchies, based on a synthetic subclass order. The result is a natural framework for integration of class hierarchies, where classes are identified using an equivalence relation induced by the subclass order 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 towards 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 data model (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 report.

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. Memoranda Informatica 90-75, University of Twente, Enschede, The Netherlands, 1990.

[4] H. Balsters and C. de Vreeze. A formal theory of sets in object-oriented contexts. Memoranda Informatica 90-74, 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. Memoranda Informatica 90-76, 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] A. Macintyre. The laws of exponentation. In *Model Theory and Arithmetic, LNM 890*, pages 185–197. Springer-Verlag, Berlin, 1979.

[16] 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.

[17] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, pages 285–309, 1955.

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

A Decidability of extensional equality

Theorem. There is a recursive procedure to decide whether two functions F and G in our language are extensionally equal.

Proof. The theorem follows from an induction argument.

If F and G are integer-valued functions, then F and G are polynomials. It follows from [15] that it is decidable whether F and G are extensionally equal.

If F and G are string-valued functions, then the bodies of F and G are concatenations of variables and constants. Claim A: Let $H_1, \cdots, H_n$ be string-valued functions.

$$\begin{array}{ll} \text{If} & \forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow \tilde{H}_i \not\equiv \tilde{H}_j], \\ \text{then} & \exists \vec{x}[\forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow \tilde{H}_i(\vec{x}) \neq \tilde{H}_j(\vec{x})]], \end{array}$$

where $\tilde{H}$ is the normal form of H according to the following rewrite rules:

$$\begin{array}{lcl} \epsilon + w & \rightarrow & w \\ w + \epsilon & \rightarrow & w \\ c + d & \rightarrow & concat(c,d), \end{array}$$

where ϵ is the empty string, w is a string expression, c and d are string constants, and $concat(c,d)$ is the concatenation of c and d. It follows that F and G are extensionally equal if and only if their normal forms are syntactically equal, which is decidable.

If F and G are record-valued (or pointer-valued) functions, then the bodies of F and G are variables. It follows that F and G are extensionally equal if and only if they are syntactically equal, which is decidable.

If F and G are set-valued functions, then the bodies of F and G are unions of a variable and a set of expressions:

$$\begin{array}{l} F \equiv \lambda\vec{x} : \vec{\tau}.y_{i_F} \cup \{F_1(\vec{z}),\cdots,F_l(\vec{z})\} \\ G \equiv \lambda\vec{x} : \vec{\tau}.y_{i_G} \cup \{G_1(\vec{z}),\cdots,G_m(\vec{z})\}, \end{array}$$

where $\vec{x} \equiv \vec{y}, \vec{z}$ and the F_i (respectively, the G_i) are mutually distinct. Then:

$$\begin{array}{l} \forall\vec{x}[F(\vec{x}) = G(\vec{x})] \Rightarrow \\ \forall\vec{z}[\{F_1(\vec{z}),\cdots,F_l(\vec{z})\} = \{G_1(\vec{z}),\cdots,G_m(\vec{z})\}] \Rightarrow \\ \forall i \in \{1,\cdots,l\}\ \forall\vec{z}[F_i(\vec{z}) = G_1(\vec{z}) \vee \cdots \vee F_i(\vec{z}) = G_m(\vec{z})]. \end{array}$$

Claim B: Let $H_1,\cdots,H_n$ be functions in our language.

$$\begin{array}{ll} \text{If} & \forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow H_i \neq H_j], \\ \text{then} & \exists \vec{x}[\forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow H_i(\vec{x}) \neq H_j(\vec{x})]]. \end{array}$$

Suppose F_i is distinct from every function in $\{G_1,\cdots,G_m\}$. According to Claim B:

$$\exists\vec{z}[F_i(\vec{z}) \neq G_1(\vec{z}) \wedge \cdots \wedge F_i(\vec{z}) \neq G_m(\vec{z})].$$

Contradiction. Hence, F_i is equal to one of the functions in $\{G_1,\cdots,G_m\}$. It follows that F and G are extensionally equal if and only if:

1. $y_{i_F} \equiv y_{i_G}$
2. $\forall i \in \{1,\cdots,l\}\exists j \in \{1,\cdots,m\}[\lambda\vec{x} : \vec{\tau}.F_i(\vec{x}) = \lambda\vec{x} : \vec{\tau}.G_j(\vec{x})]$
3. $\forall j \in \{1,\cdots,m\}\exists j \in \{1,\cdots,l\}[\lambda\vec{x} : \vec{\tau}.F_i(\vec{x}) = \lambda\vec{x} : \vec{\tau}.G_j(\vec{x})]$.

Using the induction hypothesis, it follows that extensional equality of set-valued functions is decidable.

Proof of Claim B. The claim follows from an induction argument.

If $H_1,\cdots,H_n$ are integer-valued functions, then they are polynomials. For every $i,j \in \{1,\cdots,n\}$, define $D_{(i,j)}$ to be $H_i - H_j$. Then:

$$\begin{array}{l}
\forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow H_i \neq H_j] \Rightarrow \\
\forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow D_{(i,j)} \neq 0] \Rightarrow \\
\Pi_{i\neq j} D_{(i,j)} \neq 0 \Rightarrow \\
\exists \vec{x}[(\Pi_{i\neq j} D_{(i,j)})(\vec{x}) \neq 0] \Rightarrow \\
\exists \vec{x}[\Pi_{i\neq j}(D_{(i,j)}(\vec{x})) \neq 0] \Rightarrow \\
\exists \vec{x}[\forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow D_{(i,j)}(\vec{x}) \neq 0]] \Rightarrow \\
\exists \vec{x}[\forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow H_i(\vec{x}) \neq H_j(\vec{x})]].
\end{array}$$

If $H_1,\cdots,H_n$ are string-valued functions, then Claim B follows from Claim A and the fact that a string-valued function is extensionally equal to its normal form.

If $H_1,\cdots,H_n$ are record-valued (or pointer-valued) functions, then Claim B follows from the fact that the body of a record-valued (or pointer valued) function is a variable.

If $H_1,\cdots,H_n$ are set-valued functions, then their bodies are unions of a variable and a set of expressions. Let H_i be $\lambda\vec{x} : \vec{\tau}.y_i \cup \{h_{(i,1)}(\vec{z}),\cdots,h_{(i,n_i)}(\vec{z})\}$, where $\vec{x} \equiv \vec{y},\vec{z}$. Define:

$$\begin{array}{l}
H'_i = \{h_{(i,1)},\cdots,h_{(i,n_i)}\} \\
H'_i(\vec{z}) = \{h_{(i,1)}(\vec{z}),\cdots,h_{(i,n_i)}(\vec{z})\} \\
H = H'_1 \cup \cdots \cup H'_n.
\end{array}$$

Let H be $\{h_1,\cdots,h_p\}$, such that $\forall i,j \in \{1,\cdots,p\}[i \neq j \Rightarrow h_i \neq h_j]$. Using the induction hypothesis, it follows that:

$$\exists \vec{z}[\forall i,j \in \{1,\cdots,p\}[i \neq j \Rightarrow h_i(\vec{z}) \neq h_j(\vec{z})]].$$

Hence, $\exists \vec{z}[\forall i,j \in \{1,\cdots,n\}[H'_i \neq H'_j \Rightarrow H'_i(\vec{z}) \neq H'_j(\vec{z})]]$. Then:

$$\begin{array}{l}
\forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow H_i \neq H_j] \Rightarrow \\
\forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow (y_i \not\equiv y_j \vee H'_i \neq H'_j)] \Rightarrow \\
\exists \vec{z}[\forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow (y_i \not\equiv y_j \vee H'_i(\vec{z}) \neq H'_j(\vec{z}))]] \Rightarrow \\
\exists \vec{x}[\forall i,j \in \{1,\cdots,n\}[i \neq j \Rightarrow H_i(\vec{x}) \neq H_j(\vec{x})]].
\end{array}$$

Proof of Claim A

Let N be the length of $\vec{x}$ and L be the length of the longest string constant in any of the $\tilde{H}_i$. Let K be the maximum of N and $\lceil {}^2\log L \rceil$, $\pi(i)$ be 2^{K+i}, and $\rho(i)$ be 2^{N-i}. Furthermore, let a and b be different string constants of length 1. For every $i \in \{1,\cdots,N\}$, define:

$$v_i = (concat(a^{\pi(i)}, b^{\pi(i)}))^{\rho(i)},$$

where $c^0 = \epsilon$ and $c^{k+1} = concat(c, c^k)$. For every pair of string constants c and d of length at most L, $concat(c, v_i)$ is a prefix of $concat(d, v_j)$ if and only if $i = j$ and $c = d$. Let $\vec{v}$ be $v_1,\cdots,v_N$. Then:

$$\forall i,j \in \{1,\cdots,n\}[\tilde{H}_i \not\equiv \tilde{H}_j \Rightarrow \tilde{H}_i(\vec{v}) \neq \tilde{H}_j(\vec{v})].$$

□