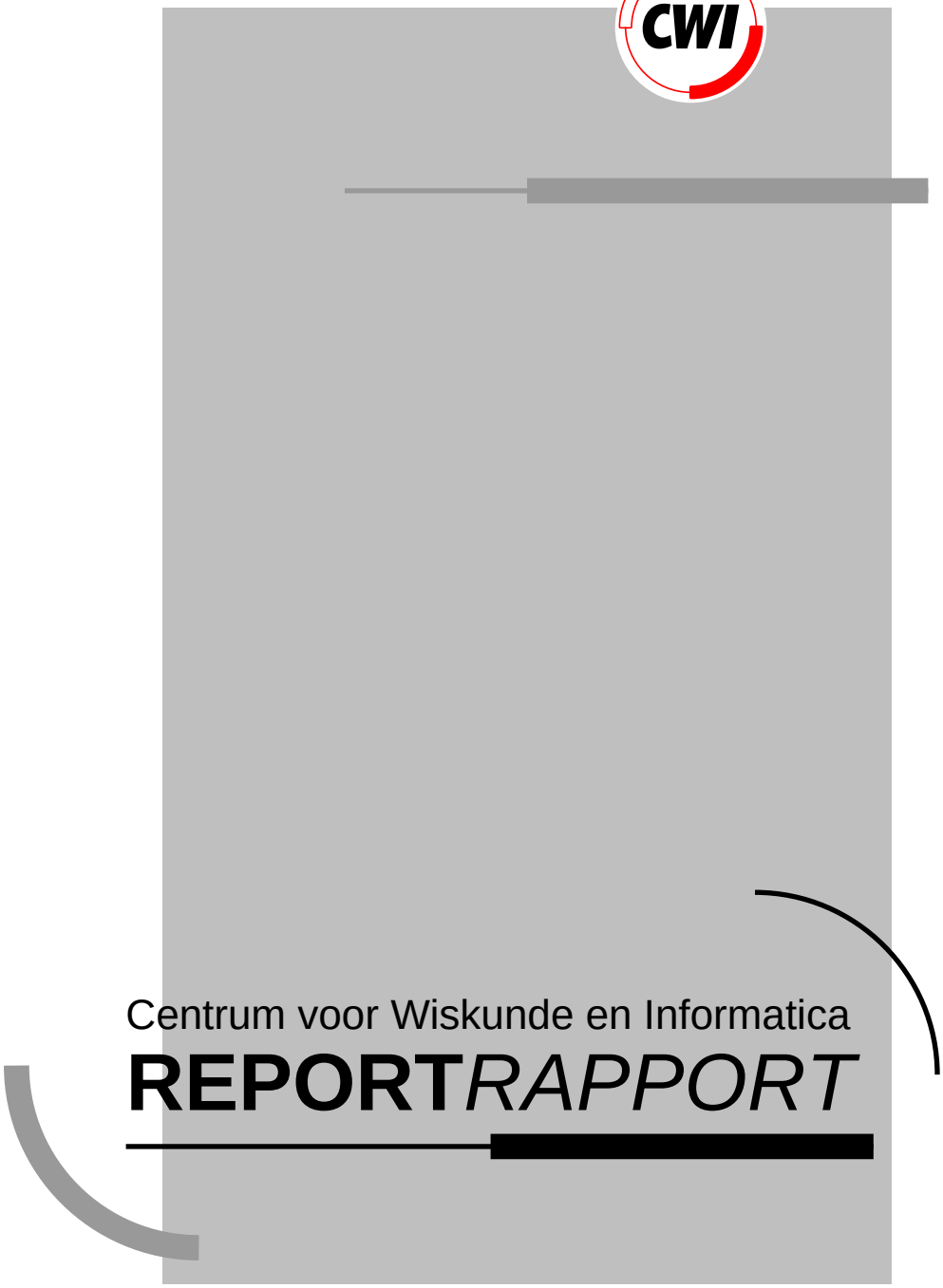


Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Schema refinement and schema integration in object-oriented databases

C. Thieme, A. Siebes

Computer Science/Department of Algorithmics and Architecture

CS-R9354 1993

Schema Refinement and 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 to compare classes. Classes are integrated in a natural way using a join operator w.r.t. the subclass order. In contrast with existing literature, our subclass order compares classes not only by their attributes, but also by the behaviour of their methods, leading to a more semantic approach towards schema integration in object-oriented databases.

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

Database design is a complex, iterative process consisting of several activities, including conceptual design and implementation design [15]. 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 [14], which includes guidelines, techniques, methods, and tools to support the activities of the designer.

This report addresses the problem of integrating classes in class hierarchies, which are the database schemas in object-oriented data models [1, 13, 3, 11]. A solution to this problem can be used to support integration of different user views in the conceptual design phase or schema optimisation in the implementation design phase of an object-oriented design methodology [10], 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.

Optimisation 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. In Section 2, class hierarchies are introduced and classes are described in terms of types and functions, similar to TM/FM [3], which is based on Cardelli's work on subtyping [7]. Furthermore, attribute specialisation and method specialisation are defined. In Section 3, a synthetic subclass order is introduced to compare classes. The subclass order is defined in terms of a weak subtype relation and a weak subfunction relation. In Section 4, a join operator w.r.t. the subclass order is introduced to integrate classes. The join operator is defined in terms of the join of the weak subtype relation and the join of the weak subfunction relation. Finally, in Section 5, the results are recapitulated and directions for future research are given.

2 Class Hierarchies

In this section, we introduce class hierarchies, similar to class hierarchies in Galileo [1], Goblin [11], O_2 [13], and TM/FM [3]. 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 refers to 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.

Definition 1. First, five disjoint sets are postulated: a set CN of class names, a set AN of attribute names, a set MN of method names, a set L of labels, and a set $Cons$ of basic constants (i.e., 'integer', 'rational', and 'string' constants). The sets are generated by the nonterminals CN, AN, MN, L, and Cons, respectively. Class hierarchies are the sentences of the following BNF-

grammar, where the plus sign ($^+$) denotes a finite, nonempty, repetition, square brackets ([ ]) denote an option, and the vertical bar (|) denotes a choice:

Hierarchy	::=	Class$^+$
Class	::=	'**Class**' CN ['**Isa**' CN$^+$]
		['**Attributes**' Att$^+$]
		['**Methods**' Meth$^+$]
		'**Endclass**'
Att	::=	AN ':' Type
Type	::=	BasType | SetType | RecType | CN
BasType	::=	'integer' | 'rational' | 'string'
SetType	::=	'{' Type '}'
RecType	::=	'<' FieldList '>'
FieldList	::=	Field | Field ',' FieldList
Field	::=	L ':' Type
Meth	::=	MN ['(' ParList ')'] '=' AsnList |
		MN '\' MN ['(' ParList ')'] '=' AsnList
ParList	::=	Par | Par ',' ParList
Par	::=	L ':' BasType
AsnList	::=	Assign | Assign ';' AsnList
Assign	::=	Dest ':=' Source | '**insert**(' Source ',' Dest ')'
Dest	::=	AN | Dest '.' L
Source	::=	Term | Term '+' Source | Term '−' Source |
		Term '×' Source | Term '÷' Source
Term	::=	Var | Cons
Var	::=	L | AN | Var '.' L | Var '.' AN

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

A class hierarchy is well-defined if it satisfies three 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 have a unique name within their class and are well-typed (see subsection on attributes). The third is that methods have a unique name within their class and are well-typed (see subsection on methods).

2.1 Attributes

In this subsection, we consider classes with attributes only. We define underlying types of classes, a subtype relation on underlying types, and attribute specialisation.

Example 1. The following class hierarchy introduces a class 'SimpleAddress' and a class 'Address', which inherits from class 'SimpleAddress' and adds a new attribute.

Class SimpleAddress
Attributes
  house : integer
  street : string
  city : string
Endclass
Class Address **Isa** SimpleAddress
Attributes
  country : string
Endclass.

□

In definitions, we abbreviate every class to a 4-tuple $C = (c, S, A, M)$, where c is the name of the class, S is the set of (names of) superclasses, A is the set of new attributes, and M is the set of new methods. The name of (abbreviated) class C is denoted by $name(C)$. Furthermore, we abbreviate a class hierarchy to the set of abbreviations of the classes in the class hierarchy.

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

Definition 2. Let H be a class hierarchy satisfying the first constraint and $C = (c, S, A, M)$ be a class in H. The set of all attributes of C, denoted by $atts(C)$, is defined as:

$$atts(C) = A \cup \{a : T \mid \exists C' \in H\ [name(C') \in S \wedge a : T \in atts(C')] \wedge \forall a' : T' \in A[a \neq a']\}.$$

Note that redefined attributes override the corresponding attributes in the superclass. □

Every class in a class hierarchy has an underlying type, which describes the structure of the class, i.e., the structure of the objects in its extensions (cf. TM/FM [3, 4]). Informally, the underlying type of a class is an aggregation of its attributes, where recursive types [2] are used to cope with attributes that refer to classes.

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

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

where

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

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

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

Example 2. The underlying types of class 'SimpleAddress' and class 'Address' of Example 1 are given by:

$$\begin{array}{l}
\tau_S = \mu\, t_S\ .\ \text{<id:oid, house:integer, street:string, city:string>} \\
\quad = \text{<id:oid, house:integer, street:string, city:string>}, \\
\tau_A = \mu\, t_A\ .\ \text{<id:oid, house:integer, street:string, city:string, country:string>} \\
\quad = \text{<id:oid, house:integer, street:string, city:string, country:string>}.
\end{array}$$

Using the definition and the rule $\mu t.\alpha = \alpha[t \setminus \mu t.\alpha]$, where $\alpha[t \setminus e]$ means that every occurrence of t in α must be replaced by e, we have obtained the same underlying types as in TM/FM. □

A natural subtype relation for underlying types is the following [7]: τ is a subtype of τ' if τ has at least the fields of τ' and the fields of τ are at least as specialised as the fields of τ'.

Definition 4. Let underlying types τ and τ' (using the rule $\mu t.\alpha = \alpha[t \setminus \mu t.\alpha]$) be given by:

$$\begin{array}{l}
\tau = < l_i : \tau_i \mid i \in I >, \\
\tau' = < l_i : \tau'_i \mid i \in I' >.
\end{array}$$

Then τ is a subtype of τ', denoted by $\tau \leq \tau'$, if

$$I \supseteq I' \wedge \forall i \in I'[\tau_i \leq \tau'_i]$$

where a basic type is a subtype of itself, a set type is a subtype of another set type if the first element type is a subtype of the second element type, and a recursive type is a subtype of another recursive type if from the assumption that the first type variable is a subtype of the second type variable it follows that the body of the first type is a subtype of the body of the second type [7, 2]:

$$B \leq B \text{ if } B \in \{\text{integer}, \text{rational}, \text{string}\},$$
$$\{v\} \leq \{v'\} \text{ if } v \leq v',$$
$$\mu t.\alpha \leq \mu t'.\alpha' \text{ if } t \leq t' \Rightarrow \alpha \leq \alpha'.$$

□

For example, τ_A is a subtype of τ_S.

Attribute specialisation is a form of attribute redefinition, where an inherited attribute $a : T$, defined in class C, is replaced by $a : T'$ in subclass C', such that $type(T')$ is a subtype of $type(T)$. If attribute specialisation is allowed, it still holds that the underlying type of a subclass is a subtype of the underlying type of the superclass (in case of general redefinition of attributes, it does not hold). Now we can reformulate the second constraint for well-defined class hierarchies: every attribute must have a unique name within its class, the type of every attribute must be well-defined[1], and the type of every inherited attribute must be a subtype of the type of the corresponding attribute in the superclass.

Example 3. The following class hierarchy introduces a class 'Person' and a class 'Employee', which specialises inherited attribute 'holiday_address' and adds attributes 'company' and 'salary'.

Class Person
Attributes
name : string
mother : Person
address : SimpleAddress
holiday_address : SimpleAddress
Endclass
Class Employee **Isa** Person
Attributes
holiday_address : Address
company : string
salary : integer
Endclass.

Let α be <id:oid, name:string, mother:t_P, address:τ_S, holiday_address:τ_S >, where t_P is a type variable and τ_S is the underlying type of class 'Simple_Address'. The underlying types of class 'Person' and class 'Employee' are given by:

$\tau_P = \mu\, t_P\, .\, \alpha$
$= <\text{id:oid, name:string, mother:}\mu\, t_P.\alpha, \text{ address:}\tau_S, \text{ holiday\_address:}\tau_S>,$
$\tau_E = \mu\, t_E\, .\, <\text{id:oid, name:string, mother:}\mu\, t_P.\alpha, \text{ address:}\tau_S, \text{ holiday\_address:}\tau_A,$
company:string, salary:integer>
$= <\text{id:oid, name:string, mother:}\mu\, t_P.\alpha, \text{ address:}\tau_S, \text{ holiday\_address:}\tau_A,$
company:string, salary:integer>,

where τ_S and τ_A are the underlying types of class 'Simple_Address' and class 'Address'. Using the definition and the rule $\mu t.\alpha = \alpha[t \setminus \mu t.\alpha]$, we have obtained underlying types that differ from the underlying types in TM/FM, which are <id:oid, name:string, mother:oid, address:oid, holiday_address:oid> and <id:oid, name:string, mother:oid, address:oid, holiday_address:oid, company:string, salary:integer>, respectively. Note that τ_E is a subtype of τ_P. □

[1]This boils down to the requirement that all labels in a record type must differ from each other.

2.2 Methods

In this subsection, we consider classes with attributes and methods. We define functional forms of methods, a subfunction relation on functional forms, and method specialisation.

Example 4. The following class hierarchy introduces a class 'Person1' with a method 'change_addr' and a class 'Person2' with a method 'move'.

Class Person1
Attributes
name : string
addr : <house:integer,street:string>
Methods
change_addr (h:integer,s:string) =
addr.house := h;
addr.street := s
Endclass
Class Person2
Attributes
name : string
address : <house:integer,street:string,city:string>
Methods
move (h:integer,s:string,c:string) =
address.house := h;
address.street := s;
address.city := c
Endclass.

□

Informally, the set of all methods of a class in a class hierarchy consists of both the new and inherited methods.

Definition 5. Let H be a class hierarchy satisfying the first constraint and $C = (c, S, A, M)$ be a class in H. The set of all methods of C, denoted by $meths(C)$, is defined as:

$$meths(C) = M \cup \{m(P) = E \mid \exists C' \in H\ [name(C') \in S \wedge m(P) = E \in meths(C')] \wedge \forall m'(P') = E' \in M[m \neq m']\}.$$

Note that redefined methods override the corresponding methods in the superclass. □

Every class in a class hierarchy has a set of functional forms (one for each of its methods), which describe the way in which the objects in the extensions of the class are manipulated (cf. TM/FM [3, 8]). Informally, the functional form of a method is a function whose body is an accumulation of the assignments in the method body.

Definition 6. Let H be a class hierarchy satisfying the first and second constraint and C be a class in H. Furthermore, let $\{a_1 : T_1, \cdots, a_k : T_k\}$ be $atts(C)$ and $m(P) = E$ be an element of $meths(C)$. The functional form of method $m(P) = E$ in class C, denoted by $func(C, m(P) = E)$, is defined as:

$$func(C, m(P) = E) = \lambda self : type(C)\ \lambda P\ .\ eval(E)(< \mathrm{id} = self.\mathrm{id}, a_1 = \sigma_0(a_1, T_1), \cdots, a_k = \sigma_0(a_k, T_k) >)$$

where

$\sigma_0(r, d) = self.r$ if $d \in CN$,
$\sigma_0(r, B) = self.r$ if $B \in \{\mathrm{integer}, \mathrm{rational}, \mathrm{string}\}$,
$\sigma_0(r, \{U\}) = self.r$,
$\sigma_0(r, < l_1 : U_1, \cdots, l_n : U_n >) = < l_1 = \sigma_0(r.l_1, U_1), \cdots, l_n = \sigma_0(r.l_n, U_n) >$,

and $eval(L_1; L_2)(\sigma) = eval(L_2)(eval(L_1)(\sigma))$, and $eval(d := s)(\sigma)$ is obtained from state σ by replacing the value of destination d by the value of source s in state σ, and $eval(\mathbf{insert}(s, d))(\sigma)$ is obtained from state σ by replacing the value of destination d by the union of the value of destination d and the set consisting of the value of source s in state σ. □

Example 5. Let C_{P1} be class 'Person1' and C_{P2} be class 'Person2' of Example 4. Let $meth_1$ be method 'change_addr' in class 'Person1' and $meth_2$ be method 'move' in class 'Person2'. The functional forms of method 'change_addr' in class 'Person1' and method 'move' in class 'Person2' are given by:

$$
\begin{aligned}
&func(C_{P1}, meth_1) = \lambda self:\tau_{P1}\ \lambda \text{h:int}\ \lambda \text{s:string}\ .\\
&\quad <\text{id}=self.\text{id}, \text{name}=self.\text{name}, \text{addr}=<\text{house=h,street=s}>>,\\
&func(C_{P2}, meth_2) = \lambda self;\tau_{P2}\ \lambda \text{h:int}\ \lambda \text{s:string}\ \lambda \text{c:string}\ .\\
&\quad <\text{id}=self.\text{id}, \text{name}=self.\text{name}, \text{address}=<\text{house=h,street=s,city=c}>,\\
&\qquad \text{company}=self.\text{company}, \text{salary}=self.\text{salary}>,
\end{aligned}
$$

where τ_{P1} is the underlying type of class 'Person1' and τ_{P2} is the underlying type of class 'Person2'. □

A natural subfunction relation for functional forms is the following: f is a subfunction of f' if f, projected onto the domain of f', is extensionally equal to f (modulo a permutation of parameters).

Definition 7. Let functional forms f and f' be given by:

$$
\begin{aligned}
f &= \lambda self:\tau\ \lambda P.\ <\text{id} = self.\text{id}, a_1 = e_1, \cdots, a_k = e_k>\\
f' &= \lambda self:\tau'\ \lambda P'.\ <\text{id} = self.\text{id}, a_1 = e'_1, \cdots, a_{k'} = e'_{k'}>,
\end{aligned}
$$

where $\tau =< \text{id} : \text{oid}, a_1 : \tau_1, \cdots, a_k : \tau_k >$ is a subtype of $\tau' =< \text{id} : \text{oid}, a_1 : \tau'_1, \cdots, a_k : \tau'_{k'} >$. We define the projection of f onto τ', provided it exists, as:

$$
\tilde{f} = \lambda self:\tau'\ \lambda \tilde{P}.\ <\text{id} = self.\text{id}, a_1 = \tilde{e}_1, \cdots, a_{k'} = \tilde{e}_{k'}>,
$$

where projection $\tilde{e}_i = proj(e_i, \tau'_i)$ is given by:

$$
\begin{aligned}
&proj(self.a.l_1.\cdots.l_n, \sigma) = self.a.l_1.\cdots.l_n\\
&\qquad \text{if } \tau'.a.l_1.\cdots.l_n \text{ exists and } \tau'.a.l_1.\cdots.l_n \leq \sigma,\\
&proj(c, \sigma) = c \ \text{ if } c \in Cons,\\
&proj(e_1 \theta e_2, \sigma) = proj(e_1, \sigma)\theta proj(e_2, \sigma) \ \text{ if } \theta \in \{+, -, \times, \div\},\\
&proj(g \cup \{g_1, \cdots, g_n\}, \{\sigma\}) = proj(g, \{\sigma\}) \cup \{proj(g_1, \sigma), \cdots, proj(g_n, \sigma)\},\\
&proj(< l_1 = g_1, \cdots, l_n = g_n >, < l_1 : \sigma_1, \cdots, l_{n'} : \sigma_{n'} >) =\\
&\qquad\qquad < l_1 = proj(g_1, \sigma_1), \cdots, l_{n'} = proj(g_{n'}, \sigma_{n'}) >,
\end{aligned}
$$

and $\tilde{P}$ is obtained from P by removing parameters that do not occur in the body of $\tilde{f}$. Then $\tilde{f}$ is extensionally equal to f' modulo a permutation of parameters, denoted by $\tilde{f} =_{ext} f'$, if and only if there exists a permutation Q' of P', such that

$$
\forall i \in \{1, \cdots, k'\}\ [\lambda self:\tau'\ \lambda \tilde{P}.\tilde{e}_i = \lambda self:\tau'\ \lambda Q'.e'_i],
$$

where $=$ denotes ordinary extensional equality. Finally, f is a subfunction of f', denoted by $f \leq f'$, if and only if $\tilde{f} =_{ext} f'$. □

Lemma 1. For the functional forms of the methods generated by the BNF-grammar of Definition 1, $=_{ext}$ and $\leq$ are decidable.

Proof. The decidability of $=_{ext}$ is proven in [16] and the decidability of $\leq$ follows from the decidability of $=_{ext}$, the fact that the subtype relation is decidable, and the fact that *proj* is computable. □

Method specialisation is a form of method redefinition, where an inherited method $m(P) = E$, defined in class C, is replaced by $m(P') = E'$ in subclass C', such that $func(C', m(P') = E')$ is a subfunction of $func(C, m(P) = E)$. Now we can reformulate the third constraint: every method must have a unique name within its class, every method must be well-typed[2], and the functional form of every inherited method must be a subfunction of the functional form of the corresponding method in the superclass.

Example 6. The following class hierarchy introduces a class 'Person' and a class 'Employee', which specialises inherited attribute 'addr' and specialises inherited method 'change_addr'.

Class Person
Attributes
  name : string
  addr : <house:integer,street:string>
Methods
  change_addr (h:integer,s:string) =
    addr.house := h;
    addr.street := s
Endclass
Class Employee **Isa** Person
Attributes
  addr : <house:integer,street:string,city:string>
Methods
  change_addr\move (h:integer,s:string,c:string) =
    addr.house := h;
    addr.street := s;
    addr.city := c
Endclass.

The method in class 'Employee' specialises the method inherited from class 'Person' and renames it to 'move'. The renaming construction is used to allow for more than one specialisation of the same method. □

3 Comparison of Classes

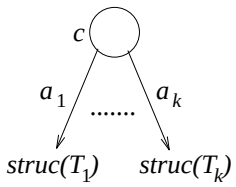
In this section, we introduce a synthetic subclass order to compare classes on the basis of syntactic and semantic similarity. The subclass order is defined in terms of a weak subtype relation on underlying types of classes and a weak subfunction relation on functional forms of methods.

3.1 Comparison of Attributes

The weak subtype relation is defined in terms of graph homomorphisms between trees representing underlying types. The trees are similar to the trees of [12]. However, instead of arrows labeled 'id' and nodes labeled 'oid', we introduce class nodes labeled by a class name.

Definition 8. Let H be a well-defined class hierarchy and C be a class in H. Furthermore, let c be $name(C)$ and $\{a_1 : T_1, \cdots, a_k : T_k\}$ be $atts(C)$. The tree representing the underlying type of class C, denoted by $struc(C)$, is defined as:

[2]This boils down to the requirement that, for every assignment, the type of the source must be a subtype of the type of the destination [16].



where

$struc(d) = struc(D)$ if $\exists D \in H[name(D) = d]$,
$struc(B)$ has only one node, labeled B, if $B \in \{\text{integer}, \text{rational}, \text{string}\}$,
$struc(\{U\})$ consists of a root, labeled by $\{\}$, a subtree $struc(U)$,
and an unlabeled arrow from the root labeled by $\{\}$ to the root of $struc(U)$,
$struc(< l_1 : U_1, \cdots, l_n : U_n >)$ consists of a root, labeled by $<>$,
subtrees $struc(U_1), \cdots, struc(U_n)$, and arrows, labeled l_i, one for each
$i \in \{1, \cdots, n\}$, from the root labeled by $<>$ to the root of $struc(U_i)$.

□

Note that the tree representing the underlying type of a recursive class is infinite.

Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H, and φ be a graph homomorphism from $struc(C_1)$ to $struc(C_2)$. If φ maps the root of $struc(C_1)$ to the root of $struc(C_2)$, then and only then we say that φ is a tree homomorphism. If φ maps labels to themselves, i.e., if for every node or arrow q in $struc(C_1)$, the following holds:

$$label(q) = l \Rightarrow label(\varphi(q)) = l,$$

where $label(q)$ denotes the label of node or arrow q, then and only then we say that φ preserves labels. And if φ maps classes to classes, i.e., if for every node n in $struc(C_1)$, the following holds:

$$label(n) \in CN \Rightarrow label(\varphi(n)) \in CN,$$

then and only then we say that φ is faithful with respect to classes. Finally, if φ 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 r_1 and for every node n_2 in $struc(C_1)$ with outgoing arrow r_2, the following holds:

$$(label(n_1) = label(n_2) \in CN \wedge label(r_1) = label(r_2) \wedge label(\varphi(n_1)) = label(\varphi(n_2))) \Rightarrow$$
$$label(\varphi(r_1)) = label(\varphi(r_2)),$$

then and only then we say that φ is faithful with respect to attributes. The following property will be used to relate any attribute $a_1 : T_1$ in class C_1 to some unique attribute $a_2 : T_2$ in class C_2, such that $type(T_2)$ is a weak subtype of $type(T_1)$.

Definition 9. Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H, and φ be a graph homomorphism from $struc(C_1)$ to $struc(C_2)$. Then φ is a subtype morphism if and only if it

1. is an injective tree homomorphism
2. preserves labels, except class names and attribute names
3. is faithful with respect to classes.

□

If there exists a subtype morphism from $struc(C_1)$ to $struc(C_2)$, then and only then we say that $type(C_2)$ is a weak subtype of $type(C_1)$.

Lemma 2. If $type(C_1) \leq type(C_2)$, then $type(C_1)$ is a weak subtype of $type(C_2)$.
Proof. We proof by induction on the length of the derivation that if $\tau \leq \tau'$, then there is a subtype morphism from $struc(\tau')$ to $struc(\tau)$, where $struc(\mu t.\alpha)$ is given by $struc(\alpha[t \setminus \mu t.\alpha])$.

Basic step: suppose $B \leq B$, where B is a basic type. Then it follows immediately that there is a subtype morphism from $struc(B)$ to $struc(B)$.

Induction step for set types: suppose $\{\upsilon\} \leq \{\upsilon'\}$. Then it has been proven that $\upsilon \leq \upsilon'$. By induction, there is a subtype morphism from $struc(\upsilon')$ to $struc(\upsilon)$. This subtype morphism can easily be extended to a subtype morphism from $struc(\{\upsilon'\})$ to $struc(\{\upsilon\})$.

Induction step for record types: suppose $< l_i : \tau_i \mid i \in I > \leq < l_i : \tau'_i \mid i \in I' >$. Then it has been proven that $\tau_i \leq \tau'_i$ for $i \in I'$. By induction, there are subtype morphisms from $struc(\tau'_i)$ to $struc(\tau_i)$. These can be combined into a subtype morphism from $struc(< l_i : \tau'_i \mid i \in I' >)$ to $struc(< l_i : \tau_i \mid i \in I >)$.

Induction step for recursive types: suppose $\mu t.\alpha(t) \leq \mu t'.\alpha'(t')$. Then either rule $\mu t.\alpha = \alpha[t \setminus \mu t.\alpha]$ has been used and $\alpha[t \setminus \mu t.\alpha] \leq \alpha'[t' \setminus \mu t'.\alpha']$ has been proven or $t \leq t' \Rightarrow \alpha \leq \alpha'$ has been proven. In the first case, there is a subtype morphism from $struc(\mu t'.\alpha') = struc(\alpha'[t' \setminus \mu t'.\alpha'])$ to $struc(\mu t.\alpha) = struc(\alpha[t \setminus \mu t.\alpha])$ by induction. In the second case, for the sake of the argument, let b be a new basic type. Then we have: $b \leq b \Rightarrow \alpha(b) \leq \alpha'(b)$. Since b is a basic type, it follows that $b \leq b$ and, hence, $\alpha(b) \leq \alpha'(b)$. By induction, there is a subtype morphism from $struc(\alpha'(b))$ to $struc(\alpha(b))$. This subtype morphism can be extended to a subtype morphism from $struc(\mu t'.\alpha')$ to $struc(\mu t.\alpha)$ by replacing every $struc(b)$ in $struc(\alpha'(b))$ (resp., $struc(\alpha(b))$) by $struc(\alpha'(b))$ (resp., $struc(\alpha(b))$) and repeating this for the resulting tree, and so forth, and extending the subtype morphism accordingly. □

Lemma 3. If there exists a subtype morphism from $struc(C_1)$ to $struc(C_2)$, then there exists a subtype morphism from $struc(C_1)$ to $struc(C_2)$ that is faithful w.r.t. attributes.
Proof. We proof this by structural induction.

Basic step: Let φ be a subtype morphism from $struc(B)$ to $struc(T)$. Then it follows immediately that φ is faithful w.r.t. attributes.

Induction step for sets: Let φ be a subtype morphism from $struc(\{U\})$ to $struc(\{U'\})$. By induction, there is a subtype morphism ψ from $struc(U)$ to $struc(U')$ that is faithful w.r.t. attributes. Then ψ can be extended to a subtype from $struc(\{U\})$ to $struc(\{U'\})$ that is faithful w.r.t. attributes.

Induction step for records: Let φ be a subtype morphism from $struc(< l_i : U_i \mid i \in I >)$ to $struc(< l_i : U'_i \mid i \in I' >)$. By induction, there are subtype morphisms φ_i from $struc(U_i)$ to $struc(U'_i)$ that are faithful w.r.t. attributes. If $struc(C)$ is a subtree of both $struc(U_i)$ and $struc(U_j)$, for some class C, and both φ_i and φ_j map $struc(C)$ to the same $struc(C')$ differently, then choose one possibility to map $struc(C)$ to $struc(C')$ and adapt the subtype morphisms accordingly. The resulting subtype morphisms can be combined into a subtype morphism from $struc(< l_i : U_i \mid i \in I >)$ to $struc(< l_i : U'_i \mid i \in I' >)$ that is faithful w.r.t. attributes.

Induction step for classes: Let φ be a subtype morphism from $struc(C)$ to $struc(C')$, $atts(C)$ be $\{a_1 : T_1, \cdots, a_k : T_k\}$, and $atts(C')$ be $\{a'_1 : T'_1, \cdots, a'_{k'} : T'_{k'}\}$ Then there are subtype morphisms φ_i from $struc(T_i)$ to $struc(T'_{f(i)})$, where $f(i) = j$ if φ maps attribute a_i to attribute a'_j. Let $struc_i$ be obtained from $struc(T_i)$ by removing subtrees $struc(C)$ whose image under φ_i is $struc(C')$ and $struc'_i$ be obtained from $struc(T'_{f(i)})$ by removing subtrees $struc(C')$ which are the images of $struc(C)$ under φ_i. By induction, the projection of φ_i onto $struc_i$ and $struc'_i$, denoted by $\tilde{\varphi}_i : struc_i \rightarrow struc'_i$, is a subtype morphism that is faithful w.r.t. attributes. Let $struc$ be obtained from $struc(C)$ by replacing every $struc(T_i)$ by $struc_i$ and $struc'$ be obtained from $struc(C')$ by replacing every $struc(T'_{f(i)})$ by $struc'_{f(i)}$. As for set types, the subtype morphisms $\tilde{\varphi}_i$ can be combined into a subtype morphism ψ from $struc$ to $struc'$ that is faithful w.r.t. attributes. Finally, ψ can be extended to a subtype morphism from $struc(C)$ to $struc(C')$ that is faithful w.r.t. attributes by extending $struc$ to $struc(C)$, extending $struc'$ to $struc(C')$, and extending ψ accordingly. □

Example 7. Let C_{P1} be class 'Person1' and C_{P2} be class 'Person2' of Example 4. Let φ be the graph homomorphism from $struc(C_{P1})$ to $struc(C_{P2})$ that maps the node labeled 'Person1'

to the node labeled 'Person2', maps the arrow labeled 'addr' to the arrow labeled 'address', and preserves the labels of the other nodes and arrows. Then φ is injective, maps the root to the root, maps labels to themselves, except class names and attribute names, and maps classes to classes. Hence, $type(C_{P2})$ is a weak subtype of $type(C_{P1})$. □

3.2 Comparison of Methods

Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H, and φ be a subtype morphism from $struc(C_1)$ to $struc(C_2)$. Furthermore, let n be a node in $struc(C_1)$, such that $label(n)$ is the name of class C in H and $label(\varphi(n))$ is the name of class C' in H. Since φ is injective, φ^{-1} is a partial function from $struc(C_2)$ to $struc(C_1)$. For example, $\varphi^{-1}(\varphi(n)) = n$.

Let $m'(P') = E'$ be a method in $meths(C')$. If $d := s$ is an assignment in E', then destination $\varphi^{-1}(d)$ in C, corresponding to destination d in C', and source $\varphi^{-1}(s)$ in C, corresponding to source s in C', provided they exist, are obtained as follows: if $r_1 \cdots r_p$ is a path in $struc(C_2)$ starting at node $\varphi(n)$, then every occurrence of term $label(r_1).\cdots.label(r_p)$ in d and s is replaced by $label(\varphi^{-1}(r_1)).\cdots.label(\varphi^{-1}(r_p))$, provided all $\varphi^{-1}(r_i)$ exist.

The weak subfunction relation is defined in terms of generalisations of methods.

Definition 10. Let $\mathbf{spec}(m'(P') = E', \varphi)$ be the condition that, for every assignment $d := s$ in E', the existence of destination $\varphi^{-1}(d)$ in C implies the existence of source $\varphi^{-1}(s)$ in C, such that the type of $\varphi^{-1}(s)$ is a subtype of the type of $\varphi^{-1}(d)$. The generalisation of $m'(P') = E'$ in C, denoted by $\varphi^{-1}(m'(P') = E')$, is defined as:

$$m'(\tilde{P}') = \tilde{E}',$$

where

1. $\tilde{E}'$ is obtained from E' by replacing all assignments $d := s$, such that destination $\varphi^{-1}(d)$ exists, by $\varphi^{-1}(d) := \varphi^{-1}(s)$ and removing all other assignments
2. $\tilde{P}'$ is obtained from P' by removing parameters that do not occur in $\tilde{E}'$.

□

If $meth' \in meths(C')$, $meth \in meths(C)$, $\mathbf{spec}(meth', \varphi)$, and $func(C, \varphi^{-1}(meth')) =_{ext} func(C, meth)$, then and only then we say that $func(C', meth')$ is a weak subfunction of $func(C, meth)$ according to φ.

Lemma 4. If $func(C_1, meth_1) \leq func(C_2, meth_2)$ and $\mathbf{spec}(meth_1, \varphi)$, where φ is the subtype morphism from $struc(C_2)$ to $struc(C_1)$ as constructed in Lemma 2, then $func(C_1, meth_1)$ is a weak subfunction of $func(C_2, meth_2)$ according to φ.

Proof. Let f_1 be $func(C_1, meth_1)$ and f_2 be $func(C_2, meth_2)$. We proof that $\tilde{f}_1 = func(C_2, \varphi^{-1}(meth_1))$. Then it follows that $f_1 \leq f_2 \Leftrightarrow \tilde{f}_1 =_{ext} f_2 \Rightarrow func(C_2, \varphi^{-1}(meth_1)) =_{ext} f_2$.

Let E be the body of $meth_1$ and $\varphi^{-1}(E)$ be the body of $\varphi^{-1}(meth_1)$. That is, $\varphi^{-1}(E)$ contains exactly those assignments in E whose destinations are well-typed in C_2, because $\varphi(x) = x$ for all paths x in $struc(C_2)$. From $\mathbf{spec}(meth_1, \varphi)$it follows that the sources of the assignments in $\varphi^{-1}(E)$ are also well-typed in C_2. Let f_1 be $eval(E)(\sigma_0) = <\mathrm{id} = self.\mathrm{id}, a_1 : e_1, \cdots, a_k = e_k >$ and τ_2 be $type(C_2) = <\mathrm{id} : \mathrm{oid}, a_1 : v_1, \cdots, a_{k'} : v_{k'} >$. Then:

$$\begin{aligned}
&\tilde{f}_1 = \lambda self : \tau_2 \lambda \tilde{P}_1. <\mathrm{id} = self.\mathrm{id}, a_1 = proj(e_1, v_1), \cdots, a_{k'} = proj(e_{k'}, v_{k'}) > = \\
&\lambda self : \tau_2 \lambda \tilde{P}_1. proj(eval(E)(\sigma_0), \tau_2) = \\
&\lambda self : \tau_2 \lambda \tilde{P}_1. proj(eval(\varphi^{-1}(E))(\sigma_0), \tau_2) = \\
&\lambda self : \tau_2 \lambda \tilde{P}_1. eval(\varphi^{-1}(E))(proj(\sigma_0, \tau_2)) = \\
&func(C_2, \varphi^{-1}(meth_1)).
\end{aligned}$$

The first equality follows from the definition of $\tilde{f}$ and the second from de definition of $proj$. The third and fourth equality follow from the fact that $\varphi^{-1}(E)$ contains exactly those assignments in

E whose destinations are part of $type(C_2) = \tau_2$ and the fact that the sources of these assignments are also part of $type(C_2) = \tau_2$. Hence, for every assignment $d := s$ in E that does not belong to $\varphi^{-1}(E)$ it holds that:

$$proj(eval(L_1; d := s; L_2)(\sigma), \tau_2) = proj(eval(L_1; L_2)(\sigma), \tau_2).$$

The fifth equality follows from the definition of $func(C_2, \varphi^{-1}(meth_1))$. □

Example 8. Let C_{P1} be class 'Person1' and C_{P2} be class 'Person2' of Example 4. Let φ be the subtype morphism from $struc(flat(C_{P1}))$ to $struc(flat(C_{P2}))$ that maps attribute 'addr' to attribute 'address'. Then φ^{-1}(address.house) = addr.house and φ^{-1}(address.street) = addr.street exist, but φ^{-1}(address.city) does not exist. Let $meth_1$ be method 'change_addr' in class 'Person1' and $meth_2$ be method 'move' in class 'Person2'. The generalisation of method 'move' in class 'Person1' is given by:

$\varphi^{-1}(meth) =$
move (h:integer,s:string) = addr.house := h; addr.street := s.

Since $\mathbf{spec}(meth_2, \varphi)$ and $func(C_{P1}, \varphi^{-1}(meth_2)) =_{ext} func(C_{P1}, meth_1)$, $func(C_{P2}, meth_2)$ is a weak subfunction of $func(C_{P1}, meth_1)$ according to φ. □

Associate with node n in $struc(C_1)$ the set of functional forms of the methods in C:

$$funcs(n) = \{func(C, meth) \mid meth \in meths(C)\}.$$

Associate with node $\varphi(n)$ in $struc(C_2)$ the set of functional forms of the generalisations in C of the methods in C':

$$funcs(\varphi(n)) = \{func(C, \varphi^{-1}(meth)) \mid meth \in meths(C') \wedge \mathbf{spec}(meth, \varphi)\}.$$

The following property will be used to relate any method $meth_1$ in class C_1 to some method $meth_2$ in class C_2, such that $func(C_2, meth_2)$ is a subfunction of $func(C_1, meth_1)$, and, similarly, for the methods in the classes C_1 refers to.

Definition 11. Let H be a well-defined class hierarchy, C_1 and C_2 be classes in H, and φ be a subtype morphism from $struc(C_1)$ to $struc(C_2)$. Then φ is faithful with respect to methods if and only if it maps methods to more specialised methods, i.e., if and only if for every node n in $struc(C_1)$, such that $label(n) \in CN$, the following holds:

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

□

Finally, we define a synthetic subclass relation on flattened classes.

Definition 12. Let H be a well-defined class hierarchy. Let C_1 and C_2 be classes in H. Then C_1 is a subclass of C_2, denoted by $C_1 \preceq C_2$, if and only if there is a subtype morphism from $struc(C_2)$ to $struc(C_1)$ that is faithful w.r.t. methods. □

Lemma 5. The subclass relation is reflexive and transitive.
Proof. To proof reflexivity, let C be a class. The identity homomorphism from $struc(C)$ to $struc(C)$ is a subtype morphism and is faithful w.r.t. methods. 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 subtype morphisms φ from $struc(C_2)$ to $struc(C_1)$ and ψ from $struc(C_3)$ to $struc(C_2)$ that are faithful w.r.t. methods. But then $\varphi \circ \psi$ is a subtype morphism from $struc(C_3)$ to $struc(C_1)$ that is faithful w.r.t methods, because function composition preserves the defining properties of subtype morphisms and faithfulness w.r.t. methods. Hence, $C_1 \preceq C_3$. □

Lemma 6. The subclass relation is decidable.

Proof. Let C_1 and C_2 be classes. According to Lemma 3, we only have to consider subtype morphisms from $struc(C_2)$ to $struc(C_1)$ that are faithful w.r.t. attributes in order to decide whether $C_1 \preceq C_2$. Since every subtype morphism from $struc(C_2)$ to $struc(C_1)$ that is faithful w.r.t. attributes is a regular combination (determined by C_1 and C_2) of a finite number of finite subtype morphisms, there are only finitely many such subtype morphisms[3], which have to be checked for faithfulness w.r.t. methods on a finite domain only. Since $=_{ext}$ is decidable (Lemma 1), $\preceq$ is decidable. □

Of course, other subclass orders 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 (subtype morphism) and the way these objects are manipulated (faithful w.r.t. methods). 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 9. The following well-defined class hierarchy is a part of the definition of drawing tool A:

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

Class SFigure
Attributes
name:string
first:Square
second:Square
Methods
copy1 = first := second
copy2 = second := first
Endclass.

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

Class Rectangle
Attributes
x_left_up:integer
y_left_up:integer
width_x:integer
width_y:integer
Methods
set (x:integer, y:integer) =
x_left_up := x; y_left_up := y
translate (delta_x:integer, delta_y:integer) =
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

Class RFigure
Attributes
code:integer
left:Rectangle
right:Rectangle
Methods
r_copy = right := left
l_copy = left := right
Endclass.

[3]It is possible to adapt the algorithm from [12] to construct these subtype morphisms.

The designer has chosen to model 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. Let C_S be class 'Square' and C_R be class 'Rectangle'. According to the synthetic subclass order: $C_R \preceq C_S$. 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. □

4 Integration of Class Hierarchies

In this section, we introduce a join operator w.r.t. the subclass order to integrate classes. The join operator is defined in terms of attribute joins and method joins.

4.1 Integration of Attributes

Let $H = \{C_1, \cdots, C_n\}$ be a well-defined class hierarchy and, for every $i \in N = \{1, \cdots, n\}$, let c_i be $name(C_i)$, A_i be $atts(C_i)$, and M_i be $meths(C_i)$. Attribute joins are defined in terms of type joins according to a set of integration mappings.

Definition 13. Let $\Phi = \{\varphi_{i,j} : A_i \to A_j\}_{i,j \in N}$ be a set of partial, injective functions, one for every $i, j \in N$. Then Φ is an integration mapping for H if and only if

1. $\forall i \in N[\varphi_{i,i} = id_{A_i}]$
2. $\forall i, j, k \in N\ \forall x \in dom(\varphi_{i,j}) \forall y \in dom(\varphi_{j,k}) \forall z \in ran(\varphi_{j,k})$
   $[(\varphi_{i,j}(x) = y \wedge \varphi_{j,k}(y) = z) \Rightarrow \varphi_{i,k}(x) = z]$.

The set of all integration mappings for H is denoted by $IMaps_H$. □

An integration mapping for H can be regarded as an integration description for H, describing, for every $i, j \in N$, whether an attribute in class C_i corresponds to an attribute in class C_j.

Let $CN(H)$ be the set of class names in H, $\nu : \wp(CN(H)) \to CN$ be an injective name-giving function for joins of classes, such that $\nu(\{c\}) = c$ for every $c \in CN(H)$, $AN(H)$ be the set of attribute names in H, and $\alpha : \wp(CN(H) \times AN(H)) \to AN$ be an injective name-giving function for joins of attributes, such that $\alpha(\{(c, a)\}) = a$ for every $(c, a) \in CN(H) \times AN(H)$. The constraints on ν and α, together with the constraints on integration mappings, guarantee that attribute joins are well-defined.

Definition 14. Let $\{C_{i_1}, \cdots, C_{i_p}\}$ be a subset of H and Φ be an integration mapping for H. The join of $\{A_{i_1}, \cdots, A_{i_p}\}$ according to Φ is defined as:

$$\begin{aligned}\sqcup_\Phi(\{A_{i_1}, \cdots, A_{i_p}\}) = \{\alpha(\{(c_{i_1}, a_1), \cdots, (c_{i_p}, a_p)\}) : T_1 \sqcup \cdots \sqcup T_p \mid \\ \varphi_{(i_1,i_2)}(a_1 : T_1) = a_2 : T_2 \wedge \\ \vdots \\ \varphi_{(i_{p-1},i_p)}(a_{p-1} : T_{p-1}) = a_p : T_p \wedge \\ T_1 \sqcup \cdots \sqcup T_p \neq \bot\}\end{aligned}$$

where the type joins are given by (cf. [7]):

1. $T, T' \in CN$: $T \sqcup T' = \nu(\{c_i\}_{i \in I \cup I'})$ if $T = \nu(\{c_i\}_{i \in I})$ and $T' = \nu(\{c_i\}_{i \in I'})$
2. $T, T' \in \{\text{integer}, \text{rational}, \text{string}\}$: $T \sqcup T' = T$ if $T = T'$ and $T \sqcup T' = \bot$ otherwise
3. $T = \{U\}, T' = \{U'\}$: $T \sqcup T' = \{U \sqcup U'\}$ if $U \sqcup U' \neq \bot$ and $T \sqcup T' = \bot$ otherwise
4. $T = < l_i : U_i \mid i \in I >, T' = < l_i : U'_i \mid i \in I' >$:
   $T \sqcup T' = < l_i : U_i \sqcup U'_i \mid i \in I \cap I' \wedge U_i \sqcup U'_i \neq \bot >$ if $\exists i \in I \cap I'[U_i \sqcup U'_i \neq \bot]$ and
   $T \sqcup T' = \bot$ otherwise

5. otherwise: $T \sqcup T' = \bot$.

□

Lemma 7. The attribute join is computable.
Proof. The definition gives a procedure to construct attribute joins. □

Let Φ be an integration mapping for H and $C_{(i,j)}$ be $(\nu(\{c_i, c_j\}), \emptyset, \sqcup_\Phi(\{A_i, A_j\}), \emptyset)$, for $i, j \in N$. For example, $C_{(i,i)} = (c_i, \emptyset, A_i, \emptyset)$, for $i \in N$, because of the constraints on ν, α, and Φ. We define H_Φ as the set of all joins of pairs of classes in H with methods removed:

$$H_\Phi = \{C_{(i,j)}\}_{i,j \in N}.$$

Then $C_{(i,i)} \preceq C_{(i,j)}$ in H_Φ, for $i, j \in N$, because $\varphi_{i,j}$ and α induce a subtype morphism φ from $struc(C_{(i,j)})$ to $struc(C_{(i,i)}) = struc(C_i)$ as follows: if $a_1 : T_1 \in A_i$, $a_2 : T_2 \in A_j$, $\varphi_{i,j}(a_1 : T_1) = a_2 : T_2$, and $T_1 \sqcup T_2 \neq \bot$, then φ maps the arrow labeled $\alpha(\{(c_i, a_1), (c_j, a_2)\})$ to the arrow labeled a_1 and tree $struc(T_1 \sqcup T_2)$ to tree $struc(T_1)$.

4.2 Integration of Methods

Let C_i and C_j be classes in H, $A_{(i,j)}$ be $atts(C_{(i,j)}) = \sqcup_\Phi(\{A_i, A_j\})$, and $\psi_{i,j}$ be the subtype morphism from $struc(C_{(i,j)})$ to $struc(C_i)$ induced by $\varphi_{i,j}$ and α. The join of M_i and M_j is constructed in two steps. First, for both C_i and C_j, the set of generalisations of their methods in $C_{(i,j)}$ is constructed. Subsequently, the constructed sets are integrated by identifying methods using extensional equality of functional forms.

Let $m(P) = E$ be a method in M_i. Suppose $\mathbf{spec}(m(P) = E, \psi_{i,j})$, i.e., for every assignment $d := s$ in E, the existence of destination $\psi_{i,j}^{-1}(d)$ in $C_{(i,j)}$ implies the existence of source $\psi_{i,j}^{-1}(s)$ in $C_{(i,j)}$. Let $C'_{(i,j)}$ be $(\nu(\{c_i, c_j\}), \emptyset, A_{(i,j)}, \{\psi_{i,j}^{-1}(m(P) = E)\})$, for $i, j \in N$, where $\psi_{i,j}^{-1}(m(P) = E)$ is a generalisation as defined in Definition 10. For example, $C'_{(i,i)} = (c_i, A_i, \{m(P) = E\})$. Then $m(P) = E$ in $C'_{(i,i)}$ is a weak subfunction of $\psi_{i,j}^{-1}(m(P) = E)$ in $C'_{(i,j)}$ and, hence, $C'_{(i,i)} \preceq C'_{(i,j)}$ in $(H_\Phi \setminus \{C_{(i,i)}, C_{(i,j)}\}) \cup \{C'_{(i,i)}, C'_{(i,j)}\}$. The set of generalisations of M_i in $C_{(i,j)}$ is defined as:

$$M'_i = \{\varphi^{-1}(meth) \mid meth \in M_i \wedge \mathbf{spec}(meth, \varphi)\}.$$

The set of generalisations of M_j is defined similarly.

Let $MN(H)$ be the set of method names in H and $\mu : \wp(CN(H) \times MN(H)) \to CN(H) \times MN(H)$ be a choice function for joins of methods, such that $\mu(V) \in V$ for every $V \in dom(\mu)$.

Definition 15. The join of M_i and M_j according to Φ is defined as:

$$\begin{aligned} M_i \sqcup_\Phi M_j = \{m_k(P_k) = E_k \mid \exists m_i(P_i) = E_i \in M'_i\ \exists m_j(P_j) = E_j \in M'_j \\ [func(C'_{(i,j)}, m_i(P_i) = E_i) =_{ext} \\ func(C'_{(i,j)}, m_j(P_j) = E_j) \wedge \\ \mu(\{(c_i, m_i), (c_j, m_j)\}) = (c_k, m_k)]\}. \end{aligned}$$

If the functional forms of two or more methods in the join are extensionally equal, then choose one of them (using μ) and remove the other ones. □

Lemma 8. The method join is computable.
Proof. The definition gives a procedure to construct method joins, since sets of generalisations are computable (because **spec** is decidable and φ^{-1} is computable) and $=_{ext}$ is decidable (Lemma 1).
□

We define $C_i \sqcup_\Phi C_j$ as $(\nu(\{c_i, c_j\}), \emptyset, \sqcup_\Phi(\{A_i, A_j\}), M_i \sqcup_\Phi M_j)$ and $\tilde{H}_\Phi$ as the set of all joins of pairs of classes in H:

$$\tilde{H}_\Phi = \{C_i \sqcup_\Phi C_j\}_{i,j \in N}.$$

We say that $\tilde{H}_\Phi$ is the pre-join hierarchy according to Φ.

Definition 16. Let H_1 and H_2 be hierarchies in $prejoins(H) = \{\tilde{H}_\Phi \mid \Phi \in IMaps_H\}$. Then H_1 is a specialisation of H_2, denoted by $H_1 \sqsubseteq H_2$, if and only if

$$\forall i,j \in N\ \exists D_1 \in H_1\ \exists D_2 \in H_2 \\ [name(D_1) = name(D_2) = \nu(\{c_i, c_j\}) \wedge D_1 \preceq D_2].$$

The join hierarchies are the most specialised pre-join hierarchies:

$$joins(H) = \{H_1 \in prejoins(H) \mid \forall H_2 \in prejoins(H)[H_2 \sqsubseteq H_1 \Rightarrow H_1 \sqsubseteq H_2]\}.$$

If H' is a class hierarchy in $joins(H)$, then H' gives a possible combination of joins, i.e., one join for each pair of classes in H. The possible joins of class C_i and class C_j, for $i,j \in N$, are given by:

$$joins(C_i, C_j) = \{C \in H' \mid H' \in joins(H) \wedge name(C) = \nu(\{c_i, c_j\})\}.$$

□

The motivation for introducing join hierarchies besides join classes is the following: if join class C is chosen as the superclass of some pair of classes in H and join class C' is chosen as the superclass of another pair of classes in H, then there must be a join hierarchy that contains both C and C'.

We will proof that $joins(C_1, C_2)$ contains all possible joins of C_1 and C_2. For that purpose, let D be a class, such that $H \cup \{D\}$ is well-defined. Then C_1 and C_2 are compatible subclasses of D, denoted by $C_1, C_2 \preceq D$, if and only if there exist subtype morphisms φ_1 from $struc(D)$ to $struc(C_1)$ and φ_2 from $struc(D)$ to $struc(C_2)$ that are compatible, i.e., if subtrees $struc(D')$ and $struc(D'')$ are mapped to subtree $struc(C_1')$ by φ_1 and to subtree $struc(C_2')$ by φ_2, and $a' : T' \in atts(D')$ is mapped to $a_1 : T_1$ by φ_1 and to $a_2 : T_2$ by φ_2, and $a'' : T'' \in atts(D'')$ is mapped to $a_1 : T_1$ by φ_1, then $a'' : T'' \in atts(D'')$ must be mapped to $a_2 : T_2$ by φ_2.

Theorem 1. If $C_1, C_2 \preceq D$, then there exists an integration mapping $\Phi \in IMaps$, such that $\tilde{H}_\Phi \in joins(H)$ and $C_1 \sqcup_\Phi C_2 \preceq D$.

Proof. Suppose there exist compatible subtype morphisms φ_1 from $struc(D)$ to $struc(C_1)$ and φ_2 from $struc(D)$ to $struc(C_2)$. We proof that there exists an integration mapping $\Phi \in IMaps_H$, such that $C_1 \sqcup_\Phi C_2 \preceq D$. Then it follows that there is an integration mapping Φ', such that $\tilde{H}_{\Phi'} \in joins(H)$ and $C_1 \sqcup_{\Phi'} C_2 \preceq C_1 \sqcup_\Phi C_2 \preceq D$.

Define $\Phi = \{id_{A_i}\}_{i \in N} \cup \{\varphi_{i,j}\}_{i \neq j \in N}$ as follows. If $a : T \in atts(D')$ is mapped to $a_1 : T_1 \in atts(C_{i_1})$ by φ_1 and to $a_2 : T_2 \in atts(C_{i_2})$ by φ_2, then $\varphi_{i_1,i_2}(A_1 : T_1)$ is defined as $a_2 : T_2$. For other attributes, the $\varphi_{i,j}$ remain undefined.

Define tree homomorphism φ from $struc(D)$ to $struc(C_1 \sqcup_\Phi C_2)$ as follows. If attribute $a : T \in atts(D)$ is mapped to $a_1 : T_1$ by φ_1 and to $a_2 : T_2$ by φ_2, then φ maps the arrow labeled a to the arrow labeled $\alpha(\{(c_1, a_1), (c_2, a_2)\})$ and subtree $struc(T)$ to subtree $struc(T_1 \sqcup T_2)$. By structural induction it follows that φ is an injective tree homomorphism, preserves labels, except class names and attribute names, and is faithful w.r.t. classes and methods. Hence, $C_1 \sqcup_\Phi C_2 \preceq D$. □

Theorem 2. The set of join hierarchies is computable.

Proof. Since $IMaps_H$ is finite and, for given Φ, $\tilde{H}_\Phi$ is computable (Lemma 7 and 8), $prejoins(H)$ is computable. Furthermore, $\sqsubseteq$ is decidable, because $\preceq$ is decidable (Lemma 6). Hence, $joins(H)$ is computable. □

Example 10. Let H be the class hierarchy of Example 9. Let C_S be class 'Square' and C_R be class 'Rectangle'. If C is a class in $joins(C_S, C_R)$, then C belongs to $\tilde{H}_\Phi$ for some $\tilde{H}_\Phi \in joins(H)$ and, hence, C is given by an injective mapping from $atts(C_S)$ to $atts(C_R)$ that maps 'x_left_up' to 'x_left_up' or 'y_left_up', 'y_left_up' to 'y_left_up' or 'x_left_up', and 'width' to 'widt_x' or 'width_y' (other mappings lead to join classes with fewer methods). The designer will probably choose to map 'x_left_up' to 'x_left_up' and 'y_left_up' to 'y_left_up'. In that case, there are two possibilities to factorise 'Square' and 'Rectangle'. If 'width' is mapped to 'width_x', then the join of 'Square' and 'Rectangle' is 'Square' itself and 'Rectangle' can be factorised as follows:

Class Rectangle **Isa** Square
Attributes
width_y:integer
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'. If 'width' is mapped to 'width_y', then the join of 'Square' and 'Rectangle' is 'Square' again and 'Rectangle' can be factorised as follows:

Class Rectangle **Isa** Square
Attributes
width_x:integer
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 [16].

Let C_{SF} be class 'SFigure' and C_{RF} be class 'RFigure'. If C is a class in $joins(C_{SF}, C_{RF})$, then C is given by an injective mapping from $atts(C_{SF})$ to $atts(C_{RF})$ that maps 'first' to 'left' or 'right', and 'second' to 'left' or 'right' (other mappings lead to join classes with fewer methods). For both mappings, the join of 'SFigure' and 'Rfigure' is the same. So there is (up to attribute renaming) only one possibility to factorise 'SFigure' and 'RFigure':

Class SFigure **Isa** Figure
Attributes
name:string
Endclass
Class RFigure **Isa** Figure
Attributes
code:int
first:Rectangle
second:Rectangle
Endclass

Class Figure
Attributes
first:Square
second:Square
Methods
copy1 = first:=second
copy2 = second:=first
Endclass

where 'Figure' is the join of 'SFigure' and 'RFigure'. □

5 Conclusion

This report develops a formal approach to integrate class hierarchies, extending the work of [16, 17]. The approach is based on a synthetic subclass order, which is defined in terms of a weak subtype relation and a weak subfunction relation. Classes are integrated in a natural way using a join operator w.r.t. the subclass order, which is defined in terms of the join of the subtype relation and the join of the subfunction relation.

In contrast with [5, 6, 9], our subclass order compares classes not only by the structure of the objects in their extensions, but also by the way these objects are manipulated. Therefore,

our subclass order yields a more semantic approach towards schema integration in object-oriented databases.

Future research will include more sophisticated join operators to cope with extensions of the data model and heuristics to make the approach computationally feasible.

6 Bibliography

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

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

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

[4] H. Balsters, R. de By, and R. Zicari. Typed sets as a basis for object-oriented database schemas. In *Proc. Computing Science in the Netherlands*, pages 62–77. Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1991.

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

[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] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] C. Thieme and A. Siebes. Schema integration in object-oriented databases. Report CS-R9320, CWI, Amsterdam, The Netherlands, 1993.

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