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.

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 Class Hierarchies
   2.1 Attributes ................................................. 3
   2.2 Methods .................................................... 6

3 Comparison of Classes
   3.1 Comparison of Attributes ............................... 8
   3.2 Comparison of Methods .................................. 10

4 Integration of Class Hierarchies
   4.1 Integration of Attributes ............................... 14
   4.2 Integration of Methods ................................. 15

5 Conclusion .................................................. 17

6 Bibliography ................................................ 17
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 Berge 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], O2 [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 U {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 \( \text{name}(C) \). 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 \( \text{atts}(C) \), is defined as:

\[
\text{atts}(C) = A \cup \{ a : T : \exists C' \in H \ [\text{name}(C') \in S \land T \in \text{atts}(C')] \land \forall a' : T' \in A[a \neq a'] \}
\]

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 \( \text{type}(C) \), is defined as:

\[
\text{type}(C) = \tau(c, \emptyset)
\]

where

\[
\begin{align*}
\tau(d, \eta) &= \mu t_d . \prec \text{id : oid}, a_1 : \tau(T_1, \eta \cup \{d\}), \ldots, a_k : \tau(T_k, \eta \cup \{d\}) \succ \\
&\text{if } d \notin \eta \text{ and } \exists D \in H \ [\text{name}(D) = d \land \text{atts}(D) = \{a_1 : T_1, \ldots, a_k : T_k\}], \\
\tau(d, \eta) &= t_d \text{ if } d \in \eta, \\
\tau(B, \eta) &= B \text{ if } B \in \{\text{integer}, \text{rational}, \text{string}\}, \\
\tau(\{U\}, \eta) &= \{\tau(U, \eta)\}, \\
\tau(< l_1 : U_1, \ldots, l_n : U_n >) &= < l_1 : \tau(U_1, \eta), \ldots, l_n : \tau(U_n, \eta) >.
\end{align*}
\]

The set \( \eta \) 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 \( \eta \) 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:

\[
\begin{align*}
\tau_S &= \mu t_S . \prec \text{id : oid}, \text{house : integer}, \text{street : string}, \text{city : string} \succ \\
&= \prec \text{id : oid}, \text{house : integer}, \text{street : string}, \text{city : string} \succ, \\
\tau_A &= \mu t_A . \prec \text{id : oid}, \text{house : integer}, \text{street : string}, \text{city : string}, \text{country : string} \succ \\
&= \prec \text{id : oid}, \text{house : integer}, \text{street : string}, \text{city : string}, \text{country : string} \succ.
\end{align*}
\]

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 \( \alpha \) 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]: \( \tau \) is a subtype of \( \tau' \) if \( \tau \) has at least the fields of \( \tau' \) and the fields of \( \tau \) are at least as specialised as the fields of \( \tau' \).

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

\[
\begin{align*}
\tau &= < l_i : \tau_i | i \in I >, \\
\tau' &= < l'_i : \tau'_i | i \in I' >.
\end{align*}
\]
Then $\tau$ is a subtype of $\tau'$, denoted by $\tau \leq \tau'$, if

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

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]:

$$B \leq B' \text{ if } B \in \{\text{integer, rational, string}\},$$

$$\{v\} \leq \{v'\} \text{ if } v \leq v',$$

$$\mu t.\alpha \leq \mu t'.\alpha' \text{ if } t \leq t' \Rightarrow \alpha \leq \alpha'.$$

\(\square\)

For example, $\tau_A$ is a subtype of $\tau_S$.

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’.

```plaintext
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 $\alpha$ be $<\text{id:oid, name:string, mother:}\mu t_p.\alpha, \text{address:}\tau_S, \text{holiday\_address:}\tau_S>$, where $t_p$ is a type variable and $\tau_S$ is the underlying type of class ‘Simple\_Address’. The underlying types of class ‘Person’ and class ‘Employee’ are given by:

$$\tau_p = \mu t_p . \alpha$$
$$= <\text{id:oid, name:string, mother:}\mu t_p.\alpha, \text{address:}\tau_S, \text{holiday\_address:}\tau_S>,$$

$$\tau_E = \mu t_E . <\text{id:oid, name:string, mother:}\mu t_p.\alpha, \text{address:}\tau_S, \text{holiday\_address:}\tau_A,$$

$$\text{company:string, salary:integer}>$$

$$= <\text{id:oid, name:string, mother:}\mu t_p.\alpha, \text{address:}\tau_S, \text{holiday\_address:}\tau_A,$$

$$\text{company:string, salary:integer}>,$$

where $\tau_S$ and $\tau_A$ 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 $<\text{id:oid, name:string, mother:oid, address:oid}, \text{holiday\_address:oid}>$ and $<\text{id:oid, name:string, mother:oid, address:oid, holiday\_address:oid, company:string, salary:integer}>$, respectively. Note that $\tau_E$ is a subtype of $\tau_p$. \(\square\)

\(^1\)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 \( \text{mths}(C) \), is defined as:

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

Note that redefined methods override the corresponding methods in the superclass. \( \Box \)

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 \( \{a_1 : T_1, \ldots, a_k : T_k\} \) be \( \text{atts}(C) \) and \( m(P) = E \) be an element of \( \text{mths}(C) \). The functional form of method \( m(P) = E \) in class \( C \), denoted by \( \text{func}(C, m(P) = E) \), is defined as:

\[
\text{func}(C, m(P) = E) = \lambda \text{self} : \text{type}(C). \lambda P.
\]

\[
eval(E)(< \text{id} = \text{self}.\text{id}, a_1 = \sigma_0(a_1, T_1), \ldots, a_k = \sigma_0(a_k, T_k)>)
\]

where

\[
\sigma_0(r, d) = \text{self}.r \quad \text{if} \ d \in CN,
\]
\[
\sigma_0(r, B) = \text{self}.r \quad \text{if} \ B \in \{\text{integer, rational, string}\},
\]
\[
\sigma_0(r, \{U\}) = \text{self}.r,
\]
\[
\sigma_0(r, < l_1 : U_1, \ldots, l_n : U_n >) = < l_1 = \sigma_0(r, l_1, U_1), \ldots, l_n = \sigma_0(r, l_n, U_n) >,
\]
and \( \text{eval}(L_1; L_2)(\sigma) = \text{eval}(L_2)(\text{eval}(L_1)(\sigma)) \), and \( \text{eval}(d := s)(\sigma) \) is obtained from state \( \sigma \) by replacing the value of destination \( d \) by the value of source \( s \) in state \( \sigma \), and \( \text{eval}(\text{insert}(s, d))(\sigma) \) is obtained from state \( \sigma \) 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 \( \sigma \). □

**Example 5.** Let \( C_{P_1} \) be class ‘Person1’ and \( C_{P_2} \) be class ‘Person2’ of Example 4. Let \( \text{meth}_1 \) be method ‘change_addr’ in class ‘Person1’ and \( \text{meth}_2 \) 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:

\[
\text{func}(C_{P_1}, \text{meth}_1) = \lambda \text{self} : \tau_{P_1} \lambda h : \text{int} : \text{string}.
\begin{align*}
& <\text{id}=\text{self}.\text{id}, \text{name}=\text{self}.\text{name}, \text{addr}=\text{<house}=h, \text{street}=s> >, \\
\end{align*}
\]

\[
\text{func}(C_{P_2}, \text{meth}_2) = \lambda \text{self} : \tau_{P_2} \lambda c : \text{string} : \text{string}.
\begin{align*}
& <\text{id}=\text{self}.\text{id}, \text{name}=\text{self}.\text{name}, \text{address}=\text{<house}=h, \text{street}=s, \text{city}=c>, \\
& \text{company}=\text{self}.\text{company}, \text{salary}=\text{self}.\text{salary}>,
\end{align*}
\]

where \( \tau_{P_1} \) is the underlying type of class ‘Person1’ and \( \tau_{P_2} \) 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:

\[
f = \lambda \text{self} : \tau \lambda P. < \text{id} = \text{self}.\text{id}, a_1 = e_1, \cdots, a_k = e_k > \\
f' = \lambda \text{self} : \tau' \lambda P'. < \text{id} = \text{self}.\text{id}, a_1 = e'_1, \cdots, a_{k'} = e'_{k'} >,
\]

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 \( \tau' \), provided it exists, as:

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

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

\[
\text{proj}(\text{self}.a.l_1, \cdots, l_n, \sigma) = \text{self}.a.l_1, \cdots, l_n \\
\text{if } \tau'.a.l_i, \cdots, l_n \text{ exists and } \tau'.a.l_i, \cdots, l_n \leq \sigma, \\
\text{proj}(c, \sigma) = c \text{ if } c \in \text{Cons}, \\
\text{proj}(e \theta e'2, \sigma) = \text{proj}(e, \sigma) \theta \text{proj}(e', \sigma) \text{ if } \theta \in \{+, -, \times, \div\}, \\
\text{proj}(\sigma \cup \{g_1, \cdots, g_n, \{\sigma\}) = \text{proj}(\sigma) \cup \{\text{proj}(g_1, \sigma), \cdots, \text{proj}(g_n, \sigma)\}, \\
\text{proj}(< l_1 = g_1, \cdots, l_n = g_n >, < l_1 : \sigma_1, \cdots, l_{n'} : \sigma_{n'} >) = \\
< l_1 = \text{proj}(g_1, \sigma_1), \cdots, l_{n'} = \text{proj}(g_{n'}, \sigma_{n'}) >,
\]

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

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

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

**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 \( \text{proj} \) 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 \( \text{func}(C', m(P') = E') \) is a subfunction of \( \text{func}(C, m(P) = E) \). 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’.

```java
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 \( \text{name}(C) \) and \( \{a_1 : T_1, \ldots, a_k : T_k\} \) be \( \text{atts}(C) \). The tree representing the underlying type of class \( C \), denoted by \( \text{struc}(C) \), 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].
\[
\begin{aligned}
\text{where} \\
\text{struc}(d) = \text{struc}(D) \text{ if } \exists D \in H \land \text{name}(D) = d, \\
\text{struc}(B) \text{ has only one node, labeled } B, \text{ if } B \in \{\text{integer, rational, string}\}, \\
\text{struc}\{U\} \text{ consists of a root, labeled by } \{\}, \text{ a subtree struc}(U), \\
\text{and an unlabeled arrow from the root labeled by } \{\} \text{ to the root of struc}(U), \\
\text{struc}(< l_1 : U_1, \ldots, l_n : U_n >) \text{ consists of a root, labeled by } <>, \\
\text{subtrees struc}(U_1), \ldots, \text{ struc}(U_n), \text{ and arrows, labeled } l_i, \text{ one for each } \\
i \in \{1, \ldots, n\}, \text{ from the root labeled by } <> \text{ to the root of struc}(U_i). \\
\end{aligned}
\]

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 \( \varphi \) be a graph homomorphism from \( \text{struc}(C_1) \) to \( \text{struc}(C_2) \). If \( \varphi \) maps the root of \( \text{struc}(C_1) \) to the root of \( \text{struc}(C_2) \), then and only then we say that \( \varphi \) is a tree homomorphism. If \( \varphi \) maps labels to themselves, i.e., if for every node or arrow \( q \) in \( \text{struc}(C_1) \), the following holds:

\[
\text{label}(q) = l \Rightarrow \text{label}(\varphi(q)) = l,
\]

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

\[
\text{label}(n) \in CN \Rightarrow \text{label}(\varphi(n)) \in CN,
\]

then and only then we say that \( \varphi \) is faithful with respect to classes. Finally, if \( \varphi \) maps attributes in one class to attributes in another class consistently, i.e., if for every node \( n_1 \) in \( \text{struc}(C_1) \) with outgoing arrow \( r_1 \) and for every node \( n_2 \) in \( \text{struc}(C_1) \) with outgoing arrow \( r_2 \), the following holds:

\[
(\text{label}(n_1) = \text{label}(n_2) \in CN \land \text{label}(r_1) = \text{label}(r_2) \land \text{label}(\varphi(n_1)) = \text{label}(\varphi(n_2))) \Rightarrow \\
\text{label}(\varphi(r_1)) = \text{label}(\varphi(r_2)),
\]

then and only then we say that \( \varphi \) 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 \( \text{type}(T_2) \) is a weak subtype of \( \text{type}(T_1) \).

**Definition 9.** Let \( H \) be a well-defined class hierarchy, \( C_1 \) and \( C_2 \) be classes in \( H \), and \( \varphi \) be a graph homomorphism from \( \text{struc}(C_1) \) to \( \text{struc}(C_2) \). Then \( \varphi \) 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.

\( \square \)

If there exists a subtype morphism from \( \text{struc}(C_1) \) to \( \text{struc}(C_2) \), then and only then we say that \( \text{type}(C_2) \) is a weak subtype of \( \text{type}(C_1) \).
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 $\text{struc}(\tau)$ to $\text{struc}(\tau)$, where $\text{struc}(\tau)$ is given by $\text{struc}(\alpha[t \setminus \mu.\alpha])$.

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

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

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

Induction step for recursive types: suppose $\mu.\alpha(t) \leq \mu.\alpha'(t')$. Then either rule $\mu.\alpha = \alpha[t \setminus \mu.\alpha]$ has been used and $\alpha[t \setminus \mu.\alpha] \leq \alpha'[t' \setminus \mu.\alpha']$ has been proven or $t \leq t' \Rightarrow \alpha \leq \alpha'$ has been proven. In the first case, there is a subtype morphism from $\text{struc}(\mu.\alpha) = \text{struc}(\alpha[t \setminus \mu.\alpha])$ to $\text{struc}(\mu.\alpha) = \text{struc}(\alpha[t' \setminus \mu.\alpha'])$ 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 $\text{struc}(\alpha'(b))$ to $\text{struc}(\alpha(b))$. This subtype morphism can be extended to a subtype morphism from $\text{struc}(\mu.\alpha')$ to $\text{struc}(\mu.\alpha)$ by replacing every $\text{struc}(b)$ in $\text{struc}(\alpha'(b))$ (resp., $\text{struc}(\alpha(b))$) by $\text{struc}(\alpha'(b))$ (resp., $\text{struc}(\alpha(b))$) 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 $\text{struc}(C_1)$ to $\text{struc}(C_2)$, then there exists a subtype morphism from $\text{struc}(C_1)$ to $\text{struc}(C_2)$ that is faithful w.r.t. attributes.

Proof. We proof this by structural induction.

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

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

Induction step for records: Let $\varphi$ be a subtype morphism from $\text{struc}(\langle l_i : U_i \mid i \in I \rangle)$ to $\text{struc}(\langle l_i' : U_i' \mid i \in I' \rangle)$. By induction, there are subtype morphisms $\varphi_i$ from $\text{struc}(U_i)$ to $\text{struc}(U_i')$ that are faithful w.r.t. attributes. If $\text{struc}(C)$ is a subtree of both $\text{struc}(U_i)$ and $\text{struc}(U_i')$, for some class $C$, and both $\varphi_i$ and $\varphi_j$ map $\text{struc}(C)$ to the same $\text{struc}(C')$ differently, then choose one possibility to map $\text{struc}(C)$ to $\text{struc}(C')$ and adapt the subtype morphisms accordingly. The resulting subtype morphisms can be combined into a subtype morphism from $\text{struc}(\langle l_i : U_i \mid i \in I \rangle)$ to $\text{struc}(\langle l_i' : U_i' \mid i \in I' \rangle)$ that is faithful w.r.t. attributes.

Induction step for classes: Let $\varphi$ be a subtype morphism from $\text{struc}(C)$ to $\text{struc}(C')$, $\text{atts}(C')$ be $\{a_1 : \tau_1, \cdots, a_k : \tau_k\}$, and $\text{atts}(C')$ be $\{a'_1 : \tau_1', \cdots, a'_{k'} : \tau_{k'}\}$. Then there are subtype morphisms $\varphi_i$ from $\text{struc}(T_i)$ to $\text{struc}(T_i')$, where $f(i) = j$ if $\varphi$ maps attribute $a_i$ to attribute $a'_j$. Let $\psi_i$ be obtained from $\text{struc}(T_i)$ by removing subtrees $\text{struc}(C)$ whose image under $\varphi_i$ is $\text{struc}(C')$ and $\text{struc}_i'$ be obtained from $\text{struc}(T_i')$ by removing subtrees $\text{struc}(C')$ which are the images of $\text{struc}(C)$ under $\varphi_i$. By induction, the projection of $\varphi_i$ onto $\text{struc}_i$ and $\text{struc}_i'$, denoted by $\tilde{\varphi}_i : \text{struc}_i \to \text{struc}_i'$, is a subtype morphism that is faithful w.r.t. attributes. Let $\text{struc}$ be obtained from $\text{struc}(C)$ by replacing every subtree $\text{struc}(C)$ by $\text{struc}_i$ and $\text{struc'}$ be obtained from $\text{struc}(C')$ by replacing every subtree $\text{struc}(T_i')$ by $\text{struc}_i'$. As for set types, the subtype morphisms $\tilde{\varphi}_i$ can be combined into a subtype morphism $\psi$ from $\text{struc}$ to $\text{struc}'$ that is faithful w.r.t. attributes. Finally, $\psi$ can be extended to a subtype morphism from $\text{struc}(C)$ to $\text{struc}(C')$ that is faithful w.r.t. attributes by extending $\text{struc}$ to $\text{struc}(C)$, extending $\text{struc}'$ to $\text{struc}(C')$, and extending $\psi$ accordingly. □

Example 7. Let $C_{P1}$ be class ‘Person1’ and $C_{P2}$ be class ‘Person2’ of Example 4. Let $\varphi$ be the graph homomorphism from $\text{struc}(C_{P1})$ to $\text{struc}(C_{P2})$ 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 \( \varphi \) is injective, maps the root to the root, maps labels to themselves, except class names and attribute names, and maps classes to classes. Hence, \( \text{type}(C_{P2}) \) is a weak subtype of \( \text{type}(C_{P1}) \). □

### 3.2 Comparison of Methods

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

Let \( m'(P') = E' \) be a method in \( \text{meths}(C') \). 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 \( \text{struct}(C_2) \) starting at node \( \varphi(n) \), then every occurrence of term \( \text{label}(r_1) \cdots \text{label}(r_p) \) in \( d \) and \( s \) is replaced by \( \text{label}(\varphi^{-1}(r_1)) \cdots \text{label}(\varphi^{-1}(r_p)) \), provided all \( \varphi^{-1}(r_i) \) exist.

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

**Definition 10.** Let \( \text{spec}(m'(P') = E', \varphi) \) 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 \( \text{meth}' \in \text{meths}(C') \), \( \text{meth} \in \text{meths}(C) \), \( \text{spec}(\text{meth}', \varphi) \), and \( \text{func}(C, \varphi^{-1}(\text{meth}')) = \text{ext} \text{func}(C, \text{meth}) \), then and only then we say that \( \text{func}(C', \text{meth}') \) is a weak subfunction of \( \text{func}(C, \text{meth}) \) according to \( \varphi \).

**Lemma 4.** If \( \text{func}(C_1, \text{meth}_1) \leq \text{func}(C_2, \text{meth}_2) \) and \( \text{spec}(\text{meth}_1, \varphi) \), where \( \varphi \) is the subtype morphism from \( \text{struct}(C_2) \) to \( \text{struct}(C_1) \) as constructed in Lemma 2, then \( \text{func}(C_1, \text{meth}_1) \) is a weak subfunction of \( \text{func}(C_2, \text{meth}_2) \) according to \( \varphi \).

**Proof.** Let \( f_1 \) be \( \text{func}(C_1, \text{meth}_1) \) and \( f_2 \) be \( \text{func}(C_2, \text{meth}_2) \). We proof that \( \tilde{f}_1 = \text{func}(C_2, \varphi^{-1}(\text{meth}_1)) \). Then it follows that \( f_1 \leq f_2 \Leftrightarrow \tilde{f}_1 = \text{ext} f_2 \Rightarrow \text{func}(C_2, \varphi^{-1}(\text{meth}_1)) = \text{ext} f_2 \).

Let \( E \) be the body of \( \text{meth}_1 \) and \( \varphi^{-1}(E) \) be the body of \( \varphi^{-1}(\text{meth}_1) \). 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 \( \text{struct}(C_2) \). From \( \text{spec}(\text{meth}_1, \varphi) \) it follows that the sources of the assignments in \( \varphi^{-1}(E) \) are also well-typed in \( C_2 \). Let \( f_1 \) be \( \text{eval}(E)(\sigma_0) = \langle \text{id} = \text{self}.\text{id}, a_1 : e_1, \ldots, a_k : e_k > \rangle \) and \( \tau_2 \) be \( \text{type}(C_2) = \langle \text{id} : \text{oid}, a_1 : v_1, \ldots, a_k : v_k > \rangle \). Then:

\[
\tilde{f}_1 = \lambda \text{self}. \tau_2 \lambda \tilde{P}_1. \langle \text{id} = \text{self}.\text{id}, a_1 = \text{proj}(e_1, v_1), \ldots, a_k = \text{proj}(e_k, v_k) > = \\
\lambda \text{self}. \tau_2 \lambda \tilde{P}_1. \text{proj}(\text{eval}(E)(\sigma_0), \tau_2) = \\
\lambda \text{self}. \tau_2 \lambda \tilde{P}_1. \text{proj}(\text{eval}(\varphi^{-1}(E))(\sigma_0), \tau_2) = \\
\lambda \text{self}. \tau_2 \lambda \tilde{P}_1. \text{eval}(\varphi^{-1}(E))(\sigma_0, \tau_2) = \\
\text{func}(C_2, \varphi^{-1}(\text{meth}_1)).
\]

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

11
$E$ whose destinations are part of $\text{type}(C_2) = \tau_2$ and the fact that the sources of these assignments are also part of $\text{type}(C_2) = \tau_2$. Hence, for every assignment $d := s$ in $E$ that does not belong to $\varphi^{-1}(E)$ it holds that:

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

The fifth equality follows from the definition of $\text{func}(C_2, \varphi^{-1}(\text{meth}_1))$. □

**Example 8.** Let $C_{P_1}$ be class 'Person1' and $C_{P_2}$ be class 'Person2' of Example 4. Let $\varphi$ be the subtype morphism from $\text{struct}(\text{flat}(C_{P_1}))$ to $\text{struct}(\text{flat}(C_{P_2}))$ that maps attribute 'addr' to attribute 'address'. Then $\varphi^{-1}(\text{address.house}) = \text{addr.house}$ and $\varphi^{-1}(\text{address.street}) = \text{addr.street}$ exist, but $\varphi^{-1}(\text{address.city})$ does not exist. Let $\text{meth}_1$ be method 'change_addr' in class 'Person1' and $\text{meth}_2$ be method 'move' in class 'Person2'. The generalisation of method 'move' in class 'Person1' is given by:

$$\varphi^{-1}(\text{meth}) =$$

move (h:integer,s:string) = addr.house := h; addr.street := s.

Since $\text{spec}(\text{meth}_2, \varphi)$ and $\text{func}(C_{P_1}, \varphi^{-1}(\text{meth}_2)) = \text{ext} \text{func}(C_{P_1}, \text{meth}_1)$, $\text{func}(C_{P_2}, \text{meth}_2)$ is a weak subfunction of $\text{func}(C_{P_1}, \text{meth}_1)$ according to $\varphi$. □

Associate with node $n$ in $\text{struct}(C_1)$ the set of functional forms of the methods in $C$:

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

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

$$\text{funcs}(\varphi(n)) = \{\text{func}(C, \varphi^{-1}(\text{meth})) | \text{meth} \in \text{meths}(C') \land \text{spec}(\text{meth}, \varphi)\}.$$ 

The following property will be used to relate any method $\text{meth}_1$ in class $C_1$ to some method $\text{meth}_2$ in class $C_2$, such that $\text{func}(C_2, \text{meth}_2)$ is a subfunction of $\text{func}(C_1, \text{meth}_1)$, 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 $\varphi$ be a subtype morphism from $\text{struct}(C_1)$ to $\text{struct}(C_2)$. Then $\varphi$ 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 $\text{struct}(C_1)$, such that $\text{label}(n) \in CN$, the following holds:

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

□

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 $\text{struct}(C_2)$ to $\text{struct}(C_1)$ that is faithful w.r.t. methods. □

**Lemma 5.** The subclass relation is reflexive and transitive.

*Proof.* To prove reflexivity, let $C$ be a class. The identity homomorphism from $\text{struct}(C)$ to $\text{struct}(C)$ 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 $\varphi$ from $\text{struct}(C_2)$ to $\text{struct}(C_1)$ and $\psi$ from $\text{struct}(C_3)$ to $\text{struct}(C_2)$ that are faithful w.r.t. methods. But then $\varphi \circ \psi$ is a subtype morphism from $\text{struct}(C_3)$ to $\text{struct}(C_1)$ 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 \( \text{struc}(C_2) \) to \( \text{struc}(C_1) \) that are faithful w.r.t. attributes in order to decide whether \( C_1 \preceq C_2 \). Since every subtype morphism from \( \text{struc}(C_2) \) to \( \text{struc}(C_1) \) 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\(^5\), which have to be checked for faithfulness w.r.t. methods on a finite domain only. Since \( \cong_{w_{\text{ext}}} \) is decidable (Lemma 1), \( \preceq \) is decidable. \( \Box \)

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: 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
```

\(^5\)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: \(C_R \sqsubseteq C_S\). 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 \(H = \{C_1, \ldots, C_n\}\) be a well-defined class hierarchy and, for every \(i \in N = \{1, \ldots, n\}\), let \(c_i\) be \(\text{name}(C_i)\), \(A_i\) be \(\text{atts}(C_i)\), and \(M_i\) be \(\text{meths}(C_i)\). Attribute joins are defined in terms of type joins according to a set of integration mappings.

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

1. \(\forall i \in N[\varphi_{i,i} = \text{id}_{A_i}]\)
2. \(\forall i, j, k \in N \forall x \in \text{dom}(\varphi_{i,j}) \forall y \in \text{dom}(\varphi_{j,k}) \forall z \in \text{ran}(\varphi_{j,k})\)
   \(\left[\varphi_{i,j}(x) = y \land \varphi_{j,k}(y) = z \Rightarrow \varphi_{i,k}(x) = z\right].\)

The set of all integration mappings for \(H\) is denoted by \(\text{IMaps}_H\). □

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 \(H\), \(\nu : \varphi(CN(H)) \rightarrow CN\) 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 \(\alpha : \varphi(CN(H) \times AN(H)) \rightarrow AN\) 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 \(\nu\) and \(\alpha\), together with the constraints on integration mappings, guarantee that attribute joins are well-defined.

**Definition 14.** Let \(\{C_{i_1}, \ldots, C_{i_p}\}\) be a subset of \(H\) and \(\Phi\) be an integration mapping for \(H\). The join of \(\{A_{i_1}, \ldots, A_{i_p}\}\) according to \(\Phi\) is defined as:

\[
\bigcup_\Phi\{A_{i_1}, \ldots, A_{i_p}\} = \{\alpha((c_{i_1}, a_{i_1}), \ldots, (c_{i_p}, a_{i_p})) : T_1 \sqcup \cdots \sqcup T_p\ |
\varphi_{i_1,i_2}(a_{i_1} : T_1) = a_2 : T_2 \land
\vdots
\varphi_{i_{p-1},i_p}(a_{p-1} : T_{p-1}) = a_p : T_p \land
T_1 \sqcup \cdots \sqcup T_p \neq \bot\}
\]

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. \(T, T' \in \{\text{integer, rational, string}\} : T \sqcup T' = T\) 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. \(T = < l_i : U_i | i \in I >, T' = < l'_i : U'_i | i \in I' > : T \sqcup T' = < l_i : U_i \sqcup U'_i | i \in I \cap I' \cap U_i \sqcup U'_i \neq \bot >\) if \(\exists i \in I \cap I'(U_i \sqcup U'_i \neq \bot)\) and \(T \sqcup T' = \bot\) otherwise

14
5. otherwise: $T \cup T' = \bot$.

□

**Lemma 7.** The attribute join is computable.

**Proof.** The definition gives a procedure to construct attribute joins. □

Let $\Phi$ be an integration mapping for $H$ and $C_{(i,j)}$ be $(\nu(\{c_{i}, c_{j}\}), 0, \cup_{\Phi}(\{A_{i}, A_{j}\}), 0)$, for $i, j \in N$. For example, $C_{(i,i)} = \{0, A_{i}, 0\}$, for $i \in N$, because of the constraints on $\nu$, $\alpha$, and $\Phi$. We define $H_{\Phi}$ 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 $C_{(i,j)} \preceq C_{(i,j)}$ in $H_{\Phi}$, for $i, j \in N$, because $\varphi_{i,j}$ and $\alpha$ induce a subtype morphism $\varphi$ from $\text{struct}(C_{(i,j)})$ to $\text{struct}(C_{(i,j)}) = \text{struct}(C_{(i,j)})$ 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 $T_{1} \cup T_{2} \neq \bot$, then $\varphi$ maps the arrow labeled $\alpha(\{(c_{i}, a_{1}), (c_{j}, a_{2})\})$ to the arrow labeled $a_{1}$ and tree $\text{struct}(T_{1} \cup T_{2})$ to tree $\text{struct}(T_{1})$.

### 4.2 Integration of Methods

Let $C_{i}$ and $C_{j}$ be classes in $H$, $A_{(i,j)}$ be $\text{atts}(C_{(i,j)}) = \cup_{\Phi}(\{A_{i}, A_{j}\})$, and $\psi_{i,j}$ be the subtype morphism from $\text{struct}(C_{(i,j)})$ to $\text{struct}(C_{i})$ induced by $\varphi_{i,j}$ and $\alpha$. 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 $\text{spec}(m(P) = E, \psi_{i,j})$, 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 $(\nu(\{c_{i}, c_{j}\}), 0, A_{(i,j)}, \{\psi_{i,j}^{-1}(m(P) = E)\})$, 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, $C'_{(i,j)} \preceq C'_{(i,j)}$ 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:

$$M'_{i} = \{\varphi^{-1}(\text{meth}) \mid \text{meth} \in M_{i} \land \text{spec}(\text{meth}, \varphi)\}.$$

The set of generalisations of $M_{j}$ is defined similarly.

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

**Definition 15.** The join of $M_{i}$ and $M_{j}$ according to $\Phi$ is defined as:

$$M_{i} \cup_{\Phi} M_{j} = \{m_{k}(P_{k}) = E_{k} \mid \exists m_{i}(P_{i}) = E_{i} \in M'_{i} \exists m_{j}(P_{j}) = E_{j} \in M'_{j}
\text{func}(C_{(i,j)}), m_{i}(P_{i}) = E_{i} = \text{ext}
\text{func}(C'_{(i,j)}), m_{j}(P_{j}) = E_{j} \land
\mu(\{(c_{i}, m_{i}), (c_{j}, m_{j})\}) = (c_{k}, m_{k})\}.$$

If the functional forms of two or more methods in the join are extensionally equal, then choose one of them (using $\mu$) 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 $\text{spec}$ is decidable and $\varphi^{-1}$ is computable) and $\text{ext}$ is decidable (Lemma 1). □

We define $C_{i} \cup_{\Phi} C_{j}$ as $(\nu(\{c_{i}, c_{j}\}), 0, \cup_{\Phi}(\{A_{i}, A_{j}\}), M_{i} \cup_{\Phi} M_{j})$ and $\tilde{H}_{\Phi}$ as the set of all joins of pairs of classes in $H$:

$$\tilde{H}_{\Phi} = \{C_{i} \cup_{\Phi} C_{j}\}_{i,j \in N}.$$
We say that $\bar{H}_\Phi$ is the pre-join hierarchy according to $\Phi$.

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

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

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

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

If $H'$ is a class hierarchy in $\text{joins}(H)$, 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:

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

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 prove that $\text{joins}(C_1, C_2)$ 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 $\varphi_1$ from $\text{struct}(D)$ to $\text{struct}(C_1)$ and $\varphi_2$ from $\text{struct}(D)$ to $\text{struct}(C_2)$ that are compatible, i.e., if subtrees $\text{struct}(D')$ and $\text{struct}(D'')$ are mapped to subtree $\text{struct}(C'_1)$ by $\varphi_1$ and to subtree $\text{struct}(C'_2)$ by $\varphi_2$, and $a' : T' \in \text{atts}(D')$ is mapped to $a_1 : T_1$ by $\varphi_1$ and to $a_2 : T_2$ by $\varphi_2$, then $a' : T' \in \text{atts}(D'')$ must be mapped to $a_2 : T_2$ by $\varphi_2$.

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

**Proof.** Suppose there exist compatible subtype morphisms $\varphi_1$ from $\text{struct}(D)$ to $\text{struct}(C_1)$ and $\varphi_2$ from $\text{struct}(D)$ to $\text{struct}(C_2)$. We proof that there exists an integration mapping $\Phi \in IMaps_H$, such that $C_1 \sqsubseteq \Phi C_2 \preceq D$. Then it follows that there is an integration mapping $\Phi'$, such that $H_\Phi \in \text{joins}(H)$ and $C_1 \sqsubseteq \Phi' C_2 \preceq D$. Define $\Phi = \{ \{\} \} \in N \cup \{ \{\} \}_{i \neq j}$ as follows. If $a : T \in \text{atts}(D')$ is mapped to $a_1 : T_1 \in \text{atts}(C_1)$ by $\varphi_1$ and to $a_2 : T_2 \in \text{atts}(C_2)$ by $\varphi_2$, 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 $\varphi : \text{struct}(D)$ to $\text{struct}(C_1 \sqcup \Phi C_2)$ as follows. If attribute $a : T \in \text{atts}(D)$ is mapped to $a_1 : T_1$ by $\varphi_1$ and to $a_2 : T_2$ by $\varphi_2$, then $\varphi$ maps the arrow labeled $a$ to the arrow labeled $\{\{a_1, a_2\}\}$ and subtree $\text{struct}(T)$ to subtree $\text{struct}(T_1 \sqcup T_2)$. By structural induction it follows that $\varphi$ is an injective tree homomorphism, preserves labels, except class names and attribute names, and is faithful w.r.t. classes and methods. Hence, $C_1 \sqsubseteq \Phi C_2 \preceq D$. □

**Theorem 2.** The set of join hierarchies is computable.

**Proof.** Since $IMaps_H$ is finite and, for given $\Phi, H_\Phi$ is computable (Lemma 7 and 8), $\text{prejoins}(H)$ is computable. Furthermore, $\sqsubseteq$ is decidable, because $\preceq$ is decidable (Lemma 6). Hence, $\text{joins}(H)$ 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 $\text{joins}(C_S, C_R)$, then $C$ belongs to $H_\Phi$ for some $H_\Phi \in \text{joins}(H)$ and, hence, $C$ is given by an injective mapping from $\text{atts}(C_S)$ to $\text{atts}(C_R)$ that maps ‘x’ to ‘x’, ‘y’ to ‘y’, ‘x’ to ‘x’, ‘y’ to ‘y’, and ‘width’ to ‘width.x’ or ‘width.y’ (other mappings lead to join classes with fewer methods). The designer will probably choose to map ‘x’ to ‘x’ and ‘y’ to ‘y’, and ‘width’ to ‘width.x’. 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.x:integer
Methods
rotate = y.left.up := y.left.up + width;
width := width.x - width;
width.y := width.y - width;
width := width + width.x
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 $\text{joins}(C_{SF}, C_{RF})$, then $C$ is given by an injective mapping from $\text{atts}(C_{SF})$ to $\text{atts}(C_{RF})$ 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 ‘SFfigure’ 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

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


