An approach to schema integration based on transformations and behaviour

C Thieme, A Siebes

Computer Science/Department of Algorithmics and Architecture

CS-R9403 1994
An Approach to Schema Integration Based on Transformations and Behaviour

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

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

Abstract

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

Keywords and Phrases: Database design, Object-oriented databases, Schema integration.
Note: This research is partly funded by the Dutch Organisation for Scientific Research through NFI-grant NF74.

Contents

1 Introduction 2
2 Database schemas 2
   2.1 Underlying types .................................................. 4
   2.2 Underlying constraints ............................................. 9
   2.3 Functional forms .................................................... 10
3 Schema transformations 13
4 Application of schema transformations 18
5 Conclusion 25
A Meet types 26
B Referential integrity formulas 26
C Types of sources and destinations 27
D Syntactic evaluation of method bodies 27
E Bibliography 29
1 Introduction

Schema integration is an important and non-trivial task in database design. It occurs when a number of different user views, developed for a new database system, or a number of existing database schemas have to be integrated into a global, unified schema. As schema integration is a difficult task, methods to support the designer with this task are essential. In [7], a framework for comparing integration methods is given. The framework identifies four steps. In the first step, the preintegration step, an integration strategy is chosen and additional information on the schemas is gathered. Subsequently, the schemas are analysed and compared to find similarities/conflicts among the schemas. In the conforming step, the conflicts found in the comparison step have to be resolved. Finally, in the last step, the schemas are merged by superimposition and the resulting schema is analysed and restructured if necessary.

For our purpose, the main characteristic of an integration method is: which similarities/conflicts are detected and how are conflicts resolved? A number of integration methods use assertions among different component schemas to compare attributes and entity types. In [20], interschema assertions, names, and types are used to compare object types. In [18], schemas are merged using schema operators and assertions among entity types and attributes in different schemas. And in [16], attribute assertions (e.g., key/non-key and lower/upper bounds) are used to compare attributes and entity types. However, the assertions must be supplied by the designer and the resolution of conflicts strongly depends on the common sense of the designer.

Other methods use schema transformations to resolve structural conflicts. In [11], structural transformations are defined to integrate compatible structures. In [19], a number of schema transformations (e.g., join and meet) are proposed to restructure schemas. And in [6], transformations between attributes, entities and relationships are used to resolve type conflicts. However, only the last one gives a heuristic (viz., concept likeness/unlikeness) for applying the transformations.

A number of recent methods use more specific information on semantical properties of attributes and entity types to detect similarities and conflicts. In [21, 26], attribute assertions are used to define relationships between an attribute on one hand and a semantic point or a set of concepts on the other hand. Again, the assertions must be supplied by the designer. In [22], a database metadictionary is used to define a semantic domain for each attribute. And in [12], a terminological knowledge base containing information on negative and positive associations between terms and information on specialisation of terms is used to compare entity types.

This report presents a new approach to schema integration, based on schema transformations and the approach taken in [23, 24], where classes are compared by structure and by behaviour. The approach consist of two steps. First, component schemas are restructured using schema transformations, and syntactical properties of methods are used to guide the restructuring process. Subsequently, the component schemas are merged using join operators, and semantical properties of methods are used to guide the merging process. There is, as far as the authors know, no other approach that uses methods to compare attributes. For sake of completeness, it should be mentioned that there is an approach to schema evolution that analyses methods ([9]), not to compare attributes, but to solve non-legitimate overriding of methods.

The outline of this report is as follows. In the next section, database schemas are introduced and formalised in terms of (recursive) types, predicates, and functions. In Section 3, a number of well-known type transformations are extended to recursive types and it is shown how these type transformations induce schema transformations. In Section 4, it is shown how methods can be used to guide schema restructuring and a heuristic algorithm is given to restructure and merge schema. In the last section, a summary and directions for further research are given.

2 Database schemas

In this section, we introduce a subset of the database schemas found in object-oriented database languages such as Galileo [2], Goblin [14], O₂ [17], and TM/FM [4]. Furthermore, we formalise these database schemas in terms of underlying types, underlying constraints, and functional forms.
Informally, an object-oriented database schema is a class hierarchy, i.e., a set of classes related by a subclass relation. A class has a name, a set of superclasses, a set of attributes, a set of constraints, and a set of methods. An attribute has a name and a type, which can be a basic, set, or record type, or a class name. Hence, classes can be recursive. An update method has a name, a list of parameters, and a body, which consists of simple assignments. A query method has a name, a list of input parameters, a result parameter, and a body, which consists of simple assignments.

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

\[
\begin{align*}
\text{Hierarchy} &::= \text{Class}^+ \\
\text{Class} &::= \text{‘Class’} \ CN [ \text{‘Isa’} \ CN^+ ] \\
&\quad \quad \text{[‘Attributes’} \ \text{Att}^+ ] \\
&\quad \quad \text{[‘Constraints’} \ \text{Key}^+ ] \\
&\quad \quad \text{[‘Methods’} \ \text{Meth}^+ ] \\
&\quad \text{‘Endclass’} \\
\text{Att} &::= \text{AN ‘:’ Type} \\
\text{Type} &::= \text{BasicType | SetType | RecordType | CN} \\
\text{BasicType} &::= \text{‘integer’ | ‘rational’ | ‘string’} \\
\text{SetType} &::= \text{‘{‘ Type ’}’} \\
\text{RecordType} &::= \text{‘<’ FieldList ‘>’} \\
\text{FieldList} &::= \text{Field | Field ‘,’ FieldList} \\
\text{Field} &::= \text{L ‘,’ Type} \\
\text{Key} &::= \text{‘key’} \ \text{Dest}^+ \\
\text{Dest} &::= \text{AN | Dest ‘,’ L} \\
\text{Meth} &::= \text{UpMeth | QueMethod} \\
\text{UpMeth} &::= \text{MN ‘{‘ [ ParList ] ‘}’} \ =\ \text{UpAsnList} \\
&\quad \quad \text{MN ‘\{’ [ ParList ] ‘}’} \ =\ \text{UpAsnList} \\
\text{ParList} &::= \text{Par | Par ‘,’ ParList} \\
\text{Par} &::= \text{L ‘,’ BasicType} \\
\text{UpAsnList} &::= \text{UpAssign | UpAssign ‘,’ UpAsnList} \\
\text{UpAssign} &::= \text{UpDest :=’ UpSource | insert(’ UpSource ‘,’ UpDest ‘)} \\
\text{UpDest} &::= \text{Dest} \\
\text{UpSource} &::= \text{Source | Object ‘,’ MN ‘{‘ [ ActParList ] ‘}’} \\
\text{Source} &::= \text{‘self’} | \text{Term | Term ‘+’ Source | Term ‘–’ Source} \\
&\quad \quad \text{Term ‘×’ Source | Term ‘÷’ Source} \\
\text{Term} &::= \text{Var | Cons} \\
\text{Var} &::= \text{L | AN | Var ‘,’ L | Var ‘,’ AN} \\
\text{Object} &::= \text{‘self’ | Var} \\
\text{ActParList} &::= \text{Source | Source ‘,’ ActParList} \\
\text{QueMeth} &::= \text{MN ‘{‘ [ ParList ] ‘→’ Result ‘}’} \ =\ \text{QueAsnList} \\
&\quad \quad \text{MN ‘\{’ [ ParList ] ‘→’ Result ‘}’} \ =\ \text{QueAsnList} \\
\text{Result} &::= \text{L ‘,’ Type} \\
\text{QueAsnList} &::= \text{QueAssign | QueAssign ‘,’ QueAsnList} \\
\text{QueAssign} &::= \text{QueDest :=’ QueSource | insert(’ QueSource ‘,’ QueDest ‘)} \\
\text{QueDest} &::= \text{L ‘,’ QueDest} \\
\text{QueSource} &::= \text{Source | new(’ CN ‘, NewParList ‘)} \\
\text{NewParList} &::= \text{NewPar | NewPar ‘,’ NewParList} \\
\text{NewPar} &::= \text{AN ‘=’ UpSource | AN ‘= nself’}
\end{align*}
\]
A class hierarchy is well-defined if it satisfies four conditions. The first condition is that the Isa relation is cyclic (see subsection on underlying types), and classes have a unique name and only refer to classes in the class hierarchy. The second is that attributes have a unique name within their class and are well-typed (see subsection on underlying types). The third is that keys are well-defined (see subsection on underlying constraints). The fourth is that methods have a unique name within their class and are well-typed (see subsection on functional forms).

2.1 Underlying types

In this subsection, we define underlying types of classes, extensions of underlying types, 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, every class is abbreviated to a 5-tuple \( C = (c, S, A, K, M) \), where \( c = name(C) \) is its name, \( S = sup\_names(C) \) is the set of the names of its superclasses, \( A \) is the set of its new attributes, \( K \) is the set of its new keys, and \( M \) is the set of its new methods. Furthermore, every class hierarchy is abbreviated to a set that contains the abbreviations of the classes in the hierarchy.

Informally, the set of all attributes of a class consists of both the new and inherited attributes. In order to formalise this, first, we formalise the Isa relation between classes.

Definition 2 (Subclass relation). Let \( H \) be an abbreviated class hierarchy satisfying the first condition for well-defined class hierarchies. The subclass relation on \( H \), denoted by \( sub(H) \), is defined as the reflexive and transitive closure of:

\[
isa(H) = \{(name(C_1), name(C_2)) \mid C_1 \in H \land C_2 \in H \land name(C_2) \in sup\_names(C_1)\}.
\]

Relation \( isa(H) \) is acyclic if its transitive closure only contains pairs of the form \((c_1, c_2)\), such that \( c_1 \neq c_2 \).

In general, object-oriented data models allow multiple inheritance of attributes, which means that attributes can be inherited from several subclasses. The mechanism defining how attributes are inherited imposes restrictions on multiple inheritance. If the inheritance mechanism is simple, then the restriction on multiple inheritance is severe.

Example 2. Let \( H \) be an abbreviated class hierarchy and \( C = (c, S, A, K, M) \) be a class in \( H \). Let us define the set of all attributes of \( C \) as

\[
attributes(C) = A \cup \{a : T \mid \exists C' \in H \ [(name(C), name(C')) \in isa(H)] \land \ a : T \in attributes(C') \land \forall a' : T' \in A[a \neq a']\}.
\]

According to this simple inheritance mechanism, class ‘TA’ in the following class hierarchy has two different attributes with name ‘address’.
Class Student
Attributes
  name : string
  address : <house:integer,street:string,city:string>
End class
Class Employee
Attributes
  name : string
  address : <house:integer,zip:integer>
End class
Class TA Isa Student Employee
End class.

Since we require that attribute names must be unique, the class hierarchy is not well-defined. □

If we use this simple inheritance mechanism, then a necessary condition for class hierarchies to be well-defined is that the types of inherited attributes with the same name must be the same. If we want to relax this restriction, then we must use a more complex inheritance mechanism.

Definition 3 (Attributes). Let H be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and \( C = (c, S, A, K, M) \) be a class in \( H \). The set of all attributes of \( C \), denoted by \( \text{atts}(C) \), is defined as:
\[
\text{atts}(C) = A \cup \{ a : T | \text{inherits}(a) \land T = \cap\{ T' | \text{inherits}(a, T') \land \forall a' : T' \in A[a \neq a'] \} \}
\]
where
\[
\text{inherits}(a, T') = \exists C' \in H [(\text{name}(C), \text{name}(C')) \in \text{isa}(H) \land a : T' \in \text{atts}(C')],
\]
\[
\text{inherits}(a) = \exists C' \in H \exists a' : T' \in \text{atts}(C') [(\text{name}(C), \text{name}(C')) \in \text{isa}(H) \land a = a'],
\]
and meet operator \( \cap \) is defined in Appendix A. Since we require that the Isa relation is acyclic, \( \text{atts} \) is well-defined. An attribute \( a : T \in \text{atts}(C) \) is well-typed if \( T \in \text{WTypes} \), where \( \text{WTypes} \) is the smallest set, such that:
1. if \( c \in CN \) is a class name, then \( c \in \text{WTypes} \)
2. if \( B \in \{\text{integer}, \text{rational}, \text{string}\} \) is a basic type, then \( B \in \text{WTypes} \)
3. if \( U \in \text{WTypes} \) is a well-defined type, then \( \{U\} \in \text{WTypes} \)
4. if \( \{U_1, \cdots, U_n\} \subseteq \text{WTypes} \) is a set of well-defined types and \( \{l_1, \cdots, l_n\} \subseteq L \) is a set of \( n \) distinct labels, then \( < l_1 : U_1 \cdots , l_n : U_n > \in \text{WTypes} \).

\( \square \)

Note that \( \perp \notin \text{WTypes} \). It follows that, in the case of single inheritance, an attribute that is well-typed in class \( C \) is also well-typed in every subclass of \( C \). Since we will use the inheritance mechanism of Definition 3 and we do not allow general redefinition of attributes, a necessary condition for class hierarchies to be well-defined is that inherited attributes with the same name must have a meet.

Example 3. Let \( H \) be the class hierarchy of Example 2. Let \( C \) be class ‘TA’. According to the inheritance mechanism of Definition 3, the attributes of class ‘Student’, of class ‘Employee’, and of class ‘TA’, viz.,
\[
\text{atts}(C) = \{\text{name:}\text{string, address:<house:integer, street:string, city:string, zip:integer>}\},
\]
are well-typed. Hence, \( H \) is well-defined. □
Every class in a class hierarchy has an underlying type, i.e., a type that describes the structure of the objects in its extensions (cf. TM/FM [4, 5]). Since a class can refer to other classes, its underlying type depends on the class hierarchy as a whole. The underlying type of a class is an aggregation of its attributes, where recursive types [3] are used to cope with attributes that refer to classes.

**Definition 4 (Underlying types).** First, we postulate a new type ‘oid’, whose extension is an enumerable set of object identifiers. Let H be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and attributes have a unique name within their class and are well-typed. Furthermore, let 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:

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

where

$$\tau(d, \eta) = \mu \text{. } t_d . \langle \text{id : oid, } a_1 : \tau(T_1, \eta \cup \{d\}), \ldots, a_k : \tau(T_k, \eta \cup \{d\}) \rangle$$

if \(d \notin \eta\) and \(\exists D \in H [\text{name}(D) = d \land \text{atts}(D) = \{a_1 : T_1, \ldots, a_k : T_k\}]\),

$$\tau(d, \eta) = t_d$$ if \(d \in \eta\),

$$\tau(B, \eta) = B$$ if \(B \in \{\text{integer, rational, string}\},

$$\tau(\{U\}, \eta) = \{\tau(U, \eta)\},$$

$$\tau(< l_1 : U_1, \ldots, l_n : U_n >, \eta) = < l_1 : \tau(U_1, \eta), \ldots, l_n : \tau(U_n, \eta) >.$$ 

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 \(\tau(d, \eta) = t_d\) indicates a repetition of the recursive type. □

A natural equivalence relation for types is the following [15]: \(\tau\) is equivalent to \(\tau'\) if the tree representing \(\tau\) is equal to the tree representing \(\tau'\).

**Definition 5 (Type equivalence).** Let \(\tau\) be type \(\mu t. \langle \text{id : oid, } a_1 : \tau_1, \ldots, a_k : \tau_k \rangle\) and \(\tau'_i\) be \(\tau[t \setminus \tau]\) for \(i \in \{1, \ldots, k\}\). The tree representing \(\tau\), denoted by \(\text{struc}(\tau)\), is defined as:

\[
\begin{tikzpicture}
  \node (root) {\tau};
  \node (oid) [below left of=root] {\text{id}};
  \node (a1) [below right of=root] {\text{struc}(\tau'_1)};
  \node (ak) [below right of=a1] {\text{struc}(\tau'_k)};
  \node (a0) [below right of=root] {\text{oid}};
  \draw [->] (oid) -- (a0);
  \draw [->] (a0) -- (a1);
  \draw [->] (a1) -- (ak);
\end{tikzpicture}
\]

where

- \(\text{struc}(B)\) has only one node, labeled \(B\), if \(B \in \{\text{integer, rational, string}\}\)

- \(\text{struc}(\{v\})\) consists of a root, labeled \(\{\}\), a subtree \(\text{struc}(v)\), and an arrow, labeled \(\in\), from the root labeled \(\{\}\) to the root of \(\text{struc}(v)\),

- \(\text{struc}(< l_1 : v_1, \ldots, l_n : v_n >)\) consists of a root, labeled \(<\>\), subtrees \(\text{struc}(v_1), \ldots, \text{struc}(v_n)\), and arrows, labeled \(l_i\), one for each \(i \in \{1, \ldots, n\}\), from the root labeled \(<\>\) to the root of \(\text{struc}(v_i)\).

Now, let \(\tau\) and \(\tau'\) be arbitrary types. Equivalence of \(\tau\) and \(\tau'\) is defined as equality of their trees:

$$\tau \cong \tau' \iff \text{struc}(\tau) = \text{struc}(\tau').$$

□

**Example 4.** The underlying types of class ‘SimpleAddress’ and class ‘Address’ of Example 1 are given by:
\[ \tau_S = \mu \, l \cdot \sigma_S. \] 

\[ \equiv <\text{id:oid, house:integer, street:string, city:string}> \]

\[ \tau_A = \mu \, l \cdot \sigma_A. \] 

\[ \equiv <\text{id:oid, house:integer, street:string, city:string}> \]

\[ \tau_A = \mu \, l \cdot \sigma_A. \] 

\[ \equiv <\text{id:oid, house:integer, street:string, city:string}> \]

\[ \tau_A = \mu \, l \cdot \sigma_A. \] 

\[ \equiv <\text{id:oid, house:integer, street:string, city:string}> \]

Using the definition and the rule \( \mu t. \alpha \equiv \alpha[t \setminus \mu t. \alpha] \), where \( \alpha[t \setminus e] \) means that every occurrence of \( t \) in \( \alpha \) must be replaced by \( e \), we have obtained types that are equivalent to the underlying types in TM/FM. \( \Box \)

**Example 5.** The following class hierarchy introduces a class ‘Person’ and a class ‘Employee’, which inherits from class ‘Person’:

```
Class Person
  Attributes
    name : string
    mother : Person
    address : SimpleAddress
    holiday address : SimpleAddress
Endclass
Class Employee IsA Person
  Attributes
    company: string
    salary : integer
Endclass.
```

Let \( \alpha \) be \(<\text{id:oid, name:string, mother:} \_tP, \text{address:} \_tS, \text{holiday address:} \_tS>\), where \( tP \) is a type variable and \( tS \) is the underlying type of class ‘Simple Address’. The underlying types of class ‘Person’ and class ‘Employee’ are given by:

\[ \tau_P = \mu tP. \alpha \]

\[ \equiv <\text{id:oid, name:string, mother:} \mu tP. \alpha, \text{address:} \_tS, \text{holiday address:} \_tS>\],

\[ \tau_E = \mu tE. <\text{id:oid, name:string, mother:} \mu tP. \alpha, \text{address:} \_tS, \text{holiday address:} \_tS, \]

\[ \text{company: string, salary: integer}> \equiv <\text{id:oid, name:string, mother:} \mu tP. \alpha, \text{address:} \_tS, \text{holiday address:} \_tS, \]

\[ \text{company: string, salary: integer}> \]

Using the definition and the rule \( \mu t. \alpha \equiv \alpha[t \setminus \mu t. \alpha] \), we have obtained underlying types that are not equivalent to the underlying types in TM/FM, which are \(<\text{id:oid, name:string, mother:oid, address:oid, holiday address:oid}>\) and \(<\text{id:oid, name:string, mother:oid, address:oid, holiday address:oid, company: string, salary: integer}>\), respectively. \( \Box \)

The extension of a type is the set of closed terms of that type.

**Definition 6 (Extensions).** Let \( \tau \) be an arbitrary type. The set of terms of type \( \tau \), denoted by \( \text{terms}(\tau) \), is defined as follows:

\[ \text{terms}(B) = \{ b \in Cons \mid b \text{ has type } B \} \text{ if } B \in \{ \text{oid, integer, string, rational} \} \]

\[ \text{terms}(\{e\}) = \{ e \in \text{terms}(e) \mid e \text{ is finite} \} \]

\[ \text{terms}(<l_1 : v_1, \ldots, l_n : v_n>) = \]

\[ \{ <l_1 = e_1, \ldots, l_n = e_n > \mid e_1 \in \text{terms}(v_1) \land \cdots \land e_n \in \text{terms}(v_n) \} \]

\[ \text{terms}(\mu t. \alpha) = \text{Var}_t \cup \{ \mu x.e \mid x \in (\text{Var}_t \setminus \text{BV}(e)) \land e \in \text{terms}(\alpha[t \setminus \mu t. \alpha]) \} \]

where \( \text{Var}_t \) is an enumerable set of instance variables, disjoint from \( \text{Var}_s \) for \( s \neq t \), and \( \text{BV}(e) \) is the set of bounded variables in term \( e \):

\[ \text{BV}(b) = \emptyset \text{ if } b \in Cons, \]

\[ \text{BV}(y) = \emptyset \text{ if } y \in \text{Var}_s \text{ for some } s, \]

\[ \text{BV}(\{e_1, \ldots, e_n\}) = \text{BV}(<l_1 = e_1, \ldots, l_n = e_n>) = \text{BV}(e_1) \cup \cdots \cup \text{BV}(e_n), \]

\[ \text{BV}(\mu y.e) = \text{BV}(e) \cup \{ y \}. \]
The extension of type \( \tau \), denoted by \( \text{ext}(\tau) \), is defined as:

\[
\text{ext}(\tau) = \{ e \in \text{terms}(\tau) \mid FV(e) = \emptyset \}
\]

where \( FV(e) \) is the set of free variables in term \( e \):

\[
\begin{align*}
FV(b) &= \emptyset \text{ if } b \in \text{Cons}, \\
FV(y) &= \{ y \} \text{ if } y \in \text{Var}, \text{ for some } s, \\
FV(\{ e_1, \ldots, e_n \}) &= FV(< l_1 = e_1, \ldots, l_n = e_n >) = FV(e_1) \cup \cdots \cup FV(e_n), \\
FV(\mu x.e) &= FV(e) - \{ y \}.
\end{align*}
\]

A natural equality relation for closed terms is the following: \( e \) is equal to \( e' \) if the tree representing \( e \) is equal to the tree representing \( e' \).

**Definition 7 (Equality of terms).** Let \( e \) and \( e' \) be closed terms of arbitrary types. Equality of \( e \) and \( e' \) is defined as equality of their trees:

\[ e = e' \iff \text{struc}(e) = \text{struc}(e'), \]

where

- \( \text{struc}(b) \) has only one node, labeled \( b \), if \( b \in \text{Cons} \),
- \( \text{struc}(\emptyset) \) has only one node, labeled \( \emptyset \),
- \( \text{struc}(\{ e_1, \ldots, e_n \}) \) consists of a root, labeled \( \emptyset \), subtrees \( \text{struc}(e_1), \ldots, \text{struc}(e_n) \), and arrows, labeled \( \epsilon \), one for each \( i \in \{ 1, \ldots, n \} \), from the root labeled \( \emptyset \) to the root of \( \text{struc}(e_i) \),
- \( \text{struc}(< l_1 = e_1, \ldots, l_n = e_n >) \) consists of a root, labeled \( <> \), subtrees \( \text{struc}(e_1), \ldots, \text{struc}(e_n) \), and arrows, labeled \( l_i \), one for each \( i \in \{ 1, \ldots, n \} \), from the root labeled \( <> \) to the root of \( \text{struc}(e_i) \),
- \( \text{struc}(\mu x.e) = \text{struc}(e[x \setminus \mu x.e]) \).

A natural subtype relation for underlying types is the following [3, 8]: \( \mu t.\alpha \) is a subtype of \( \mu t'.\alpha' \) if from the assumption that \( t \) is a subtype of \( t' \) it follows that for every field \( a : v' \) in \( \alpha' \) there is a field \( a : v \) in \( \alpha \), such that \( v \) is a subtype of \( v' \).

**Definition 8 (Subtyping).** Let underlying types \( \tau \) and \( \tau' \) be given by:

\[
\begin{align*}
\tau &= \mu t. < l_i : \tau_i \mid i \in I >, \\
\tau' &= \mu t'. < l_i : \tau'_i \mid i \in I' >.
\end{align*}
\]

Then \( \tau \) is a subtype of \( \tau' \), denoted by \( \tau \leq \tau' \), if

\[
I \supseteq I' \land \forall i \in I'[t \leq t' \Rightarrow \tau_i \leq \tau'_i],
\]

where

\[
\begin{align*}
B &\leq B \text{ if } B \in \{ \text{oid, integer, rational, string} \}, \\
\{ v \} &\leq \{ v' \} \text{ if } v \leq v', \\
< l_j : v_j \mid j \in J > &\leq < l_j : v'_j \mid j \in J' > \text{ if } J \supseteq J' \land \forall j \in J'[v_j \leq v'_j].
\end{align*}
\]

**Example 6.** Let \( \tau_S \) and \( \tau_A \) be the types given in Example 4 and \( \tau_P \) and \( \tau_E \) be the types given in Example 5. Using the subtype rules, it follows that \( \tau_A \) is a subtype of \( \tau_S \) and \( \tau_E \) is equivalent to a subtype of \( \tau_P \).
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 $\text{type}(T')$ is a subtype of $\text{type}(T)$. 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 condition for well-defined class hierarchies: every attribute must have a unique name within its class, every attribute must be well-typed, and the type of every inherited attribute must be a subtype of the types of the corresponding attributes in the superclasses.

2.2 Underlying constraints

In this subsection, we define underlying constraints of classes and class extensions.

Example 7. The following class hierarchy introduces a class ‘Person’ with a constraint and a class ‘Employee’, which inherits from class ‘Person’:

```plaintext
Class Person
Attributes
    name : string
    mother : Person
    address : SimpleAddress
    holiday_address : SimpleAddress
Constraints
    key name
Endclass
Class Employee Isa Person
Attributes
    company : string
    salary : integer
Endclass.
```

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

Definition 9 (Keys). Let $H$ be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and $C = (c, S, A, K, M)$ be a class in $H$. The set of all keys of $C$ is defined as:

$$\text{keys}(C) = K \cup \{ p \mid \exists C' \in H [(\text{name}(C), \text{name}(C')) \in \text{isa}(H) \land p \in \text{keys}(C')] \}.$$

A key $p$ in $\text{keys}(C)$ is well-defined if it is a sequence of labeling of paths in $\text{struct}(\text{type}(C))$, starting at the root. □

It follows that a key that is well-defined in class $C$ is also well-defined in every subclass of $C$.

Every class in a class hierarchy has an underlying constraint, i.e., a predicate that is satisfied by its extensions. The underlying constraint of a class is a conjunction of identifier uniqueness, partial referential integrity, and key uniqueness.

Definition 10 (Underlying constraints). Let $H$ be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and $C$ be a class in $H$. The set of extensions of class $C$ is defined as:

$$\text{exts}(C) = \{ e \subseteq \text{ext}(\text{type}(C)) \mid \text{constraint}(C, e) \},$$

where $\text{constraint}(C, e)$ is defined as the conjunction of

1. identifier uniqueness: $\forall x \in e \forall y \in e [x.id = y.id \Rightarrow x = y]$
2. partial referential integrity: one formula for every occurrence of \( \text{name}(C) \) in \( \text{atts}(C) \)

3. key uniqueness: one formula for every key \( p_1 \cdots p_n \) in \( \text{keys}(C) \):
   \[
   \forall x \in E \forall y \in E [(x.p_1 = y.p_1 \land \cdots \land x.p_n = y.p_n) \Rightarrow x = y].
   \]

The formulas for referential integrity are defined in Appendix B. □

### 2.3 Functional forms

In this subsection, we define functional forms of methods.

**Example 8.** The following class hierarchy introduces a class ‘Person’ with an update method ‘stay’ and a class ‘Employee’, which inherits from class ‘Person’:

```
Class Person
Attributes
   name : string
   mother : Person
   address : SimpleAddress
   holiday_address : SimpleAddress
Methods
   stay() = holiday_address := address
Endclass
Class Employee Isa Person
Attributes
   company : string
   salary : integer
Endclass.
```

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

**Definition 11 (Methods).** Let \( H \) be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and attributes have a unique name within their class and are well-typed, and \( C = (c, S, A, K, M) \) be a class in \( H \). The set of all methods of \( C \), denoted by \( \text{meths}(C) \), is defined as:

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

Let \( m(P \rightarrow l : T) = E \) be a query method in \( \text{meths}(C) \). The query method is well-typed if for every assignment \( d := s \) in \( E \):

1. \( d \) starts with \( l \) (i.e., only assignments to the result of the method)
2. if \( s \) is \( \text{new}(c', a_1 = e_1, \cdots, a_n = e_n) \), such that \( c' \) is the name of \( C' \in H \), then:
    \( a \) \( (\text{name}(C), c') \) is an element of \( \text{sub}(H) \)
    \( \text{i.e., only creation of objects in class } C \text{ and superclasses of } C \)
    \( b \) \( | \text{atts}(C') | = n \)
    \( c \) for every \( i \in \{1, \cdots, n\} \), \( a_i : T_i \) is an attribute in \( \text{atts}(C') \) and the type of \( e_i \) is a subtype of \( T_i \) according to subtype relation \( \leq_H \)
3. the type of \( s \) is a subtype of the type of \( d \) according to subtype relation \( \leq_H \)
   \( \text{i.e., only assignments of objects that belong to the corresponding class and its subclasses) \)
where the types of sources and destinations are defined in Appendix C, and subtype relation \( \leq_H \)
is an extension of subclass relation \( \text{sub}(H) \):
\[ d \leq_H d' \text{ if } (d, d') \in \text{sub}(H) \]
\[ B \leq_H B' \text{ if } B \in \{ \text{oid, integer, rational, string} \}, \]
\[ \{U\} \leq_H \{U'\} \text{ if } U \leq_H U', \]
\[ < l_j : U_j \mid j \in J > \leq_H < l'_j : U'_j \mid j \in J' > \text{ if } J \supseteq J' \land \forall j \in J' \{U_j \leq_H U'_j\}. \]

Now, let \( m(P) = E \) be an update method in \( \text{methods}(C) \). The update method is well-typed if for every assignment \( d := s \) in \( E \), the type of \( s \) is a subtype of the type of \( d \) (according to \( \leq_H \)) and if for every method call \( s.m'(v_1, \ldots, v_n) \) in \( E \), the following holds: the type of \( s \) must be \( \text{name}(D) \) for some \( D \in H \) and there must be a method \( m'(P' \rightarrow l' : T') = E' \) in \( \text{methods}(D) \), such that the type of each \( v_i \) is a subtype (according to \( \leq_H \)) of the type of the corresponding formal parameter in \( P' \). \( \square \)

Since we will use the inheritance mechanism of Definition 11 for methods, a necessary condition for class hierarchies to be well-defined is that inherited methods with the same name must be the same or must be redefined.

**Example 9.** According to the inheritance mechanism of Definition 11, class ‘Employee’ in the following class hierarchy has one method, viz., ‘stay() = holiday address := address’.

**Class** Person

**Attributes**
- name : string
- mother : Person
- address : SimpleAddress
- holiday address : SimpleAddress

**Methods**
- stay() = holiday address := address

**Endclass**

**Class** Employee **Isa** Person

**Attributes**
- holiday address : Address
- company : string
- salary : integer

**Endclass.**

Although method ‘stay’ is well-typed in class ‘Person’, it is not well-typed in class ‘Employee’, because attribute ‘holiday address’ has been specialised (if we do not allow attribute specialisation, then every method that is well-typed in class \( C \) is also well-typed in every subclass of \( C \)). Hence, class ‘Employee’ is not well-defined. One way to repair this is to use a different inheritance mechanism for methods, where ill-typed methods are redefined. However, we do not allow general redefinition of methods. Nevertheless, class ‘Employee’ can be redefined as a well-defined class by specialising attribute ‘address’ in the same way as attribute ‘holiday address’, i.e., adding attribute ‘address Address’ to the new attributes of class ‘Employee’. For a more complicated example, let class ‘Person’ and class ‘AgedPerson’ be given by:

**Class** Person

**Attributes**
- name : string
- address : SimpleAddress
- mother : Person
- grandmother : Person

**Methods**
- define grandmother () =
  grandmother := mother.mother1()
- mother1 (→ p:Person) =
  p := mother
Endclass

Class AgedPerson Isa Person

Attributes
address : Address
grandmother : AgedPerson

Endclass.

Method 'define_grandmother' is well-typed in class 'Person', but not in class 'AgedPerson', because attribute 'grandmother' has been specialised. Hence, class 'AgedPerson' is not well-defined. However, class 'AgedPerson' can be redefined as a well-defined class by specialising both attribute 'mother' and method 'mother1'.

Class AgedPerson Isa Person

Attributes
address : Address
mother : AgedPerson
grandmother : AgedPerson

Methods
define_grandmother () =
  grandmother := mother.mother1()
mother1 (-> p: AgedPerson) =
p := mother

Endclass.

Every class in a class hierarchy has a set of functional forms (one for each of its methods), i.e., a set of functions that describe the way in which the objects in its extensions are queried and updated (cf. TM/FM [4, 10]). The functional form of a query (resp., an update) method is a function of which the body is an accumulation of the assignments to the result (resp., to the input) of the method.

Definition 12 (Functional forms). Let $H = \{C_1, \cdots, C_h\}$ be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and attributes have a unique name within their class and are well-typed. Let $CN(H) = \{c_1, \cdots, c_h\}$ be the set of class names in $H$, $C$ be a class in $H$, $c$ be name$(C)$, and $\{a_1 : T_1, \cdots, a_k : T_k\}$ be atts$(C)$. Furthermore, let $meth = m(P \rightarrow l : T) = E$ be a query method in $meths(C)$. The functional form of query method $meth$ in class $C$, denoted by $func_q(C, meth)$, is defined as:

$$
func_q(C, meth) = \lambda objC : (CN(H) \rightarrow integer) \lambda (obj) : type(C) \lambda P .
\text{eval}_q(E)(obj, \rho_0(T), objcount),
$$

where

- $\rho_0(v, d) = \bot$ if $d \in CN$,
- $\rho_0(v, B) = \bot$ if $B \in \{\text{integer, rational, string}\}$,
- $\rho_0(v, \{U\}) = \bot$,
- $\rho_0(v, \{l_1 : U_1, \cdots, l_n : U_n\}) = \lambda l_1 = \rho_0(v, l_1, U_1), \cdots, l_n = \rho_0(v, l_n, U_n)$,

and syntactic evaluation function $\text{eval}_q$ is defined in Appendix D. Let $\lambda p_1 : \tau_1 \cdots \lambda p_n : \tau_n$ be $\lambda P$. The retrieval semantics of $meth$ are given by the function:

$$
\lambda db : \langle c_1 : \{type(C_1)\}, \cdots, c_h : \{type(C_h)\}\rangle > \lambda objC : type(C) \lambda p_1 : \tau_1 \cdots \lambda p_n : \tau_n.
\pi_q(F(db, obj)),
$$

where $F(db, obj)$ is $(func_q(C, meth))(\lambda z : CN(H), [db.x] | (obj)(p_1) \cdots (p_n))$ and $\pi_i$ is the projection of the $i$th component of a record.

Now, let $meth = m(P) = E$ be an update method in $meths(C)$. The functional form of update method $meth$ in class $C$, denoted by $func_u(C, meth)$, is defined as:
\[
\text{func}_u(C, \text{meth}) = \\
\lambda \text{objcount} : (CN(H) \rightarrow \text{integer}) \lambda (\mu x.e) : \text{type}(C) \lambda P.
\]
\[
eval_u(E)(\mu x. < e.id, a_1 = \sigma_0(a_1, T_1), \ldots, a_k = \sigma_0(a_k, T_k) >, \text{objcount}),
\]
where
\[
\sigma_0(v, d) = e.v \text{ if } d \in CN, \\
\sigma_0(v, B) = e.v \text{ if } B \in \{\text{integer, rational, string}\}, \\
\sigma_0(v, \{U\}) = e.v, \\
\sigma_0(v, < l_1 : U_1, \ldots, l_n : U_n >) = < l_1 = \sigma_0(v, l_1, U_1), \ldots, l_n = \sigma_0(v, l_n, U_n) >,
\]
and syntactic evaluation function \(\text{eval}_u\) is defined in Appendix D. Let \(\lambda p_1 : \tau_1 \cdots \lambda p_n : \tau_n\) be \(\lambda P\). The local update semantics of \(\text{meth}\) are given by the function:
\[
\lambda db : < c_1 : \{\text{type}(C_1)\}, \ldots, c_h : \{\text{type}(C_h)\} > \lambda \text{obj} : \text{type}(C) \lambda p_1 : \tau_1 \cdots \lambda p_n : \tau_n.
\]
\[
\pi_1(F(db, \text{obj}))
\]
where \(F(db, \text{obj}) = (\text{func}_u(C, \text{meth}))(\lambda x : CN(H), | db, x ||(obj)(p_1) \cdots (p_n))\).

\[\Box\]

**Example 10.** Let \(H\) be the class hierarchy of Example 9. Let \(C_P\) be class ‘Person’ and \(C_E\) be class ‘Employee’. The local update semantics of method ‘stay’ in class ‘Person’ and class ‘Employee’ are given by:
\[
\lambda db : < \text{Person} : \{\text{type}(C_P)\}, \text{Employee : \{type}(C_E)\} > \lambda (\mu x.e) : \text{type}(C_P).
\]
\[
\mu x. < \text{id} = e.id, \text{name} = e.name, \text{mother} = e.mother, \text{address} = e.address, \text{holiday address} = e.address >,
\]
\[
\lambda db : < \text{Person} : \{\text{type}(C_P)\}, \text{Employee : \{type}(C_E)\} > \lambda (\mu x.e) : \text{type}(C_E).
\]
\[
\mu x. < \text{id} = e.id, \text{name} = e.name, \text{mother} = e.mother, \text{address} = e.address, \text{holiday address} = e.address, \text{company} = e.company, \text{salary} = e.salary >.
\]

\[\Box\]

### 3 Schema transformations

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

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

**Definition 13 (Basic type transformations).** Let \(L'\) be the union of \(L\) and \(AN\) and \(Types\) be the set of types. Renaming is defined as a function of type \(L' \rightarrow L' \rightarrow Types \rightarrow Types\):

\[
\text{rename}(l')(l)(B) = B \text{ if } B \in \{\text{id, integer, rational, string}\}
\]

\[
\text{rename}(l')(l)(\{\tau\}) = \{\tau\}
\]

\[
\text{rename}(l')(l)(< l_1 : \tau_1, \ldots, l_n : \tau_n >) = < l_1[l' \setminus l] : \tau_1, \ldots, l_n[l' \setminus l] : \tau_n >,
\]

\[
\text{rename}(l')(l)(\mu t. < l_1 : \tau_1, \ldots, l_n : \tau_n >) = \mu t. < l_1[l' \setminus l] : \tau_1, \ldots, l_n[l' \setminus l] : \tau_n >
\]

if \(l' = l\),

\[
\text{rename}(l')(l)(\mu t. < l : \tau >) = \mu t. < l[l' \setminus l] : \tau >
\]

if \(l' \neq l\).

Note that we do not allow renaming of id-fields.

We distinguish between two kinds of aggregation: simple tupling and aggregation within a record type. Simple tupling is defined as a function of type \(L' \rightarrow Types \rightarrow Types\):

\[
\text{tuple}(l)(\tau) = < l : \tau >.
\]

Aggregation within a record type is defined as a function of type \(\{L'\} \rightarrow L' \rightarrow Types \rightarrow Types\):
aggregate({l, l_{i+1}, \ldots, l_j})(l)(B) = B \text{ if } B \in \{\text{oid, integer, rational, string}\)

aggregate({l, l_{i+1}, \ldots, l_j})(\{\}) = \{\tau\}

aggregate({l, l_{i+1}, \ldots, l_j})(l)(<\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>) = <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>

if \{l_{i+1}, \ldots, l_j\} \not\subseteq \{l_1, \ldots, l_n\},

aggregate({l, l_{i+1}, \ldots, l_j})(l)(<\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>) = <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>

if \{l_{i+1}, \ldots, l_j\} \not\subseteq \{l_1, \ldots, l_n\},

aggregate({l, l_{i+1}, \ldots, l_j})(l)(\mu, <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>) = \mu, <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>

if \{l_{i+1}, \ldots, l_j\} \not\subseteq \{l_1, \ldots, l_n\},

aggregate({l, l_{i+1}, \ldots, l_j})(l)(\mu, <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>) = \mu, <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>

if \text{id} \not\in \{l_{i+1}, \ldots, l_j\} \text{ and } \{l_{i+1}, \ldots, l_j\} \subseteq \{l_1, \ldots, l_n\},

aggregate({l, l_{i+1}, \ldots, l_j})(l)(\mu, <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>) = \mu, <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>

if \text{id} \in \{l_{i+1}, \ldots, l_j\} \text{ and } \{l_{i+1}, \ldots, l_j\} \subseteq \{l_1, \ldots, l_n\}.

Objectification is defined as a function of type \(\text{Types} \rightarrow \text{Types}\):

objectify(B) = B \text{ if } B \in \{\text{oid, integer, rational, string}\}

objectify(\{\}) = \{\tau\}

objectify(<\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>) = <\text{id, oid, } l_{i+1}, \ldots, l_1 : \tau_n>

objectify(\mu, <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>) = \mu, <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>

if \exists i \in \{1, \ldots, n\}[i = \text{id}],

objectify(\mu, <\text{oid, } l_{i+1}, \ldots, l_1 : \tau_n>) = \mu, <\text{id, oid, } l_{i+1}, \ldots, l_1 : \tau_n>

if \forall i \in \{1, \ldots, n\}[i \neq \text{id}].

\square

Complex type transformations are obtained by combining basic type transformations.

Example 11. Type \(\sigma = <l_1 : \mu, <\text{id, oid, } l_1, l_2 : \tau_2, l_3 : \tau_3, l_5 : \tau_3>\) can be obtained from type
\(\sigma_1 = <l_1 : \tau_1, l_2 : \tau_2, l_3 : \tau_3>\) as follows:

\(\sigma_2 = \text{rename}(l_1)(l)(\sigma_1) = <l : \tau_1, l_2 : \tau_2, l_3 : \tau_3>\)

\(\sigma_3 = \text{aggregate}({l, l_2})(l)(\sigma_2) = <l : <\text{oid, } l_1, l_2 : \tau_2, l_3 : \tau_3, l_5 : \tau_3>\>

\(\sigma_4 = <l : \text{objectify}(\text{oid, } l_1, l_2 : \tau_2, l_3 : \tau_3), l_5 : \tau_3> = \sigma.\)

\square

If \(F\) is a type transformation and \(\tau\) is a type, then \(F\) induces a function from \(\text{struc}(\tau)\) to \(\text{struc}(F(\tau))\).

Definition 14 (Functions between trees). Let \(F\) be a rename operation \(\text{rename}(l_1)(l')\) and \(\tau\) be a type, such that \(F(\tau) \neq F(\tau)\), where \(\equiv\) is syntactical equality. Then \(F\) induces a function \(\Phi\) from the set of paths in \(\text{struc}(\tau)\) to the set of paths in \(\text{struc}(F(\tau))\). Function \(\Phi\) is defined by \(\Phi(p) = \varphi(p, \tau)\), where

\(\varphi(l, v) = l'\) if \(l = l_1\) and \(v \equiv \tau\)

\(\varphi(l, v) = l\) if \(l \neq l_1\) or \(v \neq \tau\)

\(\varphi(l_1, \ldots, l_n, v) = \varphi(l_1, v), \varphi(l_1, \ldots, l_n, v, l),\)

where

\(<\ldots, l : \tau, \ldots, l = \tau,\>

\((\mu.\alpha)[t] = (\alpha[l][t \mu.\alpha]).\)

Let \(F\) be a tupling operation \(\text{tuple}(l')\) and \(\tau\) be a type, such that \(F(\tau) \neq F(\tau)\). Then \(F\) induces a function \(\Phi\) from the set of paths in \(\text{struc}(\tau)\) to the set of paths in \(\text{struc}(F(\tau))\). Function \(\Phi\) is defined by \(\Phi(p) = \varphi(p, \tau)\), where

\(\square\)
\[ \varphi(l, v) = l \cdot l \text{ if } v \equiv \tau \]
\[ \varphi(l, v) = l \text{ if } v \not\equiv \tau \]
\[ \varphi(l_1, \ldots, l_n, v) = \varphi(l_1, v), \varphi(l_1, \ldots, l_n, v, l). \]

Let \( F \) be an aggregation operation \emph{aggregate}(\{l'_1, \ldots, l'_p\}) and \( \tau \) a type, such that \( F(\tau) \neq F(\tau) \). Then \( F \) induces a function \( \Phi \) from the set of paths in \( \text{struc}(\tau) \) to the set of paths in \( \text{struc}(F(\tau)) \). Function \( \Phi \) is defined by \( \Phi(p) = \varphi(p, \tau) \), where
\[ \varphi(l, v) = l \cdot l \text{ if } l \in \{l'_1, \ldots, l'_p\} \text{ and } v \equiv \tau \]
\[ \varphi(l, v) = l \text{ if } l \not\in \{l'_1, \ldots, l'_p\} \text{ or } v \not\equiv \tau \]
\[ \varphi(l_1, \ldots, l_n, v) = \varphi(l_1, v), \varphi(l_1, \ldots, l_n, v, l). \]

Let \( F \) be an objectify operation and \( \tau \) a type, such that \( F(\tau) \neq F(\tau) \). Then \( F \) induces a function \( \Phi \) from the set of paths in \( \text{struc}(\tau) \) to the set of paths in \( \text{struc}(F(\tau)) \), which is defined by \( \Phi(p) = p \).

Type transformations induce transformations on predicates.

\textbf{Definition 15 (Transformations on predicates).} Let \( F \) be a type transformation, \( \tau \) be a type, and \( \psi \) be a conjunction of formulas, such that each formula has one of the following forms:

1. uniqueness: \( \forall x \in e \forall y \in e \{x.p_1 = y.p_1 \land \cdots \land x.p_n = y.p_n \} \Rightarrow x = y \]
2. refl: \( \forall x \in e \forall x_1 \in x.p_1 \cdots \forall x_n \in x_{n-1}.p_n \exists y \in e [x_n.p_{n+1} = y] \).

Predicate \( F(\psi) \) is obtained from \( \psi \) by replacing each formula in \( \psi \) as follows:

1. for each uniqueness formula: for every \( p_i \), if \( p_i \) is a path in \( \text{struc}(\tau) \), then \( p_i \) is replaced by \( \Phi(p_i) \)
2. for each refl formula: for every \( p_i \), if \( p_1, \ldots, p_i \) is a path in \( \text{struc}(\tau) \), then \( p_i \) is replaced by \( q_i \), such that \( \Phi(p_1, \ldots, p_i) = \Phi(p_1, \ldots, p_{i-1}).e.q_i \),

where \( \Phi \) is the function from the set of paths in \( \text{struc}(\tau) \) to the set of paths in \( \text{struc}(F(\tau)) \) induced by \( F \).

Type transformations also induce transformations on functions.

\textbf{Definition 16 (Transformations on functions).} Let type \( \tau \) and function \( f \) be given by:

\[ \tau = \mu t. < id : \text{oid}, a : < l_1 : \tau_1, \cdots, l_n : \tau_n >> \]
\[ f = \lambda(\mu x.e) : \tau \lambda P. \mu x. < id = e_0, a = < l_1 = e_1, \cdots, l_n = e_n >>. \]

Let \( F \) be a rename operation for types, such that:

\[ F(\tau) = \mu t. < id : \text{oid}, a : \text{rename}(l_1)(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >>) >>. \]

Then \( F \) induces a function \( F(f) \) on \( F(\tau) \), defined by:

\[ F(f) = \lambda(\mu x.e) : F(\tau) \lambda P. \mu x. < id = \Phi(e_0), a = < l = \Phi(e_1), l_2 = \Phi(e_2), \cdots, l_n = \Phi(e_n) >>, \]

where \( \Phi \) is the function from \( \text{struc}(\tau) \) to \( \text{struc}(F(\tau)) \) induced by \( F \). Let \( F \) be a tupling operation for types, such that:

\[ F(\tau) = \mu t. < id : \text{oid}, a : \text{tuple}(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >>) >>. \]

Then \( F \) induces a function \( F(f) \) on \( F(\tau) \), defined by:

\[ F(f) = \lambda(\mu x.e) : F(\tau) \lambda P. \mu x. < id = \Phi(e_0), a = < l = < l_1 = \Phi(e_1), \cdots, l_n = \Phi(e_n) >> >>, \]
where Φ is the function from \( \text{struc}(\tau) \) to \( \text{struc}(F(\tau)) \) induced by \( F \). Let \( F \) be an aggregation operation for types, such that:

\[
F(\tau) = \mu t. < \text{id} : \text{oid}, a : \text{aggregate}(\{l_1, l_2, \ldots, l_p\})/(l < l_1 : \tau_1, \ldots, l_n : \tau_n >).
\]

Then \( F \) induces a function \( F(f) \) on \( F(\tau) \), defined by:

\[
F(f) = \lambda (\mu x. e) : F(\tau) \lambda P. \mu x. < \text{id} = \Phi(e_0), a = < l = < l_1 = \Phi(e_1), \ldots, l_p = \Phi(e_p), \ldots, l_n = \Phi(e_n) > >,
\]

where \( \Phi \) is the function from \( \text{struc}(\tau) \) to \( \text{struc}(F(\tau)) \) induced by \( F \). Let \( F \) be an objectify operation for types, such that:

\[
F(\tau) = \mu t. < \text{id} : \text{oid}, a : \text{objectify}(l < l_1 : \tau_1, \ldots, l_n : \tau_n ) >.
\]

Then \( F \) induces a function \( F(f) \) on \( F(\tau) \), defined by:

\[
F(f) = \lambda (\mu x. e) : F(\tau) \lambda P. \mu x. < \text{id} = \Phi(e_0), a = < l = e.a.\text{id}, l_1 = \Phi(e_1), \ldots, l_n = \Phi(e_n) > >,
\]

where \( \Phi \) is the function from \( \text{struc}(\tau) \) to \( \text{struc}(F(\tau)) \) induced by \( F \). □

Both the transformation for lexical attributes and the transformation for unstable subtypes from [13] can be obtained by composing one aggregation and one objectify operation.

**Example 12.** The following class hierarchy introduces a class Person, a class Employee, which inherits from class Person, and a class Company:

```plaintext
Class Person
  Attributes
    name : string
    street : string
    house : integer
    city : string
Endclass
Class Employee Isa Person
  Attributes
    employer : Company
    salary : integer
Endclass
Class Company
  Attributes
    name : string
Endclass
```

The underlying type of class Person is:

\[
\mu tp. <\text{id:oid}, \text{name:string}, \text{street:string}, \text{house:integer}, \text{city:string}>.
\]

The underlying type of class Person can be transformed into (using \text{aggregate} \( \{\text{street}, \text{house}, \text{city}\} \) (address)):

\[
\mu tp. <\text{id:oid}, \text{name:string}, \text{address:<street:string}, \text{house:integer}, \text{city:string}> >,
\]

which can be transformed into (using \text{objectify}):

\[
\mu tp. <\text{id:oid}, \text{name:string}, \text{address:<street:string}, \text{house:integer}, \text{city:string}> >.
\]

The composite transformation \( F \) is a variant of the transformation for lexical attributes from [13]. We can redefine class Person as a class (named Person1) that refers to a new class (named X).
Class Person1
Attributes
  name : string
  address : X
Endclass

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

Since the identities of the objects in class Person become the identities of the objects in the redefined class (Person1), the underlying constraint of class Person \((\text{constr})\) is preserved by the redefined class (w.r.t. \(F\)): the underlying constraint of the redefined class implies \(F(\text{constr})\). The underlying type of class Employee is:

\[
\mu \ t_E. \ <\text{id:oid, name:string, street:string, house:integer, city:string, employer:}\tau_C, \ \text{salary:integer}>,
\]

where \(\tau_C\) is the underlying type of class Company. The underlying type of class Employee can be transformed into (using \textit{aggregate} \((\{\text{id, name, street, house, city}\})\) (employee)):

\[
\mu \ t_Y. \ <\text{employee:}\mu \ t_E. \ <\text{id:oid, name:string, street:string, house:integer, city:string, employer:}\tau_C, \ \text{salary:integer}>, \ \text{employee:}\tau_C, \ \text{salary:integer}>,
\]

which can be transformed into (using \textit{objectify}):

\[
\mu \ t_Y. \ <\text{id:oid, employee:}\mu \ t_E. \ <\text{id:oid, name:string, street:string, house:integer, city:string, employer:}\tau_C, \ \text{salary:integer}>, \ \text{employee:}\tau_C, \ \text{salary:integer}>.
\]

The composite transformation \(F\) is a variant of the transformation for unstable subtypes from [13]. We can redefine class Employee as a ‘relation’ (named \(Y\)) that refers to a new class (named Employee1):

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

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

Since the identities of the objects in class Employee become the identities of the objects in class Employee1, and not the identities of the objects in the redefined class \((Y)\), the underlying constraint of class Employee \((\text{constr})\) is not preserved by the redefined class (w.r.t. \(F\)): the underlying constraint of the redefined class and the rule \('x.\text{employee}.id = y.\text{employee}.id \Rightarrow x.\text{employee} = y.\text{employee}'\) do not imply \(F(\text{constr})\)). Therefore, we introduce a key for class \(Y\):

Class Y1
Attributes
employee : Employee1
employer : Company
salary : integer

Constraints
   key employee

Endclass.

The underlying constraint of class Y1 and the rule ‘x.employee.id = y.employee.id ⇒ x.employee = y.employee’ do imply $F(\text{constr})$. □

4 Application of schema transformations

In the previous section, we defined type transformations and showed how they induce schema transformations. In this section, we show how behaviour of methods can be used to choose among a set of schema transformations.

First, we define the set of factors of a class. A factor of a class $C$ is a class that contains a part of the attributes, keys, and methods of $C$.

Definition 17 (Factors). Let $H$ be a well-defined class hierarchy and $C$ be a class in $H$. The set of factors of class $C$ is defined as:

$$\text{factors}(C) = \{(d, 0, A, K, M) \mid d \in CN \land A \subseteq \text{atts}(C) \land K \subseteq \text{keys}(C) \land M \subseteq \text{meths}(C) \land H \cup \{(d, 0, A, K, M)\} \text{ is well-defined}\}.$$ □

Second, we give an example to illustrate that a class can be transformed in several ways, using different factors and different transformations.

Example 13. Let class Employee be the following class:

Class Employee
Attributes
   name : string
   dob : Date
   street : string
   house : integer
   city : string
   employer : Company

Methods
   move (s:string,h:integer,c:string) =
      street := s; house := h; city := c

Endclass

and class Address be a factor of Employee:

Class Address
Attributes
   street : string
   house : integer
   city : string

Methods
   move (s:string,h:integer,c:string) =
      street := s; house := h; city := c

Endclass.

One option to transform class Employee is to redefine Employee as a subclass of Address (factorisation by specialisation):
Class Employee1 Isa Address
Attributes
    name : string
    dob : Date
    employer: Company
Endclass.

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

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

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

Now, let class Person be a factor of class Employee2:

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

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

Class Employee3 Isa Person
Attributes
    employer : Company
Endclass.

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

Class Employee4
Attributes
    person : Person1
employer : Company

Methods
move (s:string,h:integer,c:string) =
    person := person.new_person(s,h,c)

End class

Class Person1

Attributes
    name : string
    dob : Date
    address : Address2

Methods
    new_person (s:string,h:integer,c:string -> l:Person1)
    l := new(Person1, name=name, dob=dob, address=address.new_address(s,h,c))

End class.

Since the identities of the objects in class Employee2 become the identities of the objects in class Employee4, we redefine method ‘move’ to be applicable to objects in class Employee4.

Yet another option is to redefine class Employee2 as a relation involving class Person:

Class Employment

Attributes
    employee : Person
    employer : Company

Constraints
    key employee

End class.

Since the identities of the objects in class Employee2 become the identities of the objects in class Person, we do not redefine method ‘move’, because it is already applicable to objects in class Person.

Note that, as an employee is a person in the real world, it is likely that options one and three are more reasonable than option two, where an employer refers to a person (as one of its attributes).

□

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

Definition 18 (Relatedness ratios). Let $H$ be a well-defined class hierarchy, $C$ be a class in $H$, $e$ be $\text{name}(C)$, and $M$ be $\text{meths}(C)$. Furthermore, for $\text{meth} \in M$, let $\text{atts}(\text{meth})$ consist of the names of attributes of $C$ that occur in $\text{meth}$. Weak relatedness of a set of attributes $A \subseteq \{a : T \in \text{atts}(C)\}$ is defined as:

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

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

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

20
Isolation of a set of attributes \( A \subseteq \{ a \mid a : T \in atts(C) \} \) is defined as:

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

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

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

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

**Example 14.** Consider class Employee of Example 13. The weak and strong relatedness ratios for street, house, city and name, dob are given by:

\[
\begin{align*}
\text{strongrel}(\text{Employee, } \{\text{street, house, city}\}) &= 1 \\
\text{weakrel}(\text{Employee, } \{\text{street, house, city}\}) &= 1 \\
\text{strongrel}(\text{Employee, } \{\text{name, dob}\}) &= 0 \\
\text{weakrel}(\text{Employee, } \{\text{name, dob}\}) &= 0.
\end{align*}
\]

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

Now, consider class Employee2 of Example 13. The weak and strong relatedness ratios for name, dob, address and name, dob, employer are given by:

\[
\begin{align*}
\text{strongrel}(\text{Employee2, } \{\text{name, dob, address}\}) &= 0 \\
\text{weakrel}(\text{Employee2, } \{\text{name, dob, address}\}) &= 0 \\
\text{strongrel}(\text{Employee2, } \{\text{name, dob, employer}\}) &= 0 \\
\text{weakrel}(\text{Employee2, } \{\text{name, dob, employer}\}) &= 0.
\end{align*}
\]

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

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

**Example 15.** Consider class Employee2 of Example 13. The isolation evidence ratio for name, dob, address is given by:

\[
isolation(\text{Employee2, } \{\text{name, dob, address}\}) = 1.
\]

Redefinition as a relation results in a relation (Employment) that represents a simple association between a person and a company. Now, if we add a method to class Employee2 that updates attribute address and attribute employer, then we will have to add a method to Employment that creates a new person and updates attribute employee and attribute employer. However, this method inserts objects into a class different from the relation and should therefore not be associated with the relation. □
So, how do we choose factors and transformations? Factors are chosen by comparing weak evidence ratios. If the weak evidence ratio of a set of attributes is greater than some threshold, there is reason to assume that the attributes can be used as a factor. If not, there is no reason. Transformations are chosen by comparing strong evidence ratios and isolation ratios. In case the strong evidence ratio is greater than some threshold, delegation is a reasonable option, because the attributes are strongly related within the set and weakly related with other attributes. In case the isolation ratio is less than one, then specialisation is possible, but redefinition as a relation is not. Otherwise, specialisation or redefinition as a relation are both possible. It should be mentioned that, in the context of schema integration, schema transformations must be applied carefully and only if necessary. In particular, this is true for factorisation by specialisation, since a lot of new classes will be generated by this type of transformation.

The considerations for choosing factors and transformations can be used in a heuristic algorithm to support schema integration. First, the attributes of every class are partitioned in such a way that the isolation ratio of every element in the partition is one, and every class is factorised by delegation if desirable. Subsequently, for every pair of promising classes, a set of possible superclasses is computed, and both classes are factorised by specialisation or redefined as a relation if desirable.

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

\[
\text{integrate}(\text{DBS1}, \text{DBS2}, \text{TSR}, \text{TWR}) =
\begin{align*}
&\text{for every class } C \text{ in } \text{DBS1} \\
&\quad \text{for every element } A \text{ in } \text{partition}(C) \\
&\quad \quad \text{if } \text{strongrel}(c,A) \geq \text{TSR} \text{ and } 1 < |A| < |\text{atts}(C)| \quad \text{then create class } C_1 \text{ as the class containing } A \text{ and the methods that refer to } A; \\
&\quad \quad \quad \quad \text{factorise } C \text{ by delegation using } C_1; \\
&\quad \quad \quad \quad \text{mark } C \text{ and } C_1 \\
&\quad \quad \text{elif } \text{weakrel}(c,A) \geq \text{TWR} \quad \text{then mark } C \\
&\quad \quad \quad \quad \text{fi} \\
&\quad \quad \text{fi} \\
&\quad \text{od}; \\
&\text{for every marked } C_1 \text{ in } \text{DBS1} \\
&\quad \text{for every marked } C_2 \text{ in } \text{DBS2} \\
&\quad \quad \text{if there is a superclass } C \text{ of a class in } \text{joins}(C_1,C_2) \text{ that can be used} \\
&\quad \quad \quad \quad \text{as a factor according to the designer} \\
&\quad \quad \quad \quad \text{then transform}(C_1,C_2,C) \\
&\quad \quad \quad \quad \text{else for every key } a.p \text{ in } \text{keys}(C_1) \text{ such that } a.D \text{ in } \text{atts}(C_1) \text{ for some class } D \\
&\quad \quad \quad \quad \text{do define class } D_1 \text{ as obtained from } C_1 \text{ by applying} \\
&\quad \quad \quad \quad \quad \text{the inverse of redefinition as a relation;} \\
&\quad \quad \quad \quad \quad \text{if there is a superclass } C \text{ of a class in } \text{joins}(D_1,C_2) \text{ that can be used} \\
&\quad \quad \quad \quad \quad \quad \text{as a factor according to the designer} \\
&\quad \quad \quad \quad \quad \text{then transform}(D_1,C_2,C) \\
&\quad \quad \quad \quad \quad \text{fi} \\
&\quad \quad \quad \quad \text{fi} \\
&\quad \text{od} \\
&\text{od};
\end{align*}
\]

\[
\text{transform}(C_1,C_2,C) = \\
\begin{align*}
&\text{begin let } \varphi_1 \text{ be an injection from } \text{atts}(C) \text{ to } \text{atts}(C_1) \text{ induced by } C_1 \preceq C; \\
&\quad \text{let } \varphi_2 \text{ be an injection from } \text{atts}(C) \text{ to } \text{atts}(C_2) \text{ induced by } C_2 \preceq C; \\
&\quad \text{define } A_1 \text{ as the attribute names in the range of } \varphi_1;
\end{align*}
\]
define A2 as the attribute names in the range of \( \varphi_1 \);
if \( \text{isolation(name(C1),A1)} < 1 \) or \( \text{isolation(name(C2),A2)} < 1 \)
then factorise C1 and C2 by specialisation using C
elif \( 1 < |A1| < |\text{atts}(C1)| \) and \( 1 < |A2| < |\text{atts}(C2)| \)
then factorise C1 and C2 or redefine C1 and C2 as relations
according to the choice of the designer
else factorise C1 and C2 by specialisation using C
fi
end;

where partition(C) is constructed as follows:

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

and joins(D1,D2) (i.e., the set of common superclasses of D1 and D2) and \( \preceq \) (i.e., the subclass relation) are as defined in [25, 24]. □

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

**Example 16.** The following class hierarchy is a part of a drawing tool:

**Class** Square
**Attributes**
- pos:Pos
- width:int
**Methods**
- set \((x:\text{int}, y:\text{int}) =\)  
  \(\text{pos} := \text{pos.set.pos}(x,y)\)
- translate \((dx:\text{int}, dy:\text{int}) =\) 
  \(\text{pos} := \text{pos.translate.pos}(dx,dy)\)

**Endclass**
**Class** Pos
**Attributes**
- \(x,0:0\text{int}\)
- \(y,0:0\text{int}\)
**Methods**
- set_pos \((x:\text{int}, y:\text{int} \rightarrow p:\text{Pos}) =\) 
  \(p := \text{new}(\text{Pos}, x,0=x, y,0=y)\)
- translate_pos \((dx:\text{int}, dy:\text{int} \rightarrow p:\text{Pos}) =\) 
  \(p := \text{new}(\text{Pos}, x,0=x,0+dx, y,0=y,0+dy)\)
**Endclass.**

Objects in class Square have a position on the screen and a width, and can be moved around on the screen using method set and method translate.
The following class hierarchy is a part of another drawing tool:

**Class** Rectangle
**Attributes**
\[
p_{\text{pos}.x}.\text{int} \\
p_{\text{pos}.y}.\text{int} \\
\text{width}.x.\text{int} \\
\text{width}.y.\text{int}
\]

Methods

\[
\text{put}(x.\text{int}, y.\text{int}) = \\
\quad \text{pos}.x := x; \text{pos}.y := y
\]

\[
\text{move}(\text{delta}.x.\text{int}, \text{delta}.y.\text{int}) = \\
\quad \text{pos}.x := \text{pos}.x + \text{delta}.x; \text{pos}.y := \text{pos}.y + \text{delta}.y
\]

Endclass.

Objects in class Rectangle have a position w.r.t the x-axis, a position w.r.t. the y-axis, a length, and a width, and can be moved around on the screen using method put and method move.

Let \( C_S \) be class Square, \( C_P \) be class Pos, and \( C_R \) be class Rectangle. The partitions of these classes are given by:

\[
\begin{align*}
\text{partition}(C_S) &= \{\{\text{pos}\}, \{\text{width}\}\} \\
\text{partition}(C_P) &= \{\{x.\text{co}, y.\text{co}\}\} \\
\text{partition}(C_R) &= \{\{\text{pos}.x, \text{pos}.y\}, \{\text{width}.x\}, \{\text{width}.y\}\}.
\end{align*}
\]

Now, let us use Algorithm 1 to integrate the class hierarchies. In the first big loop, all classes are marked and Rectangle is factorised using delegation:

Class Rectangle

Attributes

\[
\begin{array}{c}
\text{pos}.x.\text{int} \\
\text{pos}.y.\text{int} \\
\text{width}.x.\text{int} \\
\text{width}.y.\text{int}
\end{array}
\]

Methods

\[
\text{put}(x.\text{int}, y.\text{int}) = \\
\quad \text{pos} := \text{pos}.x.\text{put}.\text{pos} (x, y) \\
\text{move}(dx.\text{int}, dy.\text{int}) = \\
\quad \text{pos} := \text{pos}.\text{move}.\text{pos} (dx, dy)
\]

Endclass

Class X

Attributes

\[
\begin{array}{c}
\text{pos}.x.\text{int} \\
\text{pos}.y.\text{int}
\end{array}
\]

Methods

\[
\text{put}._{\text{pos}}(x.\text{int}, y.\text{int} \rightarrow l.X) = \\
\quad l := \textbf{new}(X, \text{pos}.x=x, \text{pos}.y=y) \\
\text{move}._{\text{pos}}(dx.\text{int}, dy.\text{int} \rightarrow l.X) = \\
\quad l := \textbf{new}(X, \text{pos}.x=\text{pos}.x+dx, \text{pos}.y=\text{pos}.y+dy)
\]

Endclass.

Let \( C_R \) be the new class Rectangle and \( C_X \) be class X. In the second big loop, the only interesting join sets are:

\[
\begin{align*}
\text{joins}(C_P, C_X) &= \{\gamma_P(C_P)\} = \{\gamma_X(C_X)\} \\
\text{joins}(C_S, C_R) &= \{\gamma_S(C_S)\},
\end{align*}
\]

where the \( \gamma \)'s are rename functions replacing attribute names and method names.

Suppose the designer chooses \( C_P \) as a factor to factorise \( C_P \) and \( C_X \) (redefinition as a relation is not possible). Since \( C_P \) and \( C_X \) are equivalent, it suffices to remove \( C_X \) and redefine every class \( D \) that refers to \( C_X \) by replacing the occurrences of \( X \) in \( D \) by \( \text{Pos} \) and redefine every method in \( D \) in which an attribute or method name \( \text{name} \) occurs that refers to an attribute or method
of $C_X$ by replacing these occurrences of name by the corresponding attribute or method name in Pos. Let $C^v_R$ be the redefined class Rectangle and suppose the designer chooses the following class as a factor to factorise $C_S$ and $C^v_R$ (again, redefinition as a relation is not possible):

Class Figure
Attributes
pos:Pos
Methods
set (x:int, y:int) =
pos := pos.set_pos (x,y)
translate (dx:int, dy:int) =
pos := pos.translate_pos (dx,dy)
Endclass.

Then the integrated class hierarchy will consist of class Pos, class Figure, and:

Class Square Isa Figure
Attributes
width:int
Endclass
Class Rectangle Isa Figure
Attributes
width_x:int
width_y:int
Endclass.

Note that the only real choice made by the designer is the choice to factorise Square and Rectangle using Figure instead of Square. □

5 Conclusion

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

Further research includes extension of the data model, extension of the set of schema transformations, properties of the schema transformations, and extension and refinement of the heuristic algorithm.
A Meet types

Let \( H \) be a class hierarchy. Furthermore, let \( T \) and \( T' \) be types that occur in \( H \). The meet of \( T \) and \( T' \), denoted by \( T \sqcap T' \), is defined as follows (cf. [8]):

1. \( T, T' \in CN, T = \text{name}(D_1), T' = \text{name}(D_2) \):
   
   (a) \( T \sqcap T' = T \) if \( T = T' \),
   
   (b) \( T \sqcap T' = d \) is a new class name corresponding to class:

   \[
   (d, \emptyset, \{ a_1 : T_1 \in \text{atts}(D_1) | \forall a_2 : T_2 \in \text{atts}(D_2)[a_1 \neq a_2] \} \\
   \cup \{ a_2 : T_2 \in \text{atts}(D_2) | \forall a_1 : T_1 \in \text{atts}(D_1)[a_1 \neq a_2] \} \\
   \cup \{ b : U | \exists a_1 : T_1 \in \text{atts}(D_1) \exists a_2 : T_2 \in \text{atts}(D_2)[b = a_1 = a_2 \land U = T_1 \sqcap T_2] , \emptyset \}, \emptyset \}
   \]

   where \( T_1 \sqcap T_2 = d \) if
   
   \[
   T_1 = \text{name}(D_1) \land T_2 = \text{name}(D_2) \lor T_1 = \text{name}(D_2) \land T_2 = \text{name}(D_1),
   \]

   if \( \forall a_1 : T_1 \in \text{atts}(D_1) \forall a_2 : T_2 \in \text{atts}(D_2)[a_1 = a_2 \Rightarrow T_1 \sqcap T_2 \neq \bot] \)
   
   (c) \( T_1 \sqcap T_2 = \bot \) otherwise

2. \( T, T' \in \{ \text{integer}, \text{rational}, \text{string} \} \): \( T \sqcap T' = T \) if \( T = T' \) and \( T \sqcap T' = \bot \) otherwise

3. \( T = \{ U \}, T' = \{ U' \} \): \( T \sqcap T' = \{ U \sqcap U' \} \) if \( U \sqcap U' \neq \bot \) and \( T \sqcap T' = \bot \) otherwise

4. \( T = < l_i : U_i | i \in I >, T' = < l'_i : U'_i | i \in I' > \):
   
   (a) \( T \sqcap T' = \\
   < l_i : U | (i \in I - I' \land U = U_i) \lor (i \in I' - I \land U = U'_i) \lor (i \in I \land I \land U = U_i \sqcap U'_i) > \)

   if \( \forall i \in I \land I \land U = U_i \land U'_i \neq \bot \)
   
   (b) \( T \sqcap T' = \bot \) otherwise

5. otherwise: \( T \sqcup T' = \bot \).

Since \( \sqcap \) is commutative and associative (modulo class renaming), \( \sqcap \{ T_1, \cdots, T_n \} = T_1 \sqcap \cdots \sqcap T_n \) is well-defined.

B Referential integrity formulas

Let \( H \) be a class hierarchy, \( C \) be a class in \( H \), \( c \) be \( \text{name}(C) \), and \( a : T \) be an attribute in \( \text{atts}(C) \) that refers to \( C \). Furthermore, let \( T^* \) be a type obtained from \( T \) by labelling one occurrence of \( c \) with \( \ast \). The access path to \( c \ast \) in \( T^* \), denoted by \( \text{path}(c \ast, T^*) \), is defined as follows:

\[
\begin{align*}
\text{path}(c \ast, c \ast) &= \text{empty} \\
\text{path}(c \ast, \{ c \ast \}) &= \in \\
\text{path}(c \ast, \{ l : c \ast, \cdots \ast \}) &= \in \\
\text{path}(c \ast, \{ U^* \}) &= \in . \text{path}(U^*) \\
\text{path}(c \ast, \{ \cdots, l : U^*, \cdots \ast \}) &= l . \text{path}(U^*) ,
\end{align*}
\]

where \( c \ast \) occurs in \( U^* \). Then every occurrence of \( c \) in \( a : T \) is uniquely defined by its access path.

Finally, the referential integrity formula corresponding to an occurrence of \( c \) in \( a : T \) with access path \( p \) is defined as:

\[
\forall x \in e \left[ \text{refint}(x.a,p,0) \right]
\]

where

\[
\begin{align*}
\text{refint}(p, \text{empty}, i) &= \exists y \in e \left[ p = y \right] \\
\text{refint}(p, l, i) &= \exists y \in e \left[ p, l = y \right] \\
\text{refint}(p, e, i) &= \forall x_{i+1} \in p . \exists y \in e . [ x_{i+1} = y ] \\
\text{refint}(p, l, q, i) &= \text{refint}(p, l, q, i) \\
\text{refint}(p, e, . q, i) &= \forall x_{i+1} \in e . \text{refint}(x_{i+1}, q, i + 1).
\end{align*}
\]

26
C Types of sources and destinations

Let \( H \) be a class hierarchy and \( C \) be a class in \( H \). Furthermore, let \( m(P \rightarrow l : T) = E \) be a query method in \( \text{meths}(C) \). We define the type of a source or destination of an assignment in \( E \) as follows:

- **the type of** \( \text{self} \) **is** \( \text{name}(C) = c \)
- if \( s : B \) is a parameter in \( P \), then the type of \( s \) is \( B \)
- if \( s = l.r_1, \ldots, r_n \) is the labeling of a path in \( \text{struc}(< l : \text{type}(T) >) \), starting at the root, then the type of \( s \) is \( T_r_1, \ldots, r_n \)
- if \( s = a_i.r_1, \ldots, r_n \) is the labeling of a path in \( \text{struc}(\text{type}(C)) \), starting at the root, then the type of \( s \) is \( T_i.r_1, \ldots, r_n \)
- **the type of a constant** \( v \in \text{ext}(B) \) **is** \( B \)
- if \( s \) is an expression, then the type of \( s \) follows from the types of the subexpressions and the standard rules for + (addition for integers and rationals; concatenation for strings), − (subtraction), × (multiplication), and ÷ (division)
- **the type of** \( \text{new}(d,e_1, \ldots, e_n) \) **is** \( d \)
- otherwise, the type of a source or destination is undefined

where

\[
\langle \ldots, l = T_1, \ldots > l = T
\]

\( c'.a = T \) if \( \exists C' \in H [\text{name}(C') = c' \land a : T \in \text{atts}(C')] \).

Now, let \( m(P) = E \) be an update method in \( \text{meths}(C) \). The type of a source or destination of an assignment in \( E \) is defined as above, with the exception that the third item (regarding selections of the result of a query method) is removed and the following item (regarding method calls) is added:

the type of a call to method \( m'(P' \rightarrow l' : T') = E' \) is \( T' \).

D Syntactic evaluation of method bodies

Let \( H \) be a class hierarchy, \( C \) be a class in \( H \), \( c \) be \( \text{name}(C) \), and \( \{a_1 : T_{a_1}, \ldots, a_k : T_{a_k}\} \) be \( \text{atts}(C) \). Furthermore, let \( m(P \rightarrow l : T) = E \) be a query method in \( \text{meths}(C) \). The syntactic evaluation of body \( E \) in state \( \sigma = (\text{obj}, f) \) (where \( \text{obj} \) is the object that is queried by the method, \( r \) is the current value of the result of the method, and \( f \) is the function that, for every class \( C \) in \( H \), gives the current number of objects in class \( C \)), denoted by \( \text{eval}(E)\sigma \), is given by:

\[
\text{eval}_q(L_1; L_2)\sigma = \text{eval}_q(L_2)((\text{eval}_q(L_1)\sigma),
\]

\[
\text{eval}_q(l := s)(\text{obj}, r, f) = (\text{obj}, \pi_1(\text{eval}(s)(\text{obj}, r, f)), \pi_2(\text{eval}(s)(\text{obj}, r, f))),
\]

\[
\text{eval}_q(l_1, \ldots, l_n := s)(\text{obj}, r, f) = (\text{obj}, r[l_1, \ldots, l_n = \pi_1(\text{eval}(s)(\text{obj}, r, f))], \pi_2(\text{eval}(s)(\text{obj}, r, f))),
\]

\[
\text{eval}_q(\text{insert}(s, l))(\text{obj}, r, f) = (\text{obj}, r \cup \{\pi_1(\text{eval}(s)(\text{obj}, r, f))\}, \pi_2(\text{eval}(s)(\text{obj}, r, f))),
\]

\[
\text{eval}_q(\text{insert}(s, l_1, \ldots, l_n))(\text{obj}, r, f) = (\text{obj}, r[l_1, \ldots, l_n = r[l_1, \ldots, l_n \cup \{\pi_1(\text{eval}(s)(\text{obj}, r, f))\}]],
\]

\[
\pi_2(\text{eval}(s)(\text{obj}, r, f))),
\]

where

\[
\text{eval}(\text{self})(\text{obj}, r, f) = (\text{obj}, f),
\]

\[
\text{eval}(a.l_1, \ldots, l_n)(\text{obj}, r, f) = (\text{obj}.a.l_1, \ldots, l_n, f)
\]

if \( a \in \text{AN} \),

\[
\text{eval}(l_1, \ldots, l_n)(\text{obj}, r, f) = (l_1, \ldots, l_n, f)
\]

27
if \( l_1 \in L \),
\[ ev(b)(\text{obj}, r, f) = (b, f) \]
if \( b \in \text{Cons} \)
\[ ev(s_1 \theta s_2)(\text{obj}, r, f) = (\pi_1(ev(s_1)(\text{obj}, r, f)) \theta \pi_1(ev(s_2)(\text{obj}, r, f)), f) \]
if \( \theta \in \{+r, -, \times, \div\} \)
\[ ev(\text{new}(c, a_1 = e_1' \ldots a_k = e_k'))(\text{obj}, r, f) = \]
\[ (\mu y. < \text{id} = \text{newid}(c, f(c)), a_1 = \pi_1(ev(e_1')(\text{obj}, r, f)), \ldots a_k = \pi_1(ev(e_k')(\text{obj}, r, f)) >, f') \]
where \( \text{newid} : \text{CN}(H) \times \text{integer} \rightarrow \text{oid} \) is some injective function
and \( f' \) is equal to \( f \), except that \( f'(c) = f(c) + 1 \)
and \( ev(\text{self})(\text{obj}, r, f) = (y, f) \)
and
\[ < l_1 = v_1, \ldots, l_n = v_n > [l_1 = v] = \]
\[ < l_1 = v_1, \ldots, l_n = v_n > [l_1, l_2, \ldots, l_n = v] = \]
\[ < l_1 = v_1[l_1, l_2, \ldots, l_n = v] >, \ldots, l_n = v_n >. \]
Now, let \( m(P) = E \) be an update method in \( \text{meths}(C) \). The syntactic evaluation of body \( E \) in state \( \sigma = (\mu x.e, f) \) (where \( \mu x.e \) is the object that is updated by the method and \( f \) is the function that, for every class \( C \) in \( H \), gives the current number of objects in class \( C \)), denoted by \( ev(E)\sigma \), is given by:
\[ ev_u(L_1; L_2)\sigma = ev_u(L_2)(ev_u(L_1)\sigma), \]
\[ ev_u(a_1 := s)(\mu x.e, f) = \]
\[ (\mu x.(\text{reduce}_x(< \text{id} = e_0, a_1 = \pi_1(ev(s)(\mu x.e, f)), \ldots, a_k = e_k >)), \pi_2(ev(s)(\mu x.e, f))), \]
\[ ev_u(a_1, a_2, \ldots, l_n := s)(\mu x.e, f) = \]
\[ (\mu x.(\text{reduce}_x(< \text{id} = e_0, a_1 = e.a_1[l_1, \ldots, l_n = \pi_1(ev(s)(\mu x.e, f))), \ldots, a_k = e_k >)), \]
\[ \pi_2(ev(s)(\mu x.e, f))), \]
\[ ev_u(\text{insert}(s, a_1))(\mu x.e, f) = \]
\[ (\mu x.(\text{reduce}_x(< \text{id} = e_0, a_1 = e.a_1 l_1, \ldots, l_n = \pi_1(ev(s)(\mu x.e, f))), \ldots, a_k = e_k >)), \]
\[ \pi_2(ev(s)(\mu x.e, f))), \]
\[ ev_u(\text{insert}(s, a_1, l_1, \ldots, l_n))(\mu x.e, f) = \]
\[ (\mu x.(\text{reduce}_x(< \text{id} = e_0, a_1 = e.a_1, l_1, \ldots, l_n = \pi_1(ev(s)(\mu x.e, f))), \ldots, a_k = e_k >)), \]
\[ \pi_2(ev(s)(\mu x.e, f))), \]
\[ ev(\text{self})(\mu x.e, f) = (x, f), \]
\[ ev(a, l_1, \ldots, l_n)(\mu x.e, r, f) = (e.a, l_1, \ldots, l_n, f) \]
if \( a \in \text{AN} \),
\[ ev(l_1, \ldots, l_n)(\mu x.e, r, f) = (l_1, \ldots, l_n, f) \]
if \( l_1 \in L \),
\[ ev(b)(\mu x.e, r, f) = (b, f) \]
if \( b \in \text{Cons} \)
\[ ev(s_1 \theta s_2)(\mu x.e, r, f) = (\pi_1(ev(s_1)(\mu x.e, r, f)) \theta \pi_1(ev(s_2)(\mu x.e, r, f)), f) \]
if \( \theta \in \{+r, -, \times, \div\} \)
\[ ev(s.m'(e_1', \ldots, e_n'))(\mu x.e, f) = \]
\[ \pi_2.s([[m'][[[f'][\pi_1(ev(s)(\mu x.e, f))](\pi_1(ev(e')(\mu x.e, f)))(\pi_1(ev(e'_n)(\mu x.e, f)))]))], \]
where \([[[m']] \) is the functional form of method \( m' \) in the class to which \( s \) refers
and
\[ \text{reduce}_x(e') = < l_1 = \text{reduce}_x(e'_1), \ldots, l_n = \text{reduce}_x(e'_n) > \]
if \( e' = < l_1 = e'_1, \ldots, l_n = e'_n > \) for some \( e'_1, \ldots, e'_n \);
\[ \text{reduce}_x(e') = x \] if \( e' = \mu x.e'_1 \) for some \( e'_1 \);
\[ \text{reduce}_x(e') = e' \] otherwise.
E Bibliography


