Object-Oriented Database Evolution

Jean-Bernard Lagorce, Arūnas Stočkus, Emmanuel Waller

{lagorce, stockus, waller, }@lri.fr
LRI, Université d'Orsay
91405 Orsay cedex, France

Abstract. An *evolution language* is composed of an instance update language, a schema update language, and a mechanism to combine them. We present a formal evolution language for object-oriented database management systems. This language allows to write programs to update simultaneously both the schema and the instance. Static checking of these programs insures that the resulting database is consistent.

We propose an autonomous *instance update language*, based on an adequate specific query language and a pure instance update language. The main features of the query language are a formal type inference system including disjunctive types, and the decidability of the satisfiability problem, despite a negation operator. The pure instance update language allows objects migration, and objects and references creation and deletion; its semantics is declarative, and an algorithm to compute it is presented.

We propose an *evolution mechanism* for combining this instance update language with a classical schema update language, and use it to obtain an evolution language. *Decidability of consistency* is shown for a fragment of this language, by reduction to first-order logic with two variables.

1 Introduction

In object-oriented databases, objects have an existence independent of their value and are grouped into classes that capture their commonalities. The class definitions model the schema and the objects are the instance of the database.

In general, modifying the schema of a database leads to an inconsistent instance. One must then adapt this instance without knowing if the resulting instance will be consistent or not.

The general problem we consider here is to define a mechanism which allows to modify the schema and the instance at the same time in order for the updated instance to fit to the updated schema. We do not consider automatic adaptation as in [BKKK87].

The specific problem we consider is: How to provide the user with an update language allowing static consistency checking? By consistency, we mean that every typing constraint of the updated schema must be valid in the updated instance. Moreover, in correct instances, objects belong to only one class, single-valued attributes have only one value, etc.

For example, "The owner of a car must be a Person". When updating the schema

this typing constraint may change, "The owner of a car must be an Employee". The cars owned by students will then not validate the typing constraints. The programmer must adapt the instance immediately, by deleting these objects or migrating them in a new class. But, if he deletes them without setting the references pointing on them, the database will be inconsistent. Because some objects will continue to refer the deleted objects. If the programmer migrates them, new typing problems may arise in the new class.

We want to be able to tell the user that the adaptation program he proposes will be sure for every instance of the schema. The problems we will then encounter are related to attribute redefinition and object identity. When considering cars, what specific class do cars owners belong to? Do they work as students in a university or as employees in a company? If we migrate them, will other objects continue to refer to them through now ill-typed references? How to get these objects?

The contributions of this paper are: an instance update programming language with static consistency checking, a typable paths query language allowing to denote objects and their types, and a mechanism to combine instance update languages and schema update languages.

Next section presents the problem and the approach we use for instance update through an example. Section 3 describes the data model. Section 4 presents the path expression query language and results of satisfiability for this language. Section 5 gives the formal definitions of the instance update language and uses some results of section 4 for decidability of consistency. Section 6 details the database evolution mechanism allowing to combine schema update languages and instance update languages.
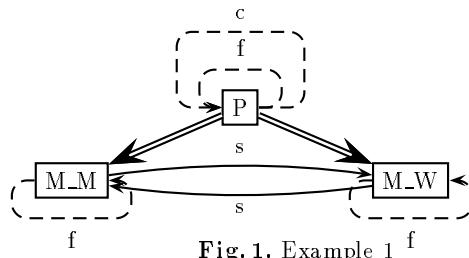

Fig. 1. Example 1

2 An Example

In this section, we only consider instance update to give an intuition of our instance update and path languages. The update programming language we introduce features five primitives. A program over this language is a set of primitive

calls. Intuitively, assuming that x and x' are formal parameters denoting object or sets of objects and c a class name, the syntax and semantics of the primitive calls are the following:

• *move x c* migrates all the objects in x into class c without changing their identity and their references.

• *cut x a x'* delete all attribute links labeled with a between objects in x and objects in x'.

• *set x a x'* establishes references labeled with a between objects in x and objects in x'.

• *new x c* creates a new object into the population of class c.

• *delete x* deletes the objects in x. Let us consider the schema of Figure 1 representing persons (P), married men and women (M_M and M_W) which inherit from P. We want to write an instance update program allowing to marry any two persons belonging to class P, denoted by formal parameters x and x'. To design this kind of update, the user can simply move one object in class M_M, the other one in class M_W and set their spouse (s) attributes. This can be done by writing the following program:

$\{move\ x\ M\_W, move\ x'\ M\_M,$

$set\ x\ s\ x',\ set\ x'\ s\ x\}$

But, as the friends (f) attribute is redefined in classes (M_M) and (M_W), some references will then become ill-typed because the migrated objects will continue to refer to objects specifically in class P through attribute f. This checking is performed statically and tells that the program is not consistent because it can generate ill-typed instances. As a consequence, One can not move the object in x to class M_M without treating its references to objects not in class M_M.

To reach these objects, we use a path query language the main constructs of which are: Given path z the semantics of which is a set of objects, we can get all the objects referenced via attribute a from the objects in z with the path expression $z.a$. Similarly, we can get all objects that refer to objects in z via attribute a with the $z.a^{-1}$ expression. We can also select objects in z that are specifically in class c with the $z : c$ expression. Union, intersection and difference of paths are also provided.

The path expression $(x.f) - (x.f : M\_M)$ denotes exactly the objects causing ill-typed references after migration. So we just have to cut the references from x to these objects and repeat the process for x' to obtain a well-typed instance. If the programmer adds the two following primitive calls, the new program is consistent.

$cut\ x\ f\ (x.f - (x.f : M\_W)),\ cut\ x'\ f\ (x'.f - (x'.f : M\_M))$

This modification is of course not mandatory, the programmer could have migrated the objects into another class. Our aim is just to decide whether a program is consistent or not.

3 Preliminaries—Data Model

We recall briefly the model defined in [AKW90], without methods, and that we extend with sets.

We assume the existence of the following disjoint countable sets of *class names* $\{c_1, c_2, ..\}$ and *attribute names* $\{a_1, a_2, ..\}$ (simply called classes and attributes in the following). A *signature* is an expression $c \to c'$ or $c \to \{c'\}$, where c and c' are classes. An *attribute definition* of a at c is a pair $(a, c \to c')$ or $(a, c \to \{c'\})$ (also denoted $a : c \to c'$, resp. $a : c \to \{c'\}$) where a is an attribute, and $c \to c'$ and $c \to \{c'\}$ are signatures.

A *schema* is a triple $(C, \leq, \Sigma)$ where
1. C is a finite set of classes and $\leq$ is (the transitive closure of) a forest on C (the root of a tree is a maximum for $\leq$); (if $c' \leq c$, c' is a *subclass* of c and c the *superclass* of c'; for each c, we denote $c^* = \{c' \mid c' \leq c\}$);
2. Σ is a set of attribute definitions with classes in C;
3. each attribute has at most one definition at c for each c;
4. if $a : c_1 \to t_1$, $a : c_2 \to t_2$ are in Σ, and $c_1 \leq c_2$, then both $t_1 = c'_1$ and $t_2 = c'_2$, or $t_1 = \{c'_1\}$ and $t_2 = \{c'_2\}$, for some c'_1, c'_2.

In a schema $(C, \leq, \Sigma)$, attribute definitions in Σ are called *explicit*. Moreover, if an attribute a is explicitly defined at c, its definition d is *implicitly* defined in each $c' \leq c$; we say that d is *inherited* in c'. We say that attribute name a is *overloaded* if there is more than one definition of a in Σ. As a consequence of overloading and inheritance, for some given c there may be several definitions of a at c (at most one explicit one). The *resolution* of (*overloading on attribute name*) a at c is the explicit definition of a in the smallest $c' \geq c$ in which a has an explicit definition. If such a c' exists, a is *(well) defined* at c, otherwise a is *undefined* at c.

We assume the existence of a countable set of *object identifiers* $\{o_1, o_2, ...\}$ (simply called objects in the following), disjoint from classes and attributes. Given a schema $(C, \leq, \Sigma)$, a *disjoint object assignment* ν for C is a total function from C to finite sets of objects, such that $c \neq c' \Rightarrow \nu(c) \cap \nu(c') = \emptyset$. For each c, we denote $\nu(c^*) = \bigcup_{c' \leq c} \nu(c')$. We will refer to $\bigcup_{c \in C} \nu(c)$ as the set of objects in ν (denoted ν).

An *instance* of a schema $S = (C, \leq, \Sigma)$ is a pair $I = (\nu, \mu)$ where ν is a disjoint object assignment for C and μ is a total function from the attribute names in S to partial functions such that:
1. if a is undefined at c, then $\mu(a)$ is undefined everywhere in $\nu(c)$; and
2. if the resolution of a at c is $a : c_1 \to c'$ (resp. $a : c_1 \to \{c'\}$) for some $c_1 \geq c$, then $\mu(a) \mid_{\nu(c)}$ is a total function into $\nu(c'^*)$ (resp. into the subsets of $\nu(c'^*)$).

We will need in the following to manipulate pairs (ν, μ) before knowing whether they are instances or not. The following definition is weaker than that of an instance, and independant of any schema. A *pre-instance* is a pair (ν, μ) such that: (1) ν is a partial function from classes into finite sets of objects (defined only for a finite number of classes); and (2) μ is a partial function from attributes into finite binary relations of objects (defined only for a finite number of attributes). For a pre-instance (ν, μ) we will simply write $(o, c) \in \nu$ if $o \in \nu(c)$,

and $o \xrightarrow{a} o' \in \mu$ if $(o, o') \in \mu(a)$ (or $o \in c$ and $o \xrightarrow{a} o'$ when no ambiguity arise); and allow to consider ν and μ as sets, with usual set operations.

Given a pre-instance I and a schema S, if the classes and attributes for which I is defined are in S (regardless of any other constraint), we say that I is a *pre-instance of S*. An instance of S is clearly a pre-instance of S. Given a pre-instance I and a schema S, practically checking whether I is an instance of S is immediate.

4 Objects Designation

4.1 Types

A *type* is an expression $c_1 \vee ... \vee c_n$ or $\{c_1 \vee ... \vee c_n\}$ or $s_1 \vee ... \vee s_n$, where $n \geq 1$, the c_i's, $1 \leq i \leq n$, are distinct classes, and each s_i, $1 \leq i \leq n$, is a type $\{c_{i,1} \vee ... \vee c_{i,n_i}\}$ where the $c_{i,j}$'s, $1 \leq j \leq n_i$, are distinct classes. It is a *type of schema S* if its classes are in S. Given S and an instance I of S, the *semantics* of type $c_1 \vee ... \vee c_n$ of S in I is $[c_1 \vee ... \vee c_n]_I = \bigcup_{1 \leq i \leq n} \nu(c_i)$, resp. $[\{c_1 \vee ... \vee c_n\}]_I = 2^{[c_1 \vee ... \vee c_n]_I}$, and $[s_1 \vee ... \vee s_n]_I = \bigcup_{1 \leq i \leq n} 2^{[s_i]_I}$ (where 2^E denotes the subsets of any set E).

4.2 Paths—Syntax and Typing

We assume the existence of a countable set of *typed variables* $\{x, y, ...\}$ disjoint from classes, attributes and objects; each variable x has an associated type, and for each type there are countably many variables of this type. To simplify the presentation, only types of the form c or $\{c\}$, for some class c, are considered here; corresponding variables may be explicitly written x^c (resp. $x^{\{c\}}$).

Definition 4.1 A *path* is one of the following recursively defined expressions, where c is a class, x a variable, a an attribute, and l and l' paths (and $.,^{-1},:$ $,\cap,\cup,-$ are auxiliary symbols): x, $l.a$, $l.a^{-1}$, $l : c$, $l \cap l'$, $l \cup l'$, $l - l'$. $\square$

Given a schema $S = (C, \leq, \Sigma)$, we define the following *typing rules* (Figure 2), where the axioms are rules 1. (For $E \subseteq C$ we denote $a(E) = \{c'^* \mid c \in E$ and the resolution of a at c is $a : c'' \to c'$ or $a : c'' \to \{c'\}\}$, and simply write $a(c)$ if $E = \{c\}$; and $a^{-1}(E) = \{c \mid a(c) \cap E \neq \emptyset\}$. Attribute definition $a : c \to c'$ is said *single-valued* and $a : c \to \{c'\}$ *set-valued*.)

The design of a type system is traditionally done as follows (see e.g. [Mit90]). First, the set of queries that are considered by the designer as intuitively "correct" is chosen, together with its complement: the set of queries that are considered intuitively not interesting. Second, the type system (the set of rules) is designed so as to capture exactly this set of "correct" query expressions, and reject the others. Finally, the result of a query is expected to be an element of the type of the query. This is the way we proceed here.

The design of the set of "correct" queries was done here based on our programming experience with a restricted maquette, and complementary examples.

1. $\dfrac{}{x^c \; : \; c'_1 \vee \ldots \vee c'_m}$ (resp. $\dfrac{}{x^{\{c\}} \; : \; \{c'_1 \vee \ldots \vee c'_m\}}$) if the c'_j's, $1 \leq j \leq m$, are the subclasses of c

2. $\dfrac{l \; : \; c_1 \vee \ldots \vee c_n}{l.a \; : \; c'_1 \vee \ldots \vee c'_m}$ if for each c_i, $1 \leq i \leq n$, a at c_i is (defined and) single-valued; and where $a(\{c_1, \ldots, c_n\}) = \{c'_1, \ldots, c'_m\}$

3. $\dfrac{l \; : \; c_1 \vee \ldots \vee c_n}{l.a \; : \; \{c'_{1,1} \vee \ldots \vee c'_{1,m_1}\} \vee \ldots \vee \{c'_{n,1} \vee \ldots \vee c'_{n,m_n}\}}$ if for each c_i, $1 \leq i \leq n$, a at c_i is set-valued; and $a(c_i) = \{c'_{i,1}, \ldots, c'_{i,m_i}\}$

4. $\dfrac{l \; : \; \{c_{1,1} \vee \ldots \vee c_{1,k_1}\} \vee \ldots \vee \{c_{n,1} \vee \ldots \vee c_{n,k_n}\}}{l.a \; : \; \{c'_{1,1} \vee \ldots \vee c'_{1,m_1}\} \vee \ldots \vee \{c'_{n,1} \vee \ldots \vee c'_{n,m_n}\}}$ if for each $c_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq k_i$, a is defined at $c_{i,j}$; and where for each i, $1 \leq i \leq n$, we have $a(\{c_{i,1}, \ldots, c_{i,k_i}\}) = \{c'_{i,1}, \ldots, c'_{i,m_i}\}$

5. $\dfrac{l \; : \; c_1 \vee \ldots \vee c_n}{l.a^{-1} \; : \; \{c'_{1,1} \vee \ldots \vee c'_{1,m_1}\} \vee \ldots \vee \{c'_{n,1} \vee \ldots \vee c'_{n,m_n}\}}$ if for each c_i, $1 \leq i \leq n$, $a^{-1}(c_i) = \{c'_{i,1}, \ldots, c'_{i,m_i}\} \neq \emptyset$

6. $\dfrac{l \; : \; \{c_{1,1} \vee \ldots \vee c_{1,k_1}\} \vee \ldots \vee \{c_{n,1} \vee \ldots \vee c_{n,k_n}\}}{l.a^{-1} \; : \; \{c'_{1,1} \vee \ldots \vee c'_{1,m_1}\} \vee \ldots \vee \{c'_{n,1} \vee \ldots \vee c'_{n,m_n}\}}$ if for each $c_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq k_i$, there exists c' such that $c_{i,j} \in a(c')$; and where for each i, $1 \leq i \leq n$, we have $a^{-1}(\{c_{i,1}, \ldots, c_{i,k_i}\}) = \{c'_{i,1}, \ldots, c'_{i,m_i}\}$

7. $\dfrac{l \; : \; c_1 \vee \ldots \vee c_n}{(l:c) \; : \; c_{i_1} \vee \ldots \vee c_{i_k}}$ (resp. $\dfrac{l \; : \; \{c_1 \vee \ldots \vee c_n\}}{(l:c) \; : \; \{c_{i_1} \vee \ldots \vee c_{i_k}\}}$) if $c^* \cap \{c_1, \ldots, c_n\} = \{c_{i_1}, \ldots, c_{i_k}\} \neq \emptyset$

In the following rules, each t_i, t'_j, t''_i or $t''_{i,j}$ is of the form $\{c_1 \vee \ldots \vee c_n\}$. (For the sake of presentation, such a type $t = \{c_1 \vee \ldots \vee c_n\}$ will be denoted below as the set $\{c_1, \ldots, c_n\}$, and we will allow to simply write $t \cap t'$ and $t \cup t'$ for such types.)

8. $\dfrac{l \; : \; t_1 \vee \ldots \vee t_n}{(l:c) \; : \; t''_1 \vee \ldots \vee t''_n}$
   if there exists i, $1 \leq i \leq n$, such that $c^* \cap t_i \neq \emptyset$; and where for each i, $1 \leq i \leq n$, $t''_i = c^* \cap t_i$ if $c^* \cap t_i \neq \emptyset$, and t''_i doesn't appear otherwise

9. $\dfrac{l \; : \; t_1 \vee \ldots \vee t_n \qquad l' \; : \; t'_1 \vee \ldots \vee t'_{n'}}{l \cap l' \; : \; t''_{1,1} \vee \ldots \vee t''_{n,n'}}$ if there exists i,j, $1 \leq i \leq n$, $1 \leq j \leq n'$, such that $t_i \cap t'_j \neq \emptyset$; and where for each i,j, $1 \leq i \leq n, 1 \leq j \leq n'$, we have $t''_{i,j} = t_i \cap t'_j$ if $t_i \cap t'_j \neq \emptyset$, and $t''_{i,j}$ doesn't appear otherwise

10. $\dfrac{l : t_1 \vee \ldots \vee t_n \qquad l' \; : \; t'_1 \vee \ldots \vee t'_{n'}}{l \cup l' \; : \; t''_{1,1} \vee \ldots \vee t''_{n,n'}}$ where for each i,j, $1 \leq i \leq n, 1 \leq j \leq n'$, $t''_{i,j} = t_i \cup t'_j$

11. $\dfrac{l \; : \; t_1 \vee \ldots \vee t_n \qquad l' \; : \; t'_1 \vee \ldots \vee t'_{n'}}{l - l' \; : \; t_1 \vee \ldots \vee t_n}$ if there exists i,j, $1 \leq i \leq n$, $1 \leq j \leq n'$, such that $t_i \cap t'_j \neq \emptyset$

Fig. 2. Typing Rules

It is thus motivated but arbitrary: If a different set of "correct" queries were to be considered, it is easy to modify the rules, while keeping the same properties. For instance, the idea of Rule 4 is to flatten sets roughly, provided attribute a is defined on each $c_{i,j}$, but possibly mono-valued for one $c_{i,j}$ and set-valued for another. It is easy to modify this rule, for instance by making the *if* clause more restrictive as follows: For each i, $1 \leq i \leq n$, attribute a should be either mono-valued for all $c_{i,j}$'s, $1 \leq j \leq k_i$, either set-valued for all. In a second variant of this rule, it should be for instance either mono-valued for all $c_{i,j}$'s, $1 \leq i \leq n$, $1 \leq j \leq k_i$, either set-valued for all.

We say that l *is typable (in S) of type* t (denoted $l : t$), if $l : t$ is derivable using the rules defined above. (That is, there exists a (finite) derivation tree, the

root of which is $l : t$, the leaves axioms, and the internal nodes related by rules.)

Proposition 4.2 Given a schema S and a path l, it is decidable whether l is typable in S. If so, there is a unique type t of S such that $l : t$, which is inferred by the natural algorithm. □

For example, in Example 1 $z.c.f.c^{-1} : M\_W$ which denotes the married mothers of the friends of the children of z has type: set of M_W, where z is in P or M_M or M_W. If the variable z is of type P, then the path: $z.c.f.c^{-1}$ has type heterogenous set of objects in classes P, M_M and M_W.

4.3   Paths—Semantics

Given a finite set X of variables, a *valuation* v over X is a mapping from X into objects and sets of objects such that for each x^c or $x^{\{c\}}$ in X, $v(x^c)$ is an object and $v(x^{\{c\}})$ a set of objects. Given a schema S and a pre-instance $I = (\nu, \mu)$ of S, a valuation v over X is said to be *in* I, if for each class c in S and variable x^c (resp. $x^{\{c\}}$) in X, we have $v(x^c) \in \nu(c^*)$, (resp. $v(x^{\{c\}}) \subseteq \nu(c^*)$). (The general case, i.e., variables of any type, follows naturally from the semantics of types.) (For function f with domain A and range B, for $E \subseteq A$ and $F \subseteq B$ we denote $f(E) = \{f(o) \mid o \in E\}$ if $f|_E$ is total, and $f^{-1}(F) = \{o \in A \mid f(o) \in F\}$ if f is single-valued, resp. $f(o) \subseteq F$ if f is set-valued, and simply write $f^{-1}(o')$ if $F = \{o'\}$.)

Definition 4.3 Given a schema S, an instance $I = (\nu, \mu)$ of S, a typable path l of S and a valuation v over the variables of l in I, the *semantics* of l in I under v (denoted $[\,l\,]_{I,v}$) is recursively defined as follows.

1. $[\,x\,]_{I,v} = v(x)$
2. $[\,l.a\,]_{I,v} = \mu(a)([\,l\,]_{I,v})$
3. $[\,l.a^{-1}\,]_{I,v} = \mu(a)^{-1}([\,l\,]_{I,v})$
4. $[\,l : c\,]_{I,v} = [\,l\,]_{I,v} \cap \nu(c^*)$ (resp. $\{[\,l\,]_{I,v}\} \cap \nu(c^*)$ if $l : c_1 \vee \ldots \vee c_n$ for some $c_1, \ldots, c_n$)
5. $[\,l \cap l'\,]_{I,v} = [\,l\,]_{I,v} \cap [\,l'\,]_{I,v}$
   $[\,l \cup l'\,]_{I,v} = [\,l\,]_{I,v} \cup [\,l'\,]_{I,v}$
   $[\,l - l'\,]_{I,v} = [\,l\,]_{I,v} - [\,l'\,]_{I,v}$ □

Proposition 4.4 Given a schema S, a path l and a type t, if $l : t$ in S, then for each instance I of S and valuation v over the variable of l in I, we have $[\,l\,]_{I,v} \subseteq [\,t\,]_I$. □

Theorem 4.5 Satisfiability of paths queries is decidable (if the construct $l.a$ with a mono-valued is not considered). □

Proof (Sketch) In other words, given a path l in schema S, we can tell whether there exists an instance and a valuation such that the evaluation of l yields a non-empty result. We consider here the fragment of first-order logic called first-order

logic with two variables, and denoted FO2 [Mor75]. The syntax is the usual one of first-order logic without function symbols, except that only two variable symbols are allowed (instead of countably many), here z and z'. We reduce satisfiability of paths queries to satisfiability of FO2. Given a schema $S = (C, \leq, \Sigma)$ and a path l typable in S, an FO2 formula φ_l is associated to l such that φ_l is satisfiable iff there exists an instance I and a valuation v such that $[l]_{I,v} \neq \emptyset$. As satisfiability of an FO2 formula is decidable, the satisfiability of our paths language is decidable. The first-order language is the same as that used to build formulas to design the semantics of the pure instance update language in Section 5.1. Intuitively, to each path l, a formula with one free variable $\varphi_l(z)$ is associated as follows: $\varphi_{l.a}(z) = \exists z'(a(z', z) \wedge \varphi_l(z'))$; $\varphi_{l.a^{-1}}(z) = \exists z'(a(z, z') \wedge \varphi_l(z'))$; $\varphi_{l:c}(z) = \varphi_l(z) \wedge (\bigvee_{c' \leq c} c'(z))$; $\varphi_{l \cup l'}(z) = \varphi_l(z) \vee \varphi_{l'}(z)$; $\varphi_{l \cap l'}(z) = \varphi_l(z) \wedge \varphi_{l'}(z)$; $\varphi_{l-l'}(z) = \varphi_l(z) \wedge \neg \varphi_{l'}(z)$. Examples: $\varphi_{x.a.b} = \exists z \exists z'(b(z', z) \wedge \exists z(a(z, z') \wedge x(z)))$; $\varphi_{x.a-y.b^{-1}} = \exists z((\exists z'(a(z', z) \wedge x(z'))) \wedge (\exists z'(b(z, z') \wedge y(z'))))$. To see that l is satisfiable iff φ_l is satisfiable, we need to define the semantics of our first-order language. This semantics is the set of structures defined in Section 5.1 for the satisfiability of $\psi_{S,P}$. $\square$

5 Instance Update

5.1 Pure Instance Update

We assume the existence of five *instance update primitive* symbols: *new*, *delete*, *move*, *set* and *cut*, distinct from classes, attributes, objects and variables. An *instance update instruction* is one of the following expressions (strictly speaking a pair, triple or 4-tuple), where c is a class, a an attribute, and x and x' variables.

1. *new x c* — called *object creation* (in class c)
2. *delete x* — *object(s) deletion* (of object(s) denoted by variable x)
3. *move x c* — *object(s) migration* (of x to c)
4. *set x a x'* — *reference(s) creation* (from x to x' for attribute a)
5. *cut x a x'* — *reference(s) deletion* (from x to x' for a)

Definition 5.1 A *pure instance update program* is a finite set of instance update instructions. $\square$

A program is *well-formed* if for each instruction *new x c* the type of x is c, and no variable in a *new* instruction also occurs in a *delete*, *move* or *cut* instruction. Its *formal parameters* are its variables but the ones in a *new* instruction.

A program is *legal for designation* upon a schema S if the classes of its formal parameters are in S; and for each *cut x a x'*, if the type of x is c or $\{c\}$ and that of x' is c' or $\{c'\}$, then a is defined at c and $c' \in a(c)$. It is *legal for specification* upon S if for each *new x c* or *move x c*, c is in S; and for *set x a x'*, a is in S. It is simply said *legal* when it is legal for both. We will consider in the following only well-formed programs, legal upon the schemas considered.

Given a schema $S = (C, \leq, \Sigma)$ and a program P upon S, we consider the following language. The usual variables ($\{z, z', ...\}$), connectives and auxiliary

symbols of first-order logic; and the predicate symbols are the classes in S and the variables in P (unary), and the attributes in S (binary). We consider the usual semantics of first-order logic [Bar77], and denote $\mathcal{M} \models \psi$ when structure $\mathcal{M}$ for this language satisfies formula ψ over this language.

A formula (over this language) is associated to each instruction in P (regardless of the others) as follows.

1. *new x_1 c* or *move x_1 c* : $\quad \forall z \ (x_1(z) \rightarrow (c(z) \wedge \bigwedge_{c' \in C - \{c\}} \neg c'(z)))$

2. *delete x_1* : $\qquad\qquad\quad \forall z \ (x_1(z) \rightarrow (\bigwedge_{c \in C} \neg c(z)))$

3. *set x_1 a x_2* : $\quad \forall z \ (x_1(z) \rightarrow [(\bigvee_{c \in C} c(z)) \wedge \forall z' \ (x_2(z') \leftrightarrow [(\bigvee_{c \in C} c(z')) \wedge a(z, z')])] \ )$

4. *cut x_1 a x_2* : $\qquad\qquad \forall z \ (x_1(z) \rightarrow \forall z' \ (x_2(z') \rightarrow \neg a(z, z')))$

The (first-order) *formula associated to P (upon S)* (denoted $\psi_{S,P}$, or ψ when no ambiguity arises) is the conjunction of all these formulas. We will consider in the following only such formulas (and not general formulas of this first-order language).

Given a schema S, a program P upon S, and a pre-instance $I = (\nu, \mu)$ of S, let X be the set of variables in P and $Y \subseteq X$ the variables occuring in an instruction *new*. We will consider in the following only valuations v over X that are both: *in I except for Y*, that is, v is in I for $X - Y$, and $y \in Y \Rightarrow v(y) \cap \nu = \emptyset$; and *disjoint for Y*, that is, $x, y \in Y, y \neq x \Rightarrow v(y) \cap v(x) = \emptyset$.

Given S, P upon S, $I = (\nu, \mu)$ pre-instance of S and v over (the variables of) P in I, we simply write (I, v) for the following structure for the language considered above. The domain is the union of the objects in v and those in ν; the predicate symbols are mapped to relations over the domain as follows: for each variable x in P, $x \mapsto v(x)$; for each class c in S, $c \mapsto \nu(c)$; and for each attribute a in S, $a \mapsto \mu(a)$ (when a is set-valued, there is a tuple for each $o \overset{a}{\mapsto} o'$). We will consider in the following only such structures.

Given S, P upon S, and v over P, the formula $\psi_{S,P}$ is *satisfiable w.r.t. v* if there exists a pre-instance I of S such that v is in I and $(I, v) \models \psi_{S,P}$; otherwise it is *unsatisfiable w.r.t. v*.

Proposition 5.2 Let be given S, P upon S, v over P, and ψ associated to P upon S. ψ is unsatisfiable w.r.t. v if, and only if, there are variables x and y of P such that $v(x) \cap v(y) \neq \emptyset$ (strictly speaking $v(x) = v(y)$ if they are not sets), and one of the following holds.

1. *delete x*, *move y c* are instructions in P
2. *delete x*, *set y a x'*
3. *delete x*, *set x' a y*
4. *move x c*, *move y c'* (with $c \neq c'$)
5. *set x a x'*, *set y a y'* are in P, and $v(x') \neq v(y')$
6. *set x a x'*, *cut y a y'* are in P, and $v(x') \cap v(y') \neq \emptyset$ □

Given S and P upon S, the pairs of variables (x, y) in cases 1, 2, 3, 4 above are said *conflictual*. Valuation v over P is *conflictual* if it is such that: for some pair (x, y) in cases 1, 2, 3 or 4 above, $v(x) \cap v(y) \neq \emptyset$; in case 5, both $v(x) \cap v(y) \neq \emptyset$ and $v(x') \neq v(y')$; and in case 6, both $v(x) \cap v(y) \neq \emptyset$ and $v(x') \cap v(y') \neq \emptyset$. In other words, Proposition 5.2 says that ψ is unsatisfiable w.r.t. v iff v is conflictual. If there exists such a v, P is said *contradictory*.

Proposition 5.3 Given S, it is decidable whether P is non contradictory. $\square$

Proof (Sketch) First, use Proposition 5.2 to look whether there are conflictual variables in P. If not, then no v is conflictual, and by Proposition 5.2 ψ is satisfiable w.r.t. v. If there are conflictual variables (x, y), check whether there exists a valuation v conflictual for (x, y), that is, $v(x) \cap v(y) \neq \emptyset$ (resp. $v(x') \neq v(y')$). But the answer is immediate, and positive; thus P is contradictory. $\square$

Given S, we consider the following partial ordering (denoted $\subseteq$) over the pre-instances of S. Let $I = (\nu, \mu)$ and $I' = (\nu', \mu')$; $I \subseteq I'$ if $\nu \subseteq \nu'$ and $\mu \subseteq \mu'$ (see Section 3 for notations).

Theorem 5.4 Let be given S, P upon S, I instance of S, v over P in I, and ψ associated to P upon S.

1. If ψ is satisfiable w.r.t. v, then the set $\{I' \mid I' \subseteq I \text{ and } \exists I'' \supseteq I' \text{ s. t. } (I'', v) \models \psi\}$ is not empty and has a unique maximum, denoted $invariant(I)$.
2. In this case, the set $\{I' \mid invariant(I) \subseteq I' \text{ and } (I', v) \models \psi\}$ is not empty and has a unique minimum, denoted $decl_{S, P, v}(I)$. $\square$

Proof (Sketch) Roughly speaking, the proof is constructive and corresponds to the following algorithm. Let $P = \{\alpha_1, ..., \alpha_m\}$. To each instruction α_i in P, $1 \leq i \leq m$, we associate a 4-tuple $(\nu_i^+, \nu_i^-, \mu_i^+, \mu_i^-)$ as follows.
1. To *new x c* we associate: $(\{(c, o)\}, \emptyset, \emptyset, \emptyset)$, for some o not in ν, and different from each other object created by such an instruction;
2. *delete x* : $(\emptyset, \{(c, o) | c \in S, o \in v(x)\}, \emptyset, \emptyset)$;
3. *move $x^{c'}$ c* : $(\{(c, o) | o \in v(x)\}, \{(c', o) | o \in v(x)\}, \emptyset, \emptyset)$;
4. *set x a x'* : $(\emptyset, \emptyset, \{(o \xrightarrow{a} o') | o \in v(x), o' \in v(x')\}, \emptyset)$;
5. *cut x a x'* : $(\emptyset, \emptyset, \emptyset, \{(o \xrightarrow{a} o') | o \in v(x), o' \in v(x')\})$.

Then we check whether this set of operations involves some contradiction, namely if there exist $1 \leq i \neq j \leq m$, such that one of the following holds.
1. $\{o \mid (o, c') \in \nu_i^-\} \cap \{o \mid (o, c'') \in \nu_j^+\} \neq \emptyset$;
2. $\{(o, a) \mid (o \xrightarrow{a} o') \in \mu_i^+\} \cap \{(o, a) \mid (o \xrightarrow{a} o'') \in \mu_j^+\} \neq \emptyset$;
3. $\mu_i^- \cap \mu_j^+ \neq \emptyset$.

If no such contradiction arises, we define:
$[P]_{I, v}^S = ((\nu - \bigcup_{1 \leq i \leq m} \nu_i^-) \cup (\bigcup_{1 \leq i \leq m} \nu_i^+), (\mu - \bigcup_{1 \leq i \leq m} \mu_i^-) \cup (\bigcup_{1 \leq i \leq m} \mu_i^+))$. $\square$

Definition 5.5 Let be given S, P upon S non contradictory, I instance of S, and v over P in I non conflictual. The *semantics* of P upon S on I under v is $[P]_{I, v}^S = decl_{S, P, v}(I)$. $\square$

Theorem 5.6 Given S and P, it is decidable whether for each I and v, $[P]_{I, v}^S$ is an instance. $\square$

5.2 Concrete Instance Update Programs

A *concrete instance update program* is a triple (L, α, P) where L is a finite set of paths, P a pure instance update program, and α a mapping from the formal parameters of P onto L; moreover, the set of variables of L and P are disjoint.

It is *legal* upon a schema S, if the paths in L are typable, P is legal upon S, and the type of each formal parameter of P is equal to that of its image by α.[1] Only legal programs will be considered in the following.

A valuation for such a program (L, α, P) is over the set of variables of L union the set Y of the variables occuring in a *new* instruction in P. Given an instance I, we consider only valuations that are in I except for Y. To each such valuation v is associated a valuation $\tilde{v}$ over P, as follows. Over Y, $\tilde{v} = v$; and for each formal parameter x of P, let $\alpha(x)$ be the path in L associated to x; then $\tilde{v}(x) = [\alpha(x)]_{I,v}$. Note that $\tilde{v}$ is itself in I except for Y.

Proposition 5.7 Given S, it is decidable whether Q is non contradictory, that is, if ψ is satisfable w.r.t. $\tilde{v}$. □

Proof (Sketch) We generalize the proof of Proposition 5.3. Given (x, y) conflictual, we have to decide whether $\tilde{v}$ is conflictual, that is, whether $\tilde{v}(x) \cap \tilde{v}(y) \neq \emptyset$ and $\tilde{v}(x) \neq \tilde{v}(y)$ (considering the latter, $l \neq l'$ iff $l - l' \neq \emptyset$ or $l' - l \neq \emptyset$). But $\tilde{v}(x) = [\alpha(x)]_{I,v}$. That is, decide whether $\alpha(x) \cap \alpha(y)$ and $(\alpha(x) - \alpha(y)) \cup (\alpha(x) - \alpha(y))$ are satisfiable, which is the case by Theorem 4.5. □

Given S, $Q = (L, \alpha, P)$, I and v, the *semantics* of Q upon S on I under v is $[Q]^S_{I,v} = [P]^S_{I,\tilde{v}}$.

Definition 5.8 A concrete instance update program Q upon a schema S is *consistent* if for each instance I and valuation v, $[Q]^S_{I,v}$ is an instance. □

Theorem 5.9 Given S and Q, it is decidable whether Q upon S is consistent. □

Proof (Sketch) As for Proposition 5.6, we only show here how to decide whether $[Q]^S_{I,v}$ has no ill-typed references. We generalize the case analysis of the proof for pure instance update programs. We treat the *move* instruction. Given $Q = (L, \alpha, P)$, let $x_1, ..., x_n$ be the parameters of P. Assume there is an instruction *move* x_1 c'_1, with $x_1 : c_1$, $a : c_1 \rightarrow c_2$, and $a : c'_1 \rightarrow c'_2$, with $c_2 \not\leq c'_2$. Then $[Q]_{I,v}$ has ill-typed references except if (that is, is well typed iff) there is an instruction (1) *move* x_i c'_2, or (2) *cut* x_j a x_i, with $[\alpha(x_1).a]_{I,v} \subseteq [\alpha(x_i)]_{I,v}$ and $[\alpha(x_1)]_{I,v} \subseteq [\alpha(x_j)]_{I,v}$ for each I and v. But this is true iff $[\alpha(x_1).a - \alpha(x_i)]_{I,v} = \emptyset$ and $[\alpha(x_1) - \alpha(x_j)]_{I,v} = \emptyset$ for each I, v; iff $\alpha(x_1).a - \alpha(x_i)$ and $\alpha(x_1) - \alpha(x_j)$ are unsatisfiable, which, as satisfiability, is decidable. □

---

[1] From a practical point of view, the user only declares the types of the paths variables; the system infers the types of the paths (general types as presented in Section 4.1) and declares the formal parameters of the pure instance update program from these types.

5.3 Pure *versus* Concrete Instance Updates

Providing the paths language makes more programs consistent:

Proposition 5.10 Given S and $Q = (L, \alpha, P)$, if the pure instance update program P is consistent, then Q is consistent. □

Proof Immediate: $[P]_{I,v}$ is an instance for each v, and $\tilde{v}$ is simply a particular valuation. □

The intuition here is simply the following. In the pure instance update language, P has to be checked against the set $\mathcal{V}(I)$ of all valuations. In the concrete instance update language, L specifies a subset of $\mathcal{V}(I)$; P needs only to be checked against this subset.

The programs made consistent by the introduction of the paths language are useful:

Theorem 5.11 The concrete instance update language is more expressive than the pure instance update language. □

Proof We show that given S, there exists a function f from the instances of S to the instances of S, such that (1) no pure instance update program expresses f, and (2) f is expressed by a concrete instance update program. Let $S = (\{c_1, c_2, c_3, c_4\}, \emptyset, \{a : c_1 \rightarrow c_2, a : c_3 \rightarrow c_4\})$, let A be any set of objects, and let f be defined as follows. For each $I = (\nu, \mu)$, if $A \subseteq \nu$, then $f(I) = (\nu', \mu')$, where $\nu' = \nu$, except that $A \subseteq \nu(c_3)$ and $A.a \subseteq \nu(c_4)$; and $\mu' = \mu$. Assume P implements f; then P has to have two *move* instructions, namely *move x c₃* and *move x' c₄* (one is not enough, because both c_3 and c_4 have to be augmented). But then, there exists a non-empty valuation v such that $v(x) \cap v(x') \neq \emptyset$ (e.g. $v(x) = v(x') = \{o\}$), which is conflictual. The following program implements f. $L = \{y, y.a\}$; $\alpha : x \mapsto y, x' \mapsto y.a$; $P = \{move\ x\ c_3, move\ x'\ c_4\}$. □
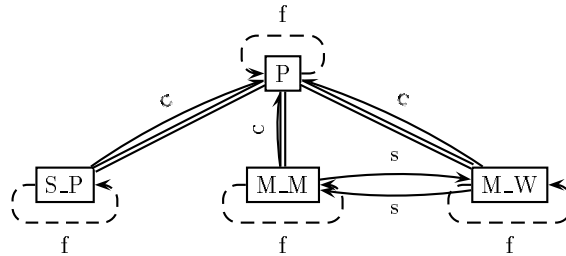

Fig. 3. Example 2

6 Database Evolution

In this section, we present the mechanism allowing to perform a well-typed database evolution by simultaneously updating the schema and the instance. The purpose of this mechanism is to be independant from the schema update language and the instance update language and to allow static typechecking. We will use the language presented in section 5 to illustrate the database evolution process.

Definition 6.1 An *evolution program* is a pair (R, Q) where R is a schema update program and Q a concrete instance update program. $\square$

Given schemas $S = (C, \leq, \Sigma)$ and $S' = (C', \leq', \Sigma')$, we denote $S \oplus S'$ the schema $(C \uplus C', \leq \uplus \leq', \Sigma \uplus \Sigma')$, where $\uplus$ is the usual disjoint union over sets. If c (resp. a) appears in both S and S', the c in S' is denoted $\tilde{c}$ (resp. $\tilde{a}$).

Let be given a schema S and an evolution program $E = (R, Q)$ such that: R is non contradictory, and legal and consistent upon S; and $Q = (L, \alpha, P)$ is legal for designation upon S, and legal for specification upon $[R]_S$. The *concrete instance update program associated to* (R, Q) *and* S, $\overline{Q} = (L, \alpha, \overline{P})$, is upon $S \oplus [R]_S$, and is defined as follows.

1. Each *delete* or *cut* instruction in P is in $\overline{P}$;
2. for each *new x c* (resp. *move x c*, *set x a x'*) in P, we have *new x $\tilde{c}$* (resp. *move x $\tilde{c}$*, *set x $\tilde{a}$ x'*) in $\overline{P}$;
3. for each class c in S such that $-c \notin R$, there is an instruction *move x_c $\tilde{c}$* in $\overline{P}$, where x_c is a variable (of type $\{c\}$) appearing only in this instruction.

Note that $\overline{Q}$ is legal. E is said *legal* upon S if $\overline{Q}$ is non contradictory upon $S \oplus [R]_S$.

For each class c in S, let $var(c)$ denote the set of variables appearing in a *move* or *delete* instruction in Q, and in the type of which c appears. Let I be an instance of S, and v a valuation over the variables of Q in I. We define the valuation $\overline{v}$ as extending v to each variable x_c introduced in item 3 above, as: $\overline{v}(x_c) = \nu(c) - \bigcup_{x \in var(c)} v(x)$; $\overline{v}$ is said *empty for c* if $\overline{v}(x_c) = \emptyset$.

Definition 6.2 Let be given S, $E = (R, Q)$ legal upon S, I instance of S and v over Q in I non conflictual; let $\overline{Q}$ be associated to E and S, and $\overline{v}$ extending v. Let v be such that $\overline{v}$ is empty for each deleted class. The *semantics* of E upon S on I under v is the pair (S', I'), where $S' = [R]_S$, and $I' = [\overline{Q}]_{I, \overline{v}}^{S \oplus S'}$. $\square$

Given S, an evolution program is *consistent*, if for each I and v, its semantics is an instance.

Proposition 6.3 Given S and R, it is decidable whether $\overline{Q}$ is contradictory. $\square$

Proof (Sketch) It is an extension of the proof of Proposition 5.7. $\square$

Theorem 6.4 Given S, it is decidable whether E is consistent (for path queries without the *l.a* construct for a mono-valued). $\square$

Proof (Sketch) It is a careful extension of the proof of Theorem 5.9 taking into account the changes in the schema. $\square$

To illustrate our topic, we consider the schema of Example 3 derived from the one in Example 1. This modification of the schema from example 1 consists in creating a class S_P which represents the single parents, which inherits from P and which has an attribute c. The attribute c has to be deleted in class P. The following schema update program is designed to perform this modification of the schema:

$+_c S\_P, +_a c : S\_P \rightarrow P, +_a c : M\_W \rightarrow P, +_a c : M\_M \rightarrow P, +_a f : S\_P \rightarrow S\_P, +_h S\_P P, -_a c P \rightarrow P$

It is obvious that if the instance stay unchanged, it will be non-consistent w.r.t. the schema. The reason is that some objects in class P will have an attribute c which is now undefined in this class.

It is clear that the schema update and the instance update can not be performed sequentially. Because the class S_P does not exists to allow the instance update program to run first. And because running the schema update program first will lead to a non-consistent instance. We could cut the instance update program and the schema update program to interlace their instructions, but the system would then not be able to detect errors that would lead to a non consistent database. To maintain consistency, we can cut the links for attribute c of objects in class P but it seems better to migrate the objects from class P, which have an attribute c different from the empty set, into class S_P. Furthermore, we will have to cut all the attribute links f from these objects to objects which are not single parents.

The only necessary parameter for the instance update program that will perform this update is the whole population of the class P and its subclasses M_M and M_W. This parameter will be denoted as z.

Using the path query language, we can define the objects that are single parents. These objects are defined by the path expression: $z.c^{-1} - (z.c^{-1} : M\_W \cup z.c^{-1} : M\_M)$

In the following this expression will be denoted as the formal parameter z_1. The instance update program performing the desired adaptation is then:

$\{move\ z_1\ S\_P,\ cut\ z_1\ f\ (z - z_1)\}$

7 Conclusion

In this paper, we first presented a declarative instance update language using typed parameters. The way of designating these parameters is based on attribute paths existing in the schema. Some typing results have been exposed for this instance update language. Then, we have defined a database evolution mechanism. This mechanism relies on the combination of two languages. The first one is a basic schema update language featuring the addition and removal of classes, inheritance links and attribute definition. The syntax of this language is close from the one in [BKKK87]. The second language is the instance update language presented in section 5. This work is different from the ones defined in [FMZ$^+$95],

[BKKK87] in the fact that the mechanism is independant from the schema and instance update languages and that there is no automatic reorganization of the instance consecutive to the schema update. The task of the system is only to tell whether the resulting database will be a consistent database (consistent schema and instance of this schema) or not.

References

[AKW90] S. Abiteboul, P. Kanellakis, and E. Waller. Method schemas. In *PODS*, pages 16–27, 1990.

[ALP91] J. Andany, M. Leonard, and C. Palisser. Management of schema evolution in databases. In *VLDB*, 1991.

[Bar77] J. Barwise. Chapter: An introduction to first-order logic. In *Handbook of Mathematical Logic*, 1977.

[BKKK87] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD*, 1987.

[FMZ+95] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the o2 object database system. In *VLDB*, pages 170–181, 1995.

[Lag] J.B. Lagorce. Aspects of updates in databases. PhD thesis in preparation.

[LSW] J.B. Lagorce, A. Stočkus, and E. Waller. Object-oriented databases evolution. Technical Report in preparation, Orsay.

[Mit90] J. C. Mitchell. Chapter: Type systems for programming languages. In *Handbook of Theoretical Computer Science*, 1990.

[MMW94] A. Mendelzon, T. Milo, and E. Waller. Object migration. In *PODS*, 1994.

[Mor75] M. Mortimer. On languages with two variables. In *Zeitschr. f. mat. Logik und Grunlagen d. Math, Bd. 21*, pages 135–140, 1975.

[NR89] G.T. NGuyen and D. Rieu. Schema evolution in object-oriented database systems. In *Data and Knowledge Engineering*, 1989.

[PS87] D. J. Penney and J. Stein. Class modification in the gemstone object-oriented dbms. In *OOPSLA*, 1987.

[Sto95] A. Stockus. Migration dans les bases de données orientées objet. DEA report, Orsay, 1995.

[Su91] J. Su. Dynamic constraints and object migration. In *VLDB*, pages 233–242, 1991.

[Wal91] E. Waller. Schema updates and consistency. In *DOOD*, 1991.

[Zdo87] S. Zdonik. Can objects change types, can types change objects? In *Workshop on OODBS*, 1987.

[Zic92] R. Zicari. A framework for schema updates in an object-oriented database system. In *The O_2 Book*, 1992.