An Extension of Path Expressions to Simplify Navigation in Object-Oriented Queries

Jan Van den Bussche[*] University of Antwerp

Gottfried Vossen[†] University of Muenster

Abstract

Path expressions, a central ingredient of query languages for object-oriented databases, are currently used as a purely navigational vehicle. We argue that this does not fully exploit their potential expressive power as a tool to specify connections between objects. In particular, a user should not be required to specify a path to be followed in full, but rather should provide enough information so that the system can infer missing details automatically. We present and study an extended mechanism for path expressions which resembles the omission of joins in universal relation interfaces. The semantics of our mechanism is given in the general framework of a calculus-like query language. Techniques from semantic query optimization are employed to obtain efficient specifications. We also consider the possibility that links can be traversed backwards, which subsumes previous proposals to specify inverse relationships at the schema level and also fully exploits the meaning of inheritance links.

1 Introduction

Query languages for object-oriented databases (OODBs) have become an active field of research in recent years [3, 12], with a central topic being the design of languages that are at the same time easy to use and powerful. Since it is common in OODBs that objects directly reference each other, query languages for such databases require some form of navigation, which is usually provided by means of *path expressions (PEs)* [16]. When PEs are used, establishing a desired navigation path may be a complex task, for example due to the requirement that correct typing must be obeyed along the path.

The goal of this paper is to demonstrate that various simplifications are possible when navigating through an OODB. The idea of simplifying (logical) navigation goes back to work on universal relation (UR) interfaces [4, 5, 15, 20].

[*]Research Assistant of the NFWO. Address: Dept. Math. & Computer Sci., University of Antwerp (UIA), Universiteitsplein 1, B-2610 Antwerp, Belgium. E-mail: vdbuss@uia.ac.be

[†]Address: Institut für Wirtschaftsinformatik, University of Muenster, Grevenerstr. 91, D-48159 Muenster, Germany. E-mail: vossen@informatik.rwth-aachen.de

There, the issue was to free users from the need to write down complex join expressions when querying a database; instead, they pose queries in terms of attributes alone, and the query processor has to expand this into an ordinary query, say, in relational algebra. Moreover, in [1] it has been shown how the UR model can be extended such that inheritance and aggregation information can be used to exploit attribute "compatibility," and how a UR query language can be provided with path expressions correspondingly.

Since PEs in an OODB can be viewed as collections of *implicit* (pre-computed) joins, it is natural to investigate the problem of simplifying logical navigation in this context as well. A first approach in this direction has been reported in [8], where the UR query language PIQUE [15] was adapted to the context of a semantic data model. [12] suggests to allow "wildcards" as abbreviations in PEs, which may be replaced by a sequence of attributes.

In the present paper, we will present a general abbreviation mechanism for PEs (without wildcards) which allows them to be specified more concisely. In particular, attributes may be omitted from the specification of a PE, and links between classes may be traversed in their given or in their inverse direction. In addition, our PEs can contain not only attribute names, but also class names. This renders them powerful enough to serve not only as where-clause specifications, but also as for-clause specifications (using SQL terminology). Indeed, in our approach to path expressions the distinction between where- and from-clauses becomes obsolete.

Abbreviated PEs are evaluated by "completing" them into so-called full ones. Completions will be inferred using knowledge of the inheritance and aggregation links from the underlying database schema. To this end, we associate with a schema a directed graph whose edges represent those links and are assigned weights based on a notion of "conceptual distance." This renders it possible to infer and disambiguate what a user has in mind by assuming that he is aiming for the *shortest* connection between data objects that is implied by his expression.

If several shortest paths exist, we take the union of them, inspired by the credulous approach in AI. In this process, we can avoid combinatorial explosion—a typical drawback of the credulous approach—by employing techniques from semantic query optimization [7]. Our credulous approach will also turn out to have the *liberal typing* for PEs, introduced in [12], as a special case. Note also the similarity in spirit with the window function for UR interfaces proposed in [4].

The core of our abbreviation mechanism will be described in the general framework of an object-oriented calculus-like query language. The formal semantics of this language is specified by a translation to the conventional relational calculus (using an obvious relational representation of an OODB). In this way, all semantical properties known of the relational calculus become readily accessible (e.g., for evaluation and optimization purposes). In particular, the implicit joins expressed by a PE are made explicit by the translation. Because our object-oriented calculus is so simple and general, it will be straightforward to incorporate our ideas into other query languages.

The organization of this paper is as follows: In Section 2, we introduce our model of OODBs, and motivate our work by discussing various shortcomings in the usage of traditional PEs in object-oriented query languages. In Section 3, we define the weighted schema graph corresponding to an object-oriented schema, using a relational representation of OODBs. In Section 4, we introduce our extended notion of PEs and study ways of automatically expanding abbreviations into full specifications. In Section 5 we exhibit a general framework for calculus-like query languages in which our PE mechanism can be used. We give formal semantics by a translation to relational calculus. This translation is shown to be safety-preserving. Finally, in Section 6, we briefly discuss a number of further issues that follow from our work. We go deeper into the matter of traversing aggregation links backwards (which we omit from the main formal development for clarity of presentation), and show how our results can be carried over into this more general setting. We also comment on the complexity of our abbreviation mechanism.

2 Motivation

To motivate our investigation, we first review the traditional notion of path expression as underlying most query languages for OODBs [16]. We indicate several shortcomings of the common usage of PEs, and show how these can be overcome. To make the presentation easier to read, we first sketch our OODB model and present a running example.

Following ORION [13], we assume that an OODB schema consists of a set of *class names* which are organized in an *isa* hierarchy. The classes are connected via aggregation links by associating with each class name a number of attribute declarations which are "local" to that class. Since we do not consider attribute overriding in this paper, we assume that if c isa c', then the sets of attributes specified for c and c' are disjoint. Of course, c will inherit all attributes from c'.

Each attribute has an arbitrary class name as type and an indication whether it is single- or set-valued. The type of an attribute is the class to which values of the attribute belong. Classes can be system-defined (e.g., *Integer*, *String*) or user-defined. We will focus on user-defined classes in this paper. Objects have a unique identifier and a possibly complex value consisting of all attribute values. For simplicity, we do not distinguish between attributes and (side-effect free) methods. This implies that we only consider methods without parameters; the generalization to methods with parameters is well-known and straightforward (e.g., [12]). Throughout this paper, we will use an abbreviated version of an example from [12] as our running example. The sample schema is depicted in the usual graphical way in Figure 1. In this figure, single arrows denote aggregation links and double arrows is-a links, and a star indicates that the attribute in question is set-valued.

Traditionally (e.g., [2]), a *path expression (PE)* over a given OODB schema is an expression of the form $x.A_1.A_2.\ldots.A_n$, $n \geq 1$, where x is a variable standing for an object of class C_1, A_1 is an attribute of that class, and for $1 < i \leq n$, A_i

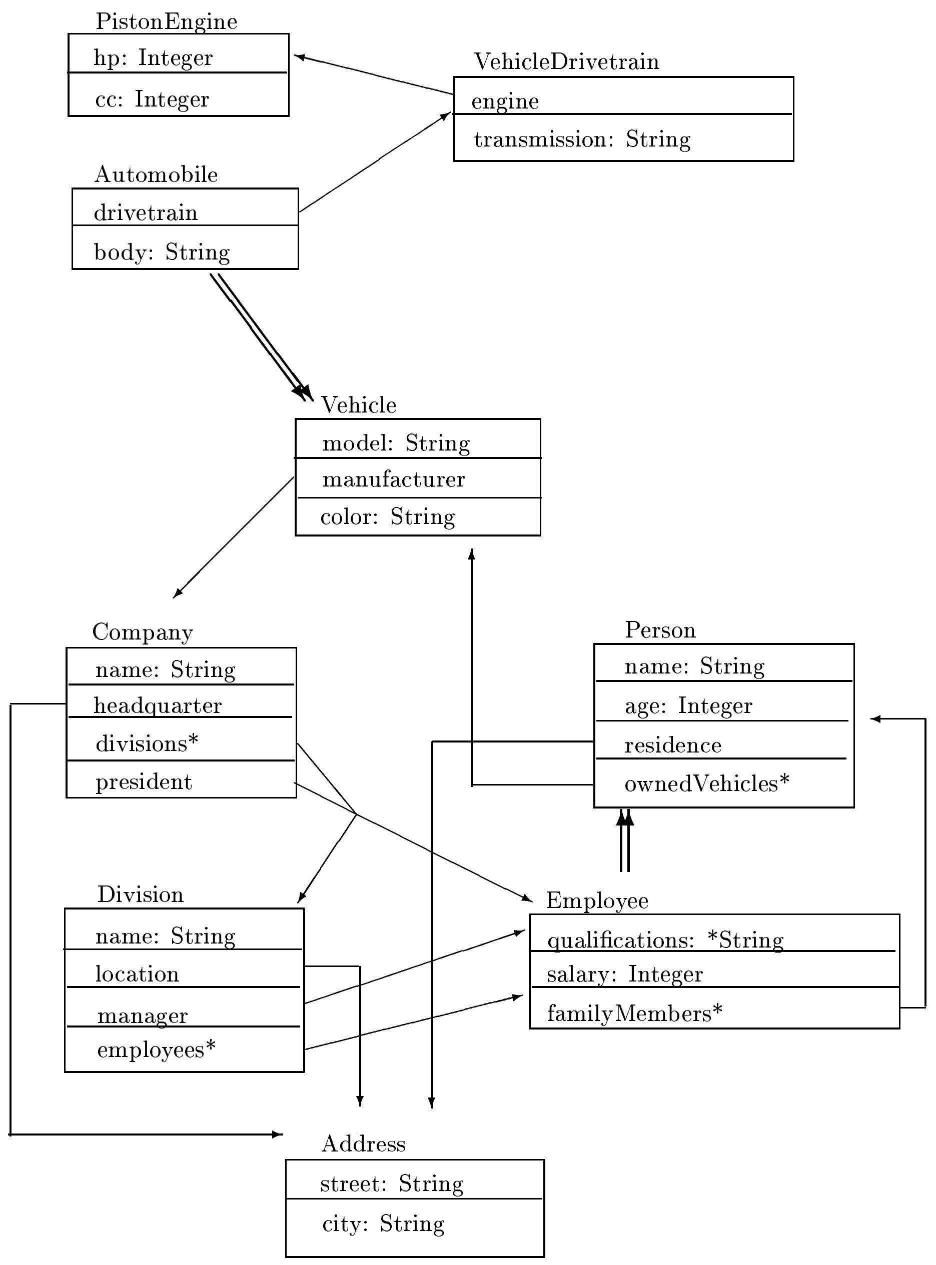

Figure 1: OODB schema of the running example.

is an attribute of class C_i, being the type of attribute A_{i-1} in class C_{i-1}.

This definition has several immediate consequences, which impose unnecessary limitations on the usage of PEs in queries. First, PEs have to be specified "in full," i.e., it is not allowed that a sequence $A_1.A_2.\dots.A_n$ of attributes is interrupted at any point. For example, if we ask for the cc value of an automobile x, we would have to use $x.drivetrain.engine.cc$. However, there is just one (and minimal) way to connect an automobile to a cc value, so that a PE like $x.cc$ suffices in principle. Note that this situation is comparable to the one where we ask for the color of automobile x. Traditionally, object-oriented systems allow to use $x.color$ for this purpose, although formally, *color* is a property of vehicles, not of automobiles. Nevertheless, the system can infer that the required connection is correct using the schema information on the inheritance link between the two classes involved. It follows that traditional PEs require a full specification unless a case of inheritance is involved, which results in an unsymmetric treatment of inheritance and aggregation links; we will demonstrate below that this is not necessary.

To illustrate a second limitation of traditional path expressions, consider the query asking for all cc values of automobiles owned by employees. In XSQL [12], a recently proposed query language which provides one of the most complete treatments of PEs to date, this query can be formulated as follows (recast into a calculus-like syntax here):

$$\begin{aligned}\{z \mid (\exists x : Employee(x))(\exists y : Automobile(y)) :\\ [x].ownedVehicles[y].drivetrain.engine.cc[z]\}\end{aligned}$$

Note the use of variables within the PE in the form of *selectors*, which was one of the main contributions of XSQL. Nevertheless, the above query formulation is still far from perfect. For example, the only use of variable y is to provide the link between (*owned*) *Vehicle* and *drivetrain*. Indeed, formally vehicles do not necessarily have an engine, while automobiles do; the $[y]$-selector enforces the additional requirement that the vehicle must actually be an automobile. It turns out that this trick can be performed automatically by the system in much the same way as the path abbreviation illustrated previously. It suffices to extend the notion of path expression to allow not only attribute names, but also class names to bind the objects participating in the path. This would lead to

$$\{z \mid (\exists x : Employee(x)) : [x].ownedVehicles.Automobile.drivetrain.engine.cc[z]\}.$$

But clearly, we can do away with the x variable in the same way, yielding

$$\{z \mid Employee.ownedVehicle.Automobile.drivetrain.engine.cc[z]\}.$$

Finally, basing ourselves on the observation (similar to an earlier one) that there is only one minimal way to connect employees to cc values, we can as well write:

$$\{z \mid Employee.cc[z]\}.$$

Our exposition below will demonstrate that this effect can formally be made precise. We also point out that a side-effect of the extension to allow class names inside path expressions is that the distinction between from- and where-clause specifications (using SQL terminology) becomes obsolete, in the sense that a qualification such as *Employee*(x) is equivalent to the PE *Employee*$[x]$ (which could occur as a subexpression of a larger PE).

As a final remark, the reader may have noticed that in the above discussion, we have implicitly considered inheritance links to be bidirectional, e.g., going from *Automobile* to *Vehicle* in the first example, and conversely in the second. Again, there is no reason why this should not hold for aggregation links as well. Indeed, several proposals have recently been made regarding this point [10, 14, 18]. A general discussion on the impact of "inversion" in path expressions will be delayed to Section 6. Until then, it suffices to say that an inversion mechanism will be easily incorporable in our proposal as described in the sequel.

3 Schema Graphs and Schema Paths

Given a schema in the model introduced in the previous section, a weighted graph can be associated with it whose edges capture the various inheritance and aggregation links comprised. We next introduce a simple and straightforward representation of the OODB model by the relational model. Using this representation we will be able to use well-understood terminology to describe the semantics of PEs.

It is well-known [11] how an OODB schema S can be represented by a relational database schema $D(S)$, consisting of unary and binary relation schemas, and a number of constraints; this representation is only used for formal or conceptual purposes and is implementation-independent. Unary relations hold the oid's associated with each class, while binary relations consist of an oid column plus one of the attributes of the corresponding class. Hence, for each class name c, there is a relation named c with c as the only attribute; for each attribute A of c there is a binary relation named c-A with attributes c and A (a tuple in c-A will consist of an oid associated with c and a value for A). The relations corresponding to our running example will include, among others:

```
Person(Person); Person-name(Person, name);
Employee(Employee); Employee-familyMembers(Employee, familyMembers);
Company(Company); Company-president(Company, president).
```

The constraints are functional, inclusion and exclusion dependencies [6], used to model the obvious properties of single-valued attributes, inheritance, referential integrity, and disjointness of classes unrelated in the inheritance hierarchy; in our running example, these will include, among others:

```
Person → name, Employee[Employee] ⊆ Person[Person]
Company-president[president] ⊆ Employee[Employee]
Vehicle-manufacturer[manufacturer] ⊆ Company[Company]
Company[Company] ∩ Person[Person] = ∅
```

Each given OODB schema can uniquely be represented in relational terms in this way. Hence, instances over the OODB schema can simply be introduced as instances over the relational representation satisfying the constraints. Again, we stress that this relational representation of an OODB is for conceptual purposes only; it allows us to rely on well-known (i.e., relational) concepts for explaining our ideas, and certainly does not imply that we assume the OODB to be stored on top of a relational system.

We next associate a schema graph with a given OODB schema, which straightforwardly follows from the relational representation of the latter. Since it will be our goal to show how a user can specify connections between database objects using PEs, this graph will turn out to be useful for making this explicit.

If S is an OODB schema, the *schema graph* $G(S)$ is a directed, weighted graph defined as follows: For each relation R in the relational representation of S there is a node in $G(S)$. If R is unary, i.e., corresponds to a class c, that node has the form '(c, c)'; if R is binary, i.e., corresponds to a class c and one of its attributes A, that node has the form '(c, A)'. The edges of $G(S)$ either connect "class identifiers" with their "component attributes", attributes with the class identifier of their type, or the class identifiers of isa-related classes. More specifically, we distinguish the following types of edges in $G(S)$, where c is a class and A is an attribute of c:

(a) an edge $(c, c) \to (c, A)$,

(b) an edge $(c, A) \to (c', c')$ if A is of type c',

(c) an edge $(c', c') \to (c, c)$ and an edge $(c, c) \to (c', c')$ if c isa c' holds in S.

Finally, we assign weights to the edges of a schema graph. The intuition behind these weights is to have a notion of "conceptual distance" between objects in the database. The goal will then be to allow a user to specify the *shortest* path or subpath between two given points by specifying these points only. Intuitively, the weights distinguish the cases where a join would be necessary in the relational representation from those where no join is needed; in the latter case, we simply assign weight 0. However, in the former case, instead of just assigning weight 1, we assign weight i $[a]$ if the edge represents an inheritance [aggregation] relationship, resp.; as will be seen, this allows us to capture precisely the semantics we have in mind for abbreviated PEs. In detail, weights are assigned as follows:

Edges of type (a) (using the above list) indicate the access of a class attribute. since attribute values are directly associated with oid's for each class, we assign these edges weight 0 (i.e., there is no conceptual distance). Edges of type (b) indicate how to go from one class to the next via an aggregation link; we assign these edges weight a. Edges of type (c) state how to go from a subclass to a superclass or vice versa. Subclass oids are a subset of their superclass oids, so we assign edges from sub- to superclass weight 0. Going from super- to subclass, e.g., to check whether a person object is in fact a student, is less trivial; these edges have weight i. Figure 2 shows an excerpt from the schema graph for our running example.

Now let S be schema and $G(S)$ the schema graph of S. A *schema path* is a sequence of nodes pairwise connected by an edge. If $p = (v_1, \ldots, v_n)$ is a schema

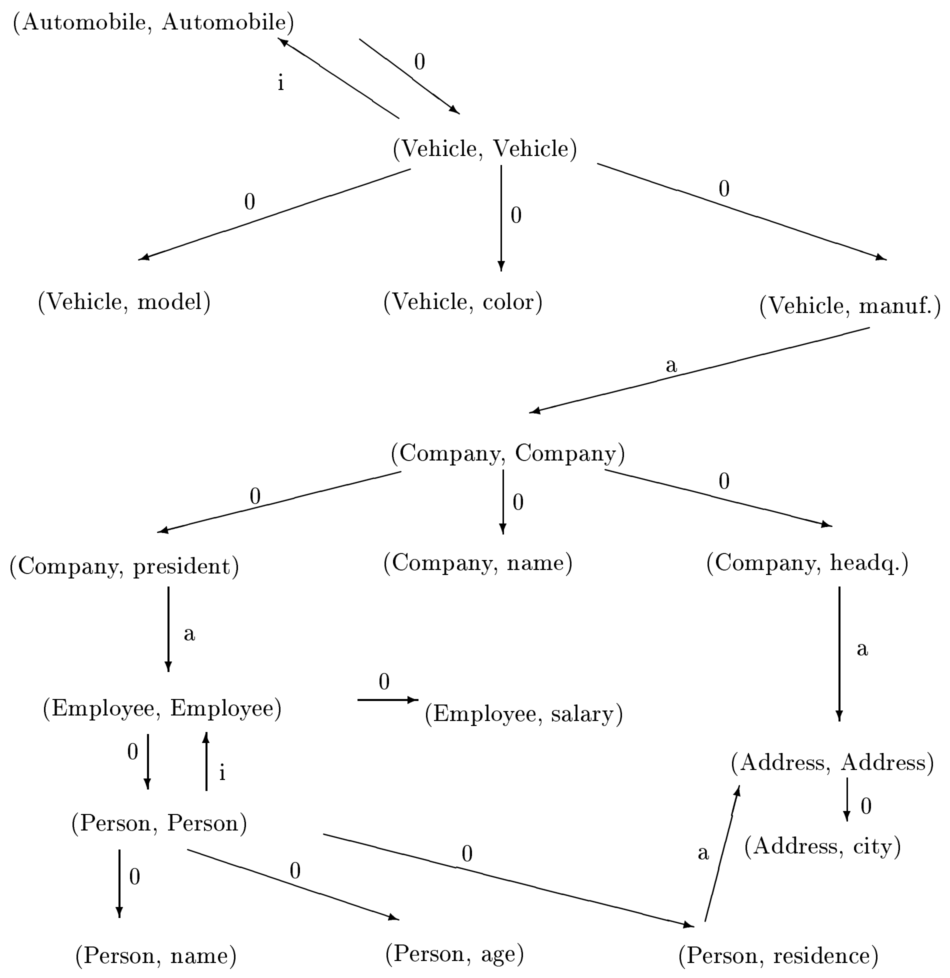

Figure 2: Part of the schema graph for our running example.

path s.t. $e_i = (v_i, v_{i+1})$ is an edge with weight $w(e_i)$, $1 \leq i < n$, the *weight* of p, denoted $w(p)$, is a string consisting of those $w(e_i)$ which are $\neq 0$. In our running example, the following are schema paths:

p_1 = *((Company, Company), (Company, headquarter), (Address, Address))*

p_2 = *((Company, Company), (Company, president), (Employee, Employee), (Person, Person), (Person, residence), (Address, Address))*

Thus, both p_1 and p_2 are paths from *(Company Company)* to *(Address, Address)*. Notice that $w(p_1) = a$, while $w(p_2) = aa$. Clearly, $a < aa$ if we compare these strings lexicographically. So given an expression of the form *Company.address* (which will be a valid path expression in our approach), it seems intuitively more reasonable to interpret this as the query asking for the address of a company's headquarter than as the query asking for the address of a company's president.

We are now ready to define the following partial order $<$ on schema paths which gives us the desired shortest path semantics: If p and p' are schema paths, then $p < p'$ if (i) $w(p) \leq w(p')$ in the lexicographical order given by $i < a$, and (ii) if $w(p) = w(p')$, then p is a subpath of p'. Here, a subpath means a path that can be obtained by omission of nodes. We can then call a schema path p from v_1 to v_n *minimal* if there is no other path p' from v_1 to v_n s.t. $p' < p$. For example, let p_1 and p_2 be as above. Then $p_1 < p_2$, since $w(p_1) \leq w(p_2)$. Next let

p_3 = *((Company, Company), (Company, president), (Employee, Employee), (Person, Person), (Person, residence))*

Then $p_3 < p_2$, since their weights are equal, but p_3 is a subpath of p_2.

To conclude this section, we give an illustration of the comparison between i and a weights. Suppose that *Person* objects also have an attribute *manager* which is of type *Employee*. An expression of the form *Person.salary* could then be interpreted either as the salary of a person as employee, or the salary of a person's manager. The former is represented by schema path *((Person, Person), (Employee, Employee), (Employee, salary))*, which has weight i, while the latter is represented by schema path *((Person, Person), (Person, manager), (Employee, Employee), (Employee, salary))*, which has weight a. Since $i < a$ the first interpretation is the minimal one. This matches the intuition that the first is the most natural interpretation of *Person.salary*, since it remains in the same "is-a context" of Persons and Employees, in the sense of Neuhold and Schrefl [17].

4 A Generalization of Path Expressions

Formally, we define a *path expression (PE)* as a string of the form

$$\{s_0.\}A_1\{s_1\}.\dots.A_n\{s_n\},\ n \geq 1$$

where each A_i, $1 \leq i \leq n$, is an attribute or a class name, and each s_j, $0 \leq j \leq n$, is a *selector* of the form $[t]$, where t is a variable or a constant. The curly braces indicate that selectors are optional.

Selectors were first proposed in XSQL as a means to bind variables to objects participating in the path, or to restrict such objects to a constant. For technical simplicity, we will assume that no PE with a class name in its first position begins with a selector. This assumption is harmless since such esoteric cases will not occur in practice. For example, $[p].Person$ is not allowed, but $Person[p]$, which is equivalent, is.

Notice that PEs as defined here need not be fully specified; a major difference to traditional PEs is that we allow abbreviations. Recalling from Section 2, the PE $Employee.ownedVehicles.Automobile.drivetrain.engine.cc[z]$ can be alternatively stated as, e.g., $Employee.Automobile.cc[z]$. The basic idea for giving such incomplete expressions a semantics is to expand them into "full" expressions, using the weight of the corresponding schema paths as a selection criterion in case there are several expansions, i.e., to use the minimal expansion. For example, in our running example there exist several paths from class *Employee* to class *Automobile*; in terms of the schema graph from Figure 2, these are:

p_1 = *(Employee, Employee), (Person, Person), (Person, ownedVehicles), (Vehicle, Vehicle), (Automobile, Automobile)*

p_2 = *(Employee, Employee), (Employee, familyMembers), (Person, Person), (Person, ownedVehicles), (Vehicle, Vehicle), (Automobile, Automobile)*

Notice that $w(p_1) = ai$, while $w(p_2) = aai$; thus, p_1 yields a "shorter" connection between employees and engines (a schema path with smaller weight), so p_1 would be used to expand the above PE. Minimal expansions are formally defined below.

A PE $A_1.\ldots.A_n$, $n \geq 1$, where selectors are ignored, is said to be *full* if A_1 is a class name, and there exist nodes $v_i = (c_i, A_i)$, $1 \leq i \leq n$, in the corresponding schema graph s.t. $(v_1 \ldots v_n)$ is a path in that graph. The following can be proved by induction:

Lemma 1 *For every full PE $A_1 \ldots A_n$ as above, the corresponding schema path $(v_1 \ldots v_n)$ is* unique. *Furthermore, $v_1 = (c_1, A_1)$ satisfies $A_1 = c_1$.*

As mentioned in Section 2, our full PEs are already more powerful than traditional ones, since they can contain class names (serving the same function as from-clause bindings in languages like XSQL); this generalization is necessary for our abbreviation mechanism to be flexible.

Now let p be a given PE. An *expansion* of p w.r.t. the underlying schema is any full PE p' which contains p as a subexpression, and contains not more selectors than p. Here, a subexpression means an expression that can be obtained by an omission of symbols. Finally, a *minimal* expansion is an expansion whose corresponding schema path is minimal.

Notice that the expansion of a given PE thus starts with a class name and has an associated schema path. Expansions of PEs are in general not unique. Continuing the previous example, the two expansions of PE $Employee.Automobile[x]$ are $Employee.Person.ownedVehicles.Vehicle.Automobile[x]$
and $Employee.familyMembers.Person.ownedVehicles.Vehicle.Automobile[x]$.

The first one is minimal and indeed corresponds to the intended meaning. Clearly, full PEs contain a lot of redundant class names, but these will be eliminated by the reduction procedure introduced in the next section.

We finally note again that the process of expanding a given PE will not always have a unique solution. For example, the expression *name[Johnson]* should yield all objects for which *name* is defined and equal to Johnson. Notice that these objects could be either companies or persons, since both corresponding classes have an attribute *name*. In other words, there are two minimal expansions *Company.name* and *Person.name*. Under a "credulous" approach, the evaluation procedure as defined in the next section will take their union. We point out that the *liberal typing* of PEs, introduced in [12], can be explained as a special case of this credulous approach.

5 Evaluating Generalized Path Expressions

In this section, we demonstrate that our extended mechanism for PEs can be adopted by various query languages for OODBs. To support this claim, we exhibit a general, calculus-like language called OOC ("object-oriented calculus"), whose syntax employs PEs as introduced in the previous section, and whose semantics will be stated in terms of the conventional relational calculus over the relational representation of OODB schemas given in Section 3.

Building up from PEs and comparisons as atoms, we define formulas and queries of OOC syntactically in the standard manner:

(i) An *atomic formula* is either a PE or of the form $t \, \Theta \, t'$, where t and t' are terms (i.e., variables or constants) and Θ is a comparison symbol;

(ii) if Φ, Φ_1, and Φ_2 are formulas and x is a variable, then $\Phi_1 \wedge \Phi_2$, $\Phi_1 \vee \Phi_2$, $\neg\Phi$, $(\exists\ x)\ \Phi$, and $(\forall\ x)\ \Phi$ are formulas.

Notice the simple and uniform format of our atomic formulas; path expressions (and comparisons) are all that is needed due to the possible presence of class names in PEs. In particular, an equivalent to from-clauses as in XSQL is not needed. We define *free* and *bound* variables in a formula in the standard way [19]. An *OOC query* is now an expression of the form

$$\{x_1 \ldots x_n \mid \Phi(x_1, \ldots, x_n)\},$$

where formula Φ has exactly $x_1, \ldots, x_n$ as free variables.

As a simple example, $\{c \mid \mathit{manufacturer}.\mathit{city}[c]\}$ is an OOC query. As will become clear, this query asks for all cities where a manufacturer (of vehicles) is located. As another example,

$$\{n \mid (\exists e)\mathit{Employee}[e].\mathit{name}[n] \wedge (\forall a)(\forall x)[e].\mathit{Automobile}[a] \wedge [a].\mathit{cc}[x] \Rightarrow x > 2000\}$$

asks for the names of those employees, all whose owned automobiles have an engine with cc larger than 2000.

We now indicate a precise way to define the semantics of an OOC query, mostly by way of examples. Since we assume that an OODB can conceptually be represented as a relational database, and since the semantics of relational calculus is well-known, it suffices to translate each PE occurring in the query into a relational calculus subformula that evaluates to the intended meaning of the PE. The relational calculus query resulting from the replacement of all PEs by their translations can then be evaluated over the relational database we associated to the OODB schema in Section 3. We emphasize again that this is only a conceptual approach.

In the previous section, we associated with each PE a number of *full* PEs, namely, its minimal expansions. Recall from Lemma 1 that full PEs have a unique corresponding schema path. This schema path is the key to evaluating the full PE. However, since full PEs are as explicit as possible, they contain a lot of redundant class names which would cause redundant steps in the evaluation procedure. We eliminate this redundancy as follows. Let p be a full PE. The *reduction* of p is obtained by eliminating each class name occurring in it, provided the elimination does not incur a loss of variables, and provided the class name does not occur either at the beginning or the end of the path.

Recall the query $\{c \mid \mathit{manufacturer.city}[c]\}$. Path expression $\mathit{manufacturer.city}[c]$ has several expansions. For example,

$$\mathit{Vehicle.manufacturer.Company.divisions.Division.employees.Employee.}$$
$$\mathit{Person.residence.Address.city}[c]$$

is an expansion (with weight $aaaa$). However, the intended meaning of the query is *not* the cities of the employees of manufacturers, but rather the cities of the headquarter of the manufacturer itself. This corresponds to

$$\mathit{Vehicle.manufacturer.Company.headquarter.Address.city}[c]$$

which is a *minimal* expansion (with weight aa), in this example the only one. Reduction yields $\mathit{manufacturer.headquarter.city}[c]$. The extent will now be a relational calculus subformula having as free variables precisely the variables occurring as selectors in p; in the example just considered, we thus obtain:

$$(\exists\, t_{11})\ldots(\exists\, t_{32})\ (\mathit{Vehicle\text{-}manufacturer}(t_{11} : \mathit{Vehicle}, t_{12} : \mathit{manufacturer}) \wedge$$
$$\mathit{Company\text{-}headquarter}(t_{21} : \mathit{Company}, t_{22} : \mathit{headquarter}) \wedge$$
$$\mathit{Address\text{-}city}(t_{31} : \mathit{Address}, t_{32} : \mathit{city}) \wedge t_{12} = t_{21} \wedge t_{22} = t_{31} \wedge t_{32} = c)$$

In general, the *extent* of a full path expression p whose reduction is of length m is defined as the relational calculus subformula

$$(\exists)(R_1 \wedge \cdots \wedge R_m \wedge \mathit{equalities}),$$

where each R_i is of the form $c_{j_i}\text{-}A_{j_i}(t_{i1} : c_{j_i}, t_{i2} : A_{j_i})$ or $c_{j_i}(t_{i1})$, t_{i1} and t_{i2} are domain variables for the attributes, and the equalities are a conjunction of atomic join or selection conditions (where the latter stem from the selectors in p), and where $(\exists)$ denotes the full existential closure of all t_{ij} variables.

Having defined extent(p) for full PEs, we now define the translation of an arbitrary PE p' as the disjunction $\bigvee_{m \in M} \text{extent}(m)$, where M is the set of minimal expansions of p'. This formalizes the credulous approach mentioned earlier.

For a different example, assume that the database schema also has a class *Professor* isa *Employee*, whose attribute *teaches* can take a set of *Courses* as value. The query $\{p \mid \mathit{president}[p].\mathit{teaches}[\mathrm{math}]\}$ asks for presidents of companies who happen to be professors as well, and teach math. The PE again has only one minimal expansion, $\mathit{president}[p].\mathit{Employee}.\mathit{Professor}.\mathit{teaches}[\mathrm{math}]$, which has as extent:

$$(\exists\ c)(\exists\ o)\ (\mathit{Company\text{-}president}(c,p) \wedge \mathit{Professor\text{-}teaches}(p,o) \wedge o = \mathrm{math})$$

We now briefly comment on the notion of *safety* in our OOC context. There is a syntactical notion of safety for relational calculus queries [19] which guarantees that they can be evaluated in finite time. We can adapt the definition of [19] to the OOC calculus; the only difference is that atomic formulas may now be PEs. We then obtain the following desirable property (the proof is omitted):

Theorem 1 *If an OOC formula is safe in the adapted sense, then its relational calculus translation is safe in the ordinary sense.*

We now return to the query presented earlier, which brings up the issue of optimization:

$$\{n \mid (\exists e)\mathit{Employee}[e].\mathit{name}[n] \wedge (\forall a)(\forall x)[e].\mathit{Automobile}[a] \wedge [a].\mathit{cc}[x] \Rightarrow x > 2000\}.$$

This query contains three PEs. If we translate each of them separately into relational calculus, we loose their interconnections, resulting in a poor overall translation. For instance, consider the second PE $[e].\mathit{Automobile}[a]$. When looked upon in isolation, e can be interpreted as a Person or an Employee. Formally, the PE has two minimal expansions. However, from the first PE, in which e also participates, it is clear that only the Employee interpretation is relevant to the query. So, when writing out the overall translation in safe form, it will contain a conjunct of the form:

$$\mathit{Employee}(e) \wedge \mathit{Person}(e) \wedge \ldots,$$

which can be simplified by removing the atom $\mathit{Person}(e)$, since we have the inclusion dependency $\mathit{Employee} \subseteq \mathit{Person}$. We are then left with two syntactically equal disjuncts, one of which can be eliminated.

There exist similar situations where exclusion dependencies can be used to eliminate disjuncts. For example, in the formula $\mathit{president}[p].\mathit{age}[60] \wedge [p].\mathit{name}[n]$, retrieving all names n of presidents aged 60, the second PE, in isolation, can interpret p as a Company or a Person; both classes have an attribute *name*. However, using the first PE, we know that p must be a person. Therefore, the absurd subformula of the form $\mathit{Company\text{-}president}(c,p) \wedge \mathit{Company\text{-}name}(p,n) \wedge \ldots$, which will appear in the result of the naive translation, can be eliminated. Indeed, this subformula is unsatisfiable, because of the dependencies $\mathit{Company\text{-}president}[\mathit{president}] \subseteq \mathit{Employee}$, $\mathit{Employee} \subseteq \mathit{Person}$, $\mathit{Company\text{-}name}[\mathit{Company}] \subseteq \mathit{Company}$, and $\mathit{Person} \cap \mathit{Company} = \emptyset$.

It turns out that the query simplifications just illustrated can be automated, using techniques known from semantic query optimization. [7] describes a powerful optimization algorithm which takes as input a non-recursive Datalog query with stratified negation, and a set of integrity constraints written as Horn clauses. The algorithm simplifies the query using the information from the constraints, and it can be adopted in our context roughly as follows:

By Theorem 1, the relational calculus translation of a safe OOC query is safe, and hence can be translated into a non-recursive Datalog query with stratified negation. Furthermore, recall from Section 3 that our constraints are simple cases of functional, inclusion, and exclusion dependencies. These dependencies can easily be rewritten as Horn clauses. For our purposes, only the inclusion and exclusion dependencies are relevant. To give the query optimizer as much information as possible, we do not only provide it with the explicitly given dependencies, but also with their logical consequences. Because the dependencies are "unary", it is relatively straightforward to generate their logical closure by a transitive closure-like procedure. The algorithm from [7] will then simplify the relational calculus translation of a safe OOC query as desired.

6 Discussion

We have argued for a more general perspective on PEs, going beyond a purely navigational usage. Extending their capabilities to specify connections between objects, we showed how incompletely specified expressions can be evaluated in a transparent way. This generalizes previous work in the context of the relational model on UR interfaces. On the other hand, our approach is general enough to be incorporated into vastly any query language for OODBs, e.g., SQL extensions as described in [16].

Our view of PEs as a connection specification mechanism is based on a generalization of PEs which allows them to contain class names in the middle, not just in the beginning. Indeed, this can be seen as an elegant way to avoid the use of "from-clauses" by explicitly including class bindings into a path. As a result, the notion of PE has been extended in several ways: Links existing between the classes of an OODB schema can freely be used in PEs in either direction, and abbreviations may be applied wherever the user wants them.

For reasons of clarity, we above restricted the use of inversions to cases where a path goes from a superclass to one of its subclasses, i.e., to inheritance links. However, our general approach to database connections also works for aggregation links and hence allows for *bi-directional* paths, i.e., paths in which attribute links may be traversed in forward or backward direction, in general. We briefly describe an example next.

Consider the expression *Employee.location*. If we just consider uni-directional paths, this expression can only be interpreted as retrieving for each employee the locations of the divisions of those companies that are manufacturers of some vehicle owned by that employee. However, a much more natural interpretation of the above expression would be to retrieve for each employee the location of the division where he is employed. This connection traverses the *employees* link backwards, as the corresponding full expansion shows:

$$Employee.employees^{-1}.Division.location.$$

Inversions in PEs can also be used explicitly, allowing for the convenient formulation of certain queries. Suppose that we ask for the names of all companies of which Perot is president. Without inversions, this query requires an evaluation of two PEs: First, *Company[c].name[n]* is used to obtain company names. Second, *[c].president.name[Perot]* is used to select those companies of which Perot is president. By allowing to use links in both directions, the same could be obtained in one expression as follows:

$$[Perot].name^{-1}.president^{-1}.Company.name[n]$$

It can be shown that all technical results presented in the preceding sections can be carried over to the generalized setting of PEs containing inversions as well. Of course, since more connections can now be followed, a PE will have less chance of having only one minimal expansion. So, the user will not always be able to use abbreviations as dramatically as is possible in the uni-directional case. However, this will largely be compensated by the added ability to use inversions.

We mention that our PE mechanism could be extended further by explicitly adding a fixpoint operator for dealing with recursion due to cycles in an OODB schema. Suppose a class *Person* has an attribute *name* of type *String*, and an attribute *child* of type *Person*. If we ask for all grandchildren of John, our current proposal allows to express this using the PEs

Person[o].name[John] and *[o].child.child.name[x]*

Asking for all descendants of John could be handled by a fixpoint construct of the form $[o].child^*.name[x]$, as in [9].

We conclude this paper with a comment on the complexity of our PE expansion mechanism. Finding minimal expansions of PEs can be accomplished using well-known efficient shortest-path algorithms. However, in case multiple minimal expansions exist, our credulous approach will take the union of all of them. This is a hidden source of complexity. Indeed, it is not difficult to construct a database schema where two classes are connected by an exponential number of paths of the same weight. However, in practice, the number of minimal expansions will usually be low, and their union can often be simplified using the query simplification techniques described at the end of the previous section.

Acknowledgement. The authors are grateful to Catriel Beeri and Michael Schrefl for helpful comments on an earlier version of this paper.

References

[1] C. Beeri, H.F. Korth: Compatible Attributes in a Universal Relation; Proc. 1st ACM PODS 1982, 55–62

[2] E. Bertino, W. Kim: Indexing Techniques for Queries on Nested Objects; IEEE TKDE 1, 1989, 196–214

[3] E. Bertino, M. Negri, G. Pelagatti, L. Sbattella: Object-Oriented Query Languages: The Notion and the Issues; IEEE TKDE 4, 1992, 223–237

[4] J. Biskup, H.H. Brüggemann: Universal Relation Views: A Pragmatic Approach; Proc. 9th VLDB 1983, 172–185

[5] V. Brosda, G. Vossen: Update and Retrieval in a Relational Database through a Universal Schema Interface: ACM TODS 13, 1988, 449–485

[6] M.A. Casanova, V.M.P. Vidal: Towards a Sound View Integration Methodology; Proc. 2nd ACM PODS 1983, 36–47

[7] U.S. Chakravarthy, J. Grant, J. Minker: Logic-Based Approach to Semantic Query Optimization; ACM TODS 15, 1990, 162–207

[8] T.H. Chang, E. Sciore: A Universal Relation Data Model with Semantic Abstractions; IEEE TKDE 4, 1992, 23–33

[9] M. Consens, A. Mendelzon: GraphLog: A Visual Formalism for Real Life Recursion; Proc. 9th ACM PODS 1990, 404–416

[10] A. Heuer, J. Fuchs, U. Wiebking: OSCAR: An Object-Oriented Database System with a Nested Relational Kernel; in: H. Kangassalo (ed.), *Entity-Relationship Approach: The Core of Conceptual Modeling*, North-Holland 1991, 103–118

[11] R. Hull, M. Yoshikawa: ILOG: Declarative Creation and Manipulation of Object Identifiers; Proc. 16th VLDB 1990, 455–468

[12] M. Kifer, W. Kim, Y. Sagiv: Querying Object-Oriented Databases; Proc. ACM SIGMOD 1992, 393–402

[13] W. Kim: *Introduction to Object-Oriented Databases*; The MIT Press, Cambridge, MA, 1990

[14] C. Lamb, G. Landis, J. Orenstein, D. Weinreb: The ObjectStore Database System; CACM 34 (10) 1991, 50–63

[15] D. Maier, D. Rozenshtein, S. Salveter, J. Stein, D.S. Warren: PIQUE: A relational query language without relations; Information Systems 12, 1987, 317–335

[16] F. Manola: Object Data Language Facilities for Multimedia Data Types; Techn. Report TR-0169-12-91-165, GTE Labs., Inc., 1991

[17] E.J. Neuhold, M. Schrefl: Dynamic Derivation of Personalized Views; Proc. 14th VLDB 1988, 183–194

[18] M.H. Scholl, C. Laasch, C. Rich, H.J. Schek, M. Tresch: The COCOON Object Model; Techn. Report 193, Departement Informatik, ETH Zürich 1992

[19] J.D. Ullman: *Principles of Database and Knowledge-Base Systems* Vol. I; Computer Science Press, Rockville, MD, 1988

[20] G. Vossen: *Data Models, Database Languages, and Database Management Systems*; Addison-Wesley 1991