Applying an update method to a set of receivers
(extended abstract)

Marc Andries, Leiden University*
Luca Cabibbo, University of Rome†
Jan Paredaens, University of Antwerp‡
Jan Van den Bussche, INRIA§

Abstract

In the context of object databases, we study the application of an update method to a collection of receivers rather than to a single one. The obvious strategy of applying the update to the receivers one after the other, in some arbitrary order, brings up the problem of order independence. On a very general level, we investigate how update behavior can be analyzed in terms of certain schema annotations called colorings. We are able to characterize those colorings which always describe order-independent updates. We also consider a more specific model of update methods implemented in the relational algebra. Order independence of such algebraic methods is undecidable in general, but decidable if the expressions used are positive. Finally, we consider an alternative parallel strategy for set-oriented application of algebraic methods and compare and relate it to the sequential strategy.

1 Introduction

In object systems, update procedures are provided by methods, which are applied to a receiver consisting of a receiving object and some argument objects. Since methods may call other methods, an update method applied to a certain receiver may not only update the properties of the receiving object, but may also have side effects. Hence, at the most general level, we can define an update method as a computable function mapping a given object base instance and a receiver to some new object base instance.

Database systems deal with whole collections of data at a time. Hence, in the context of object databases, it is important to be able to apply an update method to a collection of receivers rather than to a single one. For example, given a method to change the salary of an employee, we sometimes want to change the salaries of a whole group of employees. The purpose of the present paper is to initiate a study of various strategies for set-oriented application of update methods.

One obvious such strategy is to apply the update to the receivers one after the other, in some arbitrary order. This sequential application immediately brings up the problem of order independence: is the outcome of the sequential application independent of the order chosen? We consider three notions of order independence: absolute order independence on all possible sets of receivers; key-order independence on sets of receivers not containing a same receiving object twice with different arguments; and query-order independence on sets of receivers produced by some given query. The assumptions made by key-order independence and query-order independence are often true in practice.

On a very general level, we investigate how update behavior can be analyzed with respect to order independence in terms of certain schema annotations, which "color" each class and property name by indicating whether the update creates, deletes, or uses information of this type. While it is not difficult to formalize what it means for an update to create or delete information of a certain type, it is much less obvious how the semantics of "using information" can be axiomatized. We have studied two possible such axiomatizations, and were able to show in both cases that the colorings which describe order-independent updates are precisely those that are "simple," in a sense to be made precise. This captures the intuition that the update does not perform potentially conflicting actions. Curiously, it will turn out that

*Department of Mathematics and Computer Science, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, the Netherlands. E-mail: andries@wi.leidenuniv.nl

†Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, I-0019 Roma, Italy. E-mail: cabibbo@dis.uniroma1.it. Partially supported by MURST and CNR.

‡Dept. Math. & Computer Sci., University of Antwerp (UIA), Universiteitsplein 1, B-2610 Antwerpen, Belgium. E-mail: pareda@uia.ac.be

§INRIA (Projet VERSO), Domaine de Voluceau, Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France. E-mail: vdbuss@uia.ac.be. On leave from the University of Antwerp. Research Assistant of the Belgian National Fund for Scientific Research.

the two "axiomatizations of use" we propose are each other's dual, in the sense that the first favors inflationary updates while the second favors deflationary ones.

On a more specific level, we consider update methods implemented in the relational algebra, using a framework inspired by the algebraic model for accessing object-oriented databases proposed by Hull and Su [10]. Methods in this framework can only update properties of the receiving objects. We observe that order independence of algebraic methods is undecidable in general, but it becomes decidable if only positive expressions are used. Specifically, we establish mutual reductions between the problem of testing for order independence of an algebraic method and the problem of testing equivalence of relational algebra expressions under certain dependencies implied by the relational representation of object databases. The latter problem is shown decidable for positive expressions by combining classical techniques from relational database theory. We also present a sufficient condition for key-order independence which explains the **update** command of SQL.

Apart from the sequential strategy for set-oriented application, we also consider a natural alternative in the algebraic framework. This strategy is parallel in that it instantiates the parameter in the method body, which normally stands for a single receiver, by the whole set of receivers at once. In this approach, there is no problem of order independence; every parallel set-oriented application is well-defined. Hence, it is interesting to ask whether every order-independent algebraic method can be "parallelized," i.e., whether for each such method M there exists another method M' such that each sequential application of M yields the same result as the corresponding parallel application of M'. By observing that sequential application can express transitive closure and parity, we answer this question negatively. Nevertheless, in the important special case of key order-independence, parallelization is always possible; we actually show that for key order-independent methods the sequential and the parallel semantics coincide.

Our work relates to a lot of other work reported in the literature on database query and update languages. Recently, Laasch and Scholl [14] studied order independence of updates expressed as sequences of generic operations such as insert, delete and modify, in the context of object-oriented databases. They argued that the problem directly links to issues in concurrency control, and proposed to disallow the use of potentially conflicting operations within an update sequence so as to guarantee order independence.

But also less recently researchers have pointed at the intricacies involved in set-oriented application of updates. Most notably, Aho and Ullman [3] considered sequential and parallel execution strategies for looping constructs of the form **for each** t **in** R **do** in database manipulation languages which are closely analogous to the sequential and parallel strategies we consider in the present paper. They questioned the appropriateness of the sequential strategy, however, since sequential application is (of course) not always guaranteed to be order independent. More subtly, Chandra [8] proposed the study of when and how for-each loops can be given a deterministic semantics, in the context of a programming language based on relational algebra and relational assignment, as an interesting research issue. We like to think of our work as first steps in this direction.

It should be noted that for-each loops have also been used as a potentially non-deterministic construct, e.g., in the work of Qian [17] or in the language SETL [19]. In this respect it is also interesting to note that the parallel strategy as a means to provide an alternative deterministic semantics to such constructs is very similar in spirit to the "relationally computable" semantics of a rule in a non-deterministic rule triggering system introduced by Simon and de Maindreville [20].

To conclude, we must point out that different, "coarser grained" parallel interpretations of for-each loops than the one we have considered up to now also have received considerable attention in the literature. Abiteboul and Vianu [1] defined a parallel semantics for applying an update to a set of receivers which first computes the different effects of the update applied to each receiver separately, and then combines the obtained results by taking the union. This combination approach is comparable to that of structural recursion [5, 4] where the different results of a function parameterized by the elements of a set are collected using a commutative and associative accumulation operator. Abiteboul and Vianu gave evidence that as combination operator, a simple union is in principle sufficient. Nevertheless, the study of combination operators more sophisticated than union and their relationship to the other semantics is, in our opinion, an interesting issue for further research. One which seems to be well-behaved is the operator combining the output databases $D_1, \ldots, D_n$ for the different receivers on some input database D as $\bigcap_i D_i \cup \bigcup_i (D_i - D)$.

The remainder of this text contains a technical exposition of the issues discussed above, in the order as they were treated. We will refer to the Appendix for sketches of proofs.

2 Update methods

In this section, we present a few preliminary definitions.

It is customary in object-based models to depict a database schema as a graph. Thereto, we assume the existence of disjoint sets of *class names* and *property names*, and define an *object-base schema* as a finite, edge-labeled, directed graph. The nodes of the graph are class names, and the edges are triples (B, e, C), where B and C are nodes and the edge label e is a property name. Different edges must have different labels. If (B, e, C) is an edge in the schema, we call e a *property of B of type C.*

An object-base instance can now be defined as a graph consisting of objects and property-links, whose structure is constrained by some object-base schema. So, we assume that for each class name C there is a universe of *objects of type C*, such that different class names have disjoint universes. For an arbitrary schema S, we then define an *instance* of S as a finite, labeled, directed graph. The nodes of the graph are objects. Each node o is labeled by its type $\lambda(o)$, which must be a class name of S. The edges are triples (o, e, p), where o and p are nodes and the edge label e is a property name of S such that $(\lambda(o), e, \lambda(p))$ is an edge of S. The set of all objects in an instance labeled by the same class name C will be called *the class C.*

We now turn to update methods. An update method has a signature, specifying the types (class names) of the receiving object and the argument objects, and a behavior, which for the time being we define simply as some computable update of the object base instance. Formally, we have the three following definitions:

A *method signature* σ over S is a non-empty tuple of class names in S. The first element of the signature is called the *receiving class* of σ; the remaining positions in σ comprise the *argument classes.*

Given a method signature $\sigma = [C_0, \dots, C_k]$ over S and an instance I of S, a *receiver* over I of type σ is a tuple of the form $[o_0, \dots, o_k]$, where $o_0, \dots, o_k$ are objects in I of types $C_0, \dots, C_k$, respectively. The first object o_0 is called the *receiving object*; the remaining tuple $o_1, \dots, o_k$ comprises the *arguments* of the receiver.

Finally, given a method signature σ over S, an *update method M* of type σ is a computable function which, when given an instance I of S and a receiver t over I of type σ, yields an instance $M(I, t)$ of S.

3 Sequential application

In this section, we introduce the sequential application of an update method to a set of receivers as well as three different notions of order independence of a method. In what follows, we fix a schema S, a signature σ over S, and a method M of type σ.

We can apply an update method to a sequence, not a set, of receivers in the obvious way. So, if I is an instance and $s = t_1, \dots, t_n$ is a sequence of distinct receivers, $M(I, s)$ equals $M(\dots M(I, t_1), \dots, t_n)$, provided the value of this expression is well-defined (this may fail if, e.g., t_2 is not a receiver over $M(I, t_1)$).

Sequential application to a set of receivers may now be defined formally as follows:

Definition 3.1 Given an instance I and a set T of receivers, we say that M is *order independent on (I, T)* if for any two sequential enumerations s and s' of T, we have $M(I, s) = M(I, s')$. In this case we define the *sequential application $M_{\text{seq}}(I, T)$ of M on (I, T)* as $M(I, s)$ for an arbitrary sequential enumeration s.

The above definition leads to three global notions of order independence:

- If M is order independent on any pair (I, T) then M is called *order independent.*
- Call a set of receivers T a *key set* if, viewing T as a relation, the first column (holding the receiving objects) is a key for T. If M is order independent on any pair (I, T) where in T is a key set then M is called *key-order independent.*
- Finally, let Q be a function which maps each instance I to a set $Q(I)$ of receivers. If M is order independent on $(I, Q(I))$ for any I then M is called *Q-order independent.*

Example 3.2 We will use Ullman's classical example schema containing class names Drinker, Bar, and Beer, with Drinker having properties 'frequents' and 'likes' of types Bar and Beer respectively, and Bar having property 'serves' of type Beer. Consider the following two methods of type [Drinker, Bar]: *add_bar*, which adds the argument bar to those frequented by the receiving drinker, and *favorite_bar*, which removes all edges from the receiving drinker to bars currently frequented, and adds a single new one to the argument bar. The method *add_bar* is order independent, but *favorite_bar* is not. However, *favorite_bar* is key-order independent, and hence also Q-order independent for any query Q producing a list of drinkers and bars with a unique favorite bar for each drinker. Such a query might, for example, retrieve for each drinker the bar serving all beers that drinker likes, if unique and existing. ■

If we define methods as general computable functions, as we did, all of the notions of order independence defined above are undecidable, by Rice's theorem. We will later show however that order independence is decidable for more restricted kinds of methods. Thereto, we will rely on the following lemma, which is quite standard once we recall that any permutation can be written as a composition of transpositions of adjacent elements.

Lemma 3.3 *Method M is order independent if and only if M is order independent on any pair (I, T) where T consists of two elements.*

Lemma 3.3 also holds for key-order independence: we then have to consider sets T consisting of two elements with different receiving objects. However, the lemma fails in the case of query-order independence; we will come back to this issue in Section 5.

4 Schema colorings

In this section, we present the beginnings of a theory of schema colorings. Such colorings describe the behavior of an update by annotating each type of information in the schema with a subset of the letters **c**, **d**, or **u**, thereby indicating whether the update creates, deletes, or uses information of this type. The main difficulty which we encounter here is to formalize what it means for an update to "use" information of a certain type. We investigate two possible definitions, and in both cases characterize those colorings which describe order-independent updates.

Before we move on with the technical exposition, let us make clear that we do not intend to claim that colorings based on three letters are rich enough as a tool to analyze the behavior of updates. Indeed, although our results are perhaps satisfying from a purely mathematical point of view, their practical usefulness is rather limited. A study of colorings which can distinguish between more kinds of update behavior is an interesting issue for further research.

In what follows, we fix a schema S and a method signature over S. An *item* of a graph is either a node or an edge of that graph. Since schemas and instances are graphs, this terminology applies to both of them; it will be used a lot in the sequel.

It is not difficult to formalize what it means for an update to create or delete information of a certain type. Let X be a schema item. An update method M is said to *create information of type* X if there exists an instance I and a receiver t over I such that $M(I,t)$ contains an item labeled X that is not in I. The dual notion of when M *deletes information of type* X is defined analogously.

In order to define what it means for an update to use information of a certain type, we introduce the auxiliary notion of *partial instance*. A partial instance is a structure that can be obtained from an instance by removing some items. So, a partial instance differs from an instance in that it may contain "dangling edges," since a node may be removed without removing all its incident edges. The instance obtained from a partial instance J by removing all dangling edges is denoted by $\mathcal{I}(J)$. Note that, by viewing a partial instance as a set consisting of nodes and edges, we can apply set-theoretic operations such as union and difference to partial instances.

If $\mathcal{X}$ is a set of schema items and I is an instance of S, the *restriction of* I *to* $\mathcal{X}$ is the partial instance obtained by removing from I all items whose label is not in $\mathcal{X}$, and is denoted by $I|_{\mathcal{X}}$.

We now introduce our first proposed axiomatization of use. Informally, it expresses the intuition that when we want to update an instance, we can as well update only the part of the instance used by the update, and add the part not used afterwards. Formally:

Definition 4.1 Let $\mathcal{X}$ be a set of schema items. A method M is said to *use only information of type* $\mathcal{X}$ if for any instance I and receiver t over I,

$$M(I,t) = \mathcal{I}\big(M(I|_{\mathcal{X}},t) \cup (I - I|_{\mathcal{X}})\big).$$

We still have to introduce the notion of *schema coloring* formally: a function κ assigning to each schema item a subset of $\{\mathbf{u},\mathbf{c},\mathbf{d}\}$. For some schema item X, if $\kappa(X)$ contains **u** then we say that X is colored **u** by κ (and similarly for the other colors). Note that we can extend the subset ordering to colorings in the canonical way.

By the following theorem, we can associate to each update method a unique coloring which describes its behavior. The proof is sketched in the Appendix.

Theorem 4.2 *For each method M there exists a unique minimal coloring such that the following conditions are satisfied:*

1. *If M creates information of type X then X is colored* **c**;
2. *If M deletes information of type X then X is colored* **d**;
3. *If $\mathcal{U}$ is the set of items in S colored* **u**, *then M uses only information of type $\mathcal{U}$;*
4. *Each class name in the method signature is colored* **u**;
5. *If an edge in S is colored* **u**, *then so are its incident nodes.*

The last item guarantees that for any instance I, $I|_{\mathcal{U}}$ is an instance so that item 3 makes sense (the condition in Definition 4.1 makes sense only if $I|_{\mathcal{X}}$ is an instance).

The unique coloring associated to a method M by Theorem 4.2 will be referred to simply as *the minimal coloring of* M. It is undecidable whether a given coloring is the minimal coloring of a given method.

A coloring is called *simple* if each item has at most one color. Simple colorings exhibit the following property.

Proposition 4.3 *Let M be an update method. If the minimal coloring of M is simple then M is inflationary, i.e., $I \subseteq M(I,t)$ for each instance I and receiver t over I.*

The proof of Proposition 4.3, which is sketched in the Appendix, is based on a soundness criterion for colorings. A coloring is called *sound* if it is the minimal coloring of some method. Some colorings are not sound; actually, it is possible to characterize the sound colorings syntactically. The interested reader is referred to Proposition A.2 in the Appendix.

We are now in a position to return to our original motivation: the notion of order independence. The colorings describing order-independent updates can be characterized as follows. The proof is sketched in the Appendix.

Theorem 4.4 *Let κ be a sound coloring. All methods having κ as their minimal coloring are order independent, if and only if κ is simple.*

Example 4.5 Recall the schema of Example 3.2. Consider the method of type [Drinker] which adds to the bars frequented by the receiving drinker all those serving a beer he likes. The minimal coloring of this method assigns $\{\mathbf{u}\}$ to the nodes Drinker, Bar, and Beer and the edges labeled 'likes' and 'serves,' and assigns $\{\mathbf{c}\}$ to the edge labeled 'frequents'. This coloring is simple, and the method is indeed inflationary and order independent. ■

As announced earlier, we have also investigated an alternative axiomatization of use, which we present next. Informally, it expresses the intuition that items of information that are needed by the update cannot be removed without changing the result of the update. Formally:

Definition 4.6 Let $\mathcal{X}$ be a set of items in S. A method M is said to *use only information of type* $\mathcal{X}$ if for any instance I, any receiver t over I, and any item x in I whose label is *not* in $\mathcal{X}$, $M(\mathcal{I}(I - \{x\}), t) = \mathcal{I}(M(I,t) - \{x\})$.

Notice how conceptually different the above definition is from our first Definition 4.1. In a sense, the first definition is more global while the second is more local. The two definitions are also formally different. For example, consider the method which deletes all objects of a certain class X. If this method uses only information of type $\mathcal{X}$ according to Definition 4.1, X must be in $\mathcal{X}$, but this is not true under Definition 4.6. On the other hand, consider the method which always adds some fixed object of type X. Now it is according to Definition 4.6 that X must be in $\mathcal{X}$, but no longer under Definition 4.1. In a sense, the two definitions are each other's dual in the way they treat deletion and creation of information.

It turns out that we can repeat the entire development under the new Definition 4.6. We can reprove the verbatim analogs of Theorems 4.2 and 4.4. Curiously, an additional illustration of the duality alluded upon above is that as analog of Proposition 4.3, we can show that simple colorings under the new definition describe *deflationary* rather than inflationary updates (i.e., $M(I,t) \subseteq I$ for all (I,t)). Space limitations prevent us to give the proofs. The new arguments are similar in spirit to the old ones, but often technically different and actually somewhat easier.

5 Algebraic update methods

In this section, we consider a more specific framework of update methods implemented in the relational algebra, inspired by the algebraic model of object-oriented database access introduced by Hull and Su [10].

It is well-known (e.g., [15, 10, 11]) that object-base schemas and instances can be naturally viewed as relational database schemas and instances. Formally, assume that all class names and property names are attribute names. Following the standard convention, we will omit the set braces from relation schemes, writing $\{A, B, C\}$ simply as ABC. Now consider a given object-base schema S. The relational database schema corresponding to S contains for each class name C in S the unary relation scheme C. The domain Δ_C of C is the universe of all objects of type C. Furthermore, for each edge (C, a, B) in S, there is a binary relation scheme Ca. The domain Δ_a of a is Δ_B. As integrity constraints, the schema contains inclusion dependencies $Ca[C] \subseteq C[C]$ and $Ca[a] \subseteq B[B]$ for each edge (C, a, B) in S.

It is clear that the object-base instances of S correspond precisely to the relational database instances of the relational database schema corresponding to S. Henceforth, we will blur the distinction between an object-base schema or instance and its relational representation. We fix a schema S in what follows.

We are now ready to define our algebraic model of update methods. We consider methods which can only update the properties of the receiving object. These updates are performed via a simple assignment statement, the right-hand side of this statement being a relational algebra expression parameterized by the receiver of the method.

Formally, let $\sigma = [C_0, \dots, C_k]$ be a method signature, and let a be a property of the receiving class C_0. An *algebraic update statement on a of type σ* is an expression of the form $a := E$, where E is a relational algebra expression over the relation schemes in S and the special unary relation schemes *self* and arg_i for $1 \leq i \leq k$. The result scheme of E must be unary.

An *algebraic update method* of type σ is a set of algebraic update statements of type σ containing at most one update on each property of the receiving class. The result of applying an algebraic update method M to an instance I and a receiver $t = (o_0, \dots, o_k)$ over I is defined in the obvious way. For each statement $a := E$, the expression E is evaluated on I, where the special relation *self* is interpreted as the singleton

$\{o_0\}$ containing the receiving object, and where arg_i is interpreted as the ith argument $\{o_i\}$, for $1 \leq i \leq k$. Call the result of this evaluation $E(I,t)$. The updated instance $M(I,t)$ is then obtained from I by replacing all edges labeled a leaving the receiving object by edges to all elements of $E(I,t)$.

Example 5.1 In writing algebraic methods, we will abbreviate class and property names by their first letter (Bar and Beer are abbreviated as Ba and Be). The method *favorite_bar* of Example 3.2 can be implemented simply as $f := arg_1$, and the method *add_bar* as $f := \pi_f(self \underset{self=D}{\bowtie} Df) \cup \rho_{arg_1 \to f}(arg_1)$. The method of Example 4.5 can be implemented as

$$f := \pi_f(self \underset{self=D}{\bowtie} Df) \cup \rho_{Ba\to f}\pi_{Ba}(self \underset{self=D}{\bowtie} Dl \underset{l=s}{\bowtie} Bas).$$

In practice, syntactic sugar such as dot notations and path expressions can be used to write algebraic update methods more easily. ■

In order for $M(I,t)$ to be a well-defined instance of S, each statement $a := E$ in M must respect the integrity constraints of S. More precisely, if a is of type B, then for any instance I and receiver t, $E(I,t) \subseteq B(I)$. We say in this case that E is *typed*. Typedness is undecidable,[1] but it is possible to define a syntactic "strongly typed" subclass of the typed expressions and show that on object-base instances, any typed expression is equivalent to some strongly typed one [6]. Hence, well-definedness does not pose a problem in practice.

Let us now turn to the issue of order independence of algebraic methods. Our main result of this section is the following. The proof is sketched in the Appendix.

Theorem 5.2 *The problem of deciding equivalence between relational algebra expressions is reducible to the problem of deciding order independence of algebraic methods. Conversely, method order independence is reducible to expression equivalence under functional dependencies.*

The functional dependencies appear because the special relations *self* and arg_i occurring in the expressions in a method body are singletons; this is captured using functional dependencies.

We can also prove a version of Theorem 5.2 for key-order independence (omitted). Consequently, both order independence, key-order independence, as well as query-order independence (with a relational algebra query part of the input) of algebraic methods are all undecidable. In the remainder of this section we will present a sufficient condition for key-order independence in the general case, followed by a decidability result in the special case of "positive" methods.

[1]By reduction from the satisfiability problem and observing that arbitrary relational algebra expressions can be simulated using expressions over object-base instances.

Proposition 5.3 *Let M be an algebraic method such that each expression in its body does not access the relations corresponding to the properties updated by M. Then M is key-order independent.*

The above proposition, trivial as it may be, explains why the **update** ... **set** ... **where** ... command of SQL has a well-defined semantics. The **where** clause selects from the table a subset of "receiving" tuples to be updated. The expressions in the **set** clause (which can be subqueries) are evaluated over the current instance, effectively yielding a key set of receivers providing for each receiving tuple the arguments for the update, which can then be performed by a simple run through the set without accessing any more information.

Example 5.4 The method *favorite_bar* satisfies the condition of Proposition 5.3 (see Example 5.1) and is indeed key-order independent. Observe that *add_bar* does not satisfy the condition but is still order independent; this shows that the condition is only sufficient. ■

An important special kind of algebraic method is the *positive* method, which uses only positive expressions, i.e., expressions not containing the difference operator. Selection for non-equality ($\sigma_{\neq}$) is still allowed, however. Note that positive methods can still delete information:

Example 5.5 The method *delete_bar* of type $[D, Ba]$ which deletes the argument bar from those frequented by the receiving drinker is positive, as it can be implemented as

$$f := \pi_f(self \underset{self=D}{\bowtie} Df \underset{f\neq arg}{\bowtie} arg). \quad ■$$

Our main positive result of this section is the following:

Theorem 5.6 *Well-definedness, order independence, and key-order independence of positive algebraic methods are decidable.*

The proof (sketched in the Appendix) is based on the decidability of containment of positive relational algebra expressions over object-base instances under functional dependencies by combining classical techniques from relational database theory.[2] Decidability of well-definedness then immediately follows; for order independence, we must additionally observe that in the proof of Theorem 5.2 the reduction of method order independence to expression equivalence preserves positivity. By the same techniques, it is decidable whether a positive query Q always returns a key set of receiver tuples, which can be useful to test for Q-order independence in case we already know that M is key-order indenpedent.

It remains open whether the following problem is decidable: *given a positive query Q and a positive method M, is M Q-order independent?* The reason why our techniques fail to solve this problem is that they crucially rely on Lemma 3.3, which fails for query-order independence. The interested reader is referred to Example A.3 in the Appendix for counterexamples.

6 Parallel application

In this section, remaining in the algebraic framework, we study an alternative, "parallel" way of applying an update method to a set of receivers.

Let E be an expression occurring as the right-hand side of an assignment statement in the algebraic update method M. The expression E can access the different components of the receiver using the special unary singleton relations *self* and arg_i. However, suppose we prefer to store the entire receiver in one single relation *rec* over the scheme $self\, arg_1 \ldots arg_k$. This is equivalent; it suffices to substitute in E '*self*' by '$\pi_{self}(rec)$' and 'arg_i' by '$\pi_{arg_i}(rec)$.'

Using this relation *rec* suggests a natural semantics for applying the method to a set of receivers: we instantiate *rec* not by a single receiver but by the whole set at once. However, in order to do so in a sensible way, we must take care that arguments belonging to different receiving objects are not mixed up. Thereto, we keep a copy of the receiving object *self* throughout the evaluation of the expression. So, the simple substitutions described in the previous paragraph will not do; instead we modify E as follows:

- Each relation name R is replaced by $\pi_{self}(rec) \times R$;
- *self* is replaced by $\pi_{self}(rec)$, and each arg_i is replaced by $\pi_{self,arg_i}(rec)$;
- each projection $\pi_{A_1,\ldots,A_p}$ is replaced by $\pi_{self,A_1,\ldots,A_p}$;
- each Cartesian product or join is modified so as to join (also) on the new common attribute *self*.

Denote the resulting expression by $\widetilde{E}$. Note that the result scheme of $\widetilde{E}$ is that of E to which the attribute *self* is added.

The result of applying M in parallel to an instance I and a set T of receivers over I is now defined in the obvious way. For each statement $a := E$, the expression $\widetilde{E}$ is evaluated on I, where the special relation *rec* is interpreted by T. Call the result of this evaluation $\widetilde{E}(I,t)$. The updated instance $M_{\mathrm{par}}(I,T)$ is then obtained from I by replacing for each receiving object o_0 occurring in T all edges labeled a leaving o_0 by edges to all objects linked to o_0 in $\widetilde{E}(I,T)$. Note that if T is a singleton $\{t\}$ then $M_{\mathrm{par}}(I,\{t\}) = M(I,t)$.

Example 6.1 Consider the scheme consisting of one class name C and two edges labeled e and tc. Let M be the method of type $[C,C]$ having the single statement

$$tc := \pi_e(self \underset{self=C}{\bowtie} Ce) \cup \pi_e(self \underset{self=C}{\bowtie} Ctc \underset{tc=C}{\bowtie} Ce).$$

This method is order independent. Let I be an instance containing only e-edges, and let T be the set of receivers $C \times C$. Then the sequential application $M_{\mathrm{seq}}(I,T)$ computes the transitive closure of I in the tc-edges, while the parallel application $M_{\mathrm{par}}(I,T)$ simply duplicates each e-edge with a tc-edge. Indeed, $\widetilde{E}$ (E being the expression assigned to tc) equals

$$\begin{aligned}\pi_{self,e}(\pi_{self}(rec) &\underset{\substack{self=self\\ self=C}}{\bowtie} (\pi_{self}(rec) \times Ce))\\ \cup\, \pi_{self,e}(\pi_{self}(rec) &\underset{\substack{self=self\\ self=C}}{\bowtie} (\pi_{self}(rec) \times Ctc)\\ &\underset{\substack{self=self\\ tc=C}}{\bowtie} (\pi_{self}(rec) \times Ce)),\end{aligned}$$

which on an instance without tc-edges is equivalent to $\pi_{self,e}(\pi_{self}(rec) \underset{self=C}{\bowtie} Ce)$. ■

The above example shows that parallel application is less powerful than sequential application, since sequential application can express transitive closure while parallel application, by definition, does not have more power than the relational algebra. (One can also express parity using sequential application.)

When we restrict attention to key sets of receivers, however, parallel and sequential application are equivalent. The proof of this last result is sketched in the Appendix.

Proposition 6.2 *If M is key-order independent, then $M_{\mathrm{seq}}(I,T) = M_{\mathrm{par}}(I,T)$ for any instance I and key set of receivers T.*

[2] A similar result, supporting only a weak form of union but allowing a weak form of negation, was presented by Chan [7]. We believe that our approach based on classical database theory techniques sheds new light on Chan's results, which were proven using ad-hoc techniques.

References

[1] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(2):181–229, 1990.

[2] A.V. Aho, Y. Sagiv, and J.D. Ullman. Equivalences among relational expressions. *SIAM Journal on Computing*, 8(2):218–246, 1979.

[3] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[4] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In P. Kanellakis and J.W. Schmidt, editors, *Database Programming Languages: Bulk Types and Persistent Data*, pages 9–19. Morgan Kaufmann, 1992.

[5] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *Automata, Languages, and Programming*, Lecture Notes in Computer Science 510, pages 60–75, 1991.

[6] L. Cabibbo and J. Van den Bussche. Relative expressiveness of typed and untyped relational algebra. Unpublished manuscript.

[7] E.P.F. Chan. Containment and minimization of positive conjunctive queries in OODB's. In *Proceedings 11th ACM Symposium on Principles of Database Systems*, pages 202–211, 1992.

[8] A. Chandra. Programming primitives for database languages. In *Proceedings 8th ACM Symposium on Principles of Programming Languages*, pages 50–62, 1981.

[9] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings 9th ACM Symposium on Theory of Computing*, pages 77–90. ACM, 1977.

[10] R. Hull and J. Su. On accessing object-oriented databases: Expressive power, complexity, and restrictions. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, volume 18:2 of *SIGMOD Record*, pages 147–158. ACM Press, 1989.

[11] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 455–468. Morgan Kaufmann, 1990.

[12] D.S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences*, 28:167–189, 1984.

[13] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.

[14] C. Laasch and M.H. Scholl. Deterministic semantics of set-oriented update sequences. In *Proceedings, Ninth International Conference on Data Engineering*, pages 4–13. IEEE Computer Society Press, 1993.

[15] P. Lyngbaek and V. Vianu. Mapping a semantic database model to the relational model. In U. Dayal and I. Traiger, editors, *Proceedings of the ACM SIGMOD 1987 Annual Conference*, volume 16:3 of *SIGMOD Record*, pages 132–142. ACM Press, 1987.

[16] D. Maier, A.O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4:455–469, 1979.

[17] X. Qian. The expressive power of the bounded-iteration construct. *Acta Informatica*, 28:631–656, 1991.

[18] Y. Sagiv and M. Yannakakis. Equivalence among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.

[19] J.T. Schwartz et al. *Programming with sets: an introduction to SETL*. Springer-Verlag, 1986.

[20] E. Simon and C. de Maindreville. Deciding whether a production rule is relational computable. In M. Gyssens, J. Paredaens, and D. Van Gucht, editors, *ICDT'88*, volume 326 of *Lecture Notes in Computer Science*, pages 205–222. Springer-Verlag, 1988.

A Appendix

Proof of Theorem 4.2. (Sketch) Note that the full coloring, which assigns all colors to all labels, satisfies the conditions of the theorem. Note also that the lattice of subsets of the colors $\{\mathbf{u}, \mathbf{c}, \mathbf{d}\}$ can be canonically extended to a lattice of colorings. Hence, it suffices to show that if κ_1 and κ_2 are colorings satisfying the conditions in Theorem 4.2, then so is $\kappa_1 \cap \kappa_2$.

Thereto, put $\kappa = \kappa_1 \cap \kappa_2$. For $i = 1, 2$, let $\mathcal{U}_i$ be the set of items in S colored $\mathbf{u}$ by κ_i, and let $\mathcal{U}$ be the set of items colored $\mathbf{u}$ by κ. Note that for any instance I,

$(I|_{\mathcal{U}_1})|_{\mathcal{U}_2} = I_{\mathcal{U}}$. Since κ_1 and κ_2 satisfy condition 3 of Theorem 4.2, we have

$$\begin{aligned} M(I,t) &= \mathcal{I}\big(M(I|_{\mathcal{U}_1},t) \cup (I - I|_{\mathcal{U}_1})\big) & (1)\\ &= \mathcal{I}\big(M(I|_{\mathcal{U}_2},t) \cup (I - I|_{\mathcal{U}_2})\big). & (2)\end{aligned}$$

We must prove that κ satisfies the conditions in Theorem 4.2; in this sketch, we will concentrate on the proof of condition 3:

$$M(I,t) = \mathcal{I}\big(M(I|_{\mathcal{U}},t) \cup (I - I|_{\mathcal{U}})\big).$$

By applying equations (1) and (2) in succession, we obtain

$$\begin{aligned} M(I,t) &= \mathcal{I}(M(I|_{\mathcal{U}_1},t) \cup (I - I|_{\mathcal{U}_1}))\\ &= \mathcal{I}(\mathcal{I}(M((I|_{\mathcal{U}_1})|_{\mathcal{U}_2},t) \cup (I|_{\mathcal{U}_1} - (I|_{\mathcal{U}_1})|_{\mathcal{U}_2}))\\ &\quad \cup (I - I|_{\mathcal{U}_1}))\\ &= \mathcal{I}(\mathcal{I}(M(I|_{\mathcal{U}},t) \cup (I|_{\mathcal{U}_1} - I|_{\mathcal{U}}))\\ &\quad \cup (I - I|_{\mathcal{U}_1})).\end{aligned}$$

To prove that the graph denoted by the last expression above equals $\mathcal{I}(M(I|_{\mathcal{U}},t) \cup (I - I|_{\mathcal{U}}))$, we will consider the nodes and edges separately. For the nodes, the equality follows readily once we observe that the nodes in $(I|_{\mathcal{U}_1} - I|_{\mathcal{U}}) \cup (I - I|_{\mathcal{U}_1})$ are precisely those of $I - I|_{\mathcal{U}}$. For the edges, a straightforward calculation shows that the crux of the proof is to establish the following equivalence: an edge e together with its incident nodes n and m belongs to $\mathcal{I}(M(I|_{\mathcal{U}},t) \cup (I|_{\mathcal{U}_1} - I|_{\mathcal{U}})) \cup (I - I|_{\mathcal{U}_1})$ if and only if e, n and m belong simply to $M(I|_{\mathcal{U}},t) \cup (I|_{\mathcal{U}_1} - I|_{\mathcal{U}}) \cup (I - I|_{\mathcal{U}_1})$.

The only-if direction of this equivalence is trivial. To show the if-direction, note that we can concentrate on the case $e \in (I|_{\mathcal{U}_1} - I|_{\mathcal{U}})$, the other cases being trivial ($M(I|_{\mathcal{U}},t)$ is already an instance, so the operator $\mathcal{I}$ has no effect on it, and $(I - I|_{\mathcal{U}_1})$ is outside the scope of the application of $\mathcal{I}$ altogether). In particular, then, $e \in I|_{\mathcal{U}_1}$ and hence $n, m \in I|_{\mathcal{U}_1}$ since κ_1 satisfies condition 5 of Theorem 4.2. As a consequence, n and m are not in $(I - I|_{\mathcal{U}_1})$ and hence must belong to $M(I|_{\mathcal{U}},t) \cup (I_{\mathcal{U}_1} - I|_{\mathcal{U}})$. We can therefore conclude that e, n and m belong to $\mathcal{I}(M(I|_{\mathcal{U}},t) \cup (I_{\mathcal{U}_1} - I|_{\mathcal{U}}))$, as had to be shown. ■

Proof of Proposition 4.3. (Sketch) Let κ be the minimal coloring of method M. We will prove the following technical lemma:

Lemma A.1 *If a node in the schema is colored* **d** *by* κ, *then it is also colored* **u**. *If an edge is colored* **d** *by* κ, *then either it is also colored* **u** *or one of its incident nodes is colored* **d**.

The above lemma implies the proposition to be proven: if κ is simple, it cannot not color anything **d** and hence M will never delete any information, i.e., M is inflationary.

Let $\mathcal{U}$ be the set of items in S colored **u**. The proof of the first statement is straightforward: if X is a node in the schema colored **d**, then there exists an instance I and a receiver t such that I contains an object n of type X that is not in $M(I,t)$. If X would not be colored **u**, then n would be in $I - I|_{\mathcal{U}}$ and hence in $\mathcal{I}(M(I|_{\mathcal{U}},t) \cup (I - I|_{\mathcal{U}}))$, which equals $M(I,t)$ by condition 3 of Theorem 4.2; a contradiction.

To prove the second statement, assume there is an edge in S, labeled e, which is colored **d** but not **u**. Then there exists an instance I and a receiver t such that I contains an edge $x = (n,e,m)$ not in $M(I,t)$. Since e is assumed not in $\mathcal{U}$, x is in $I - I|_{\mathcal{U}}$ and hence in $M(I|_{\mathcal{U}},t) \cup (I - I|_{\mathcal{U}})$. We must show that at least one of the labels $\lambda(n)$ and $\lambda(m)$ of n and m is colored **d**. Assume the contrary. We consider two possibilities for object n:

1. If $\lambda(n)$ is colored **u**, n is in $I|_{\mathcal{U}}$. Since $\lambda(n)$ is assumed not colored **d**, n is in $M(I|_{\mathcal{U}},t)$ and hence in $M(I|_{\mathcal{U}},t) \cup (I - I|_{\mathcal{U}})$.
2. If $\lambda(n)$ is not colored **u**, then n is in $I - I|_{\mathcal{U}}$ and hence in $M(I|_{\mathcal{U}},t) \cup (I - I|_{\mathcal{U}})$.

The same case analysis applies to the object m. Consequently, both of the nodes incident to edge x are in $M(I|_{\mathcal{U}},t) \cup (I - I|_{\mathcal{U}})$ and thus x is in $\mathcal{I}(M(I|_{\mathcal{U}},t) \cup (I - I|_{\mathcal{U}}))$, which equals $M(I,t)$; a contradiction. ■

Proposition A.2 *A coloring is sound if and only if it has the following properties:*

1. *The property expressed by Lemma A.1;*
2. *If an edge is colored* **c**, *then its incident nodes are colored* **u** *or* **c**;
3. *At least one node is colored* **u**;
4. *If an edge is colored* **u**, *then so are its incident nodes.*

Proof. (Sketch) We concentrate on the if-direction. Let κ be a sound coloring. We can construct an update method having κ as its minimal coloring as follows. The signature of the method may be arbitrarily fixed as long as all its elements are colored **u**. Regardless of the particular receiver to which it is applied, the update performed by the method is the following:

1. For each node X in the schema with $\kappa(X) = \{\mathbf{c}\}$, some fixed object of type X is added;
2. For each edge X in the schema with $\kappa(X) = \{\mathbf{c}\}$ or $\{\mathbf{c}, \mathbf{d}\}$, some fixed edge e with label X is added. If one of the incident nodes of X in the schema is not colored **c**, then e is added only if the corresponding incident node of e is already present;

3. For each schema item X with $\kappa(X) = \{\mathbf{u}\}$, the update is undefined on instances that do not contain some fixed item with label X;

4. For each schema item X with $\kappa(X) = \{\mathbf{u}, \mathbf{d}\}$, some fixed item with label X is deleted;

5. For each node X in the schema with $\kappa(X) = \{\mathbf{u}, \mathbf{c}\}$, a fixed object of type X is added on condition that it is not already present; otherwise, some other fixed object is added;

6. For each edge X in the schema with $\kappa(X) = \{\mathbf{u}, \mathbf{c}\}$, the behavior is similar to that described in item 2, with the exception that the fixed edge is added only if it is not already present; otherwise, some other fixed edge is added;

7. Finally, for each schema item X with $\kappa(X) = \{\mathbf{u}, \mathbf{c}, \mathbf{d}\}$, the behavior combines those described for $\kappa(X) = \{\mathbf{u}, \mathbf{d}\}$ and $\kappa(X) = \{\mathbf{u}, \mathbf{c}\}$. ■

Proof of Theorem 4.4. (Sketch) *If:* By Proposition 4.3, any method having κ as minimal coloring is inflationary. Furthermore, for any such method M and for any instance I and receiver t, we have $I|_{\mathcal{U}} = M(I,t)|_{\mathcal{U}}$. Indeed, since M is inflationary, $I|_{\mathcal{U}} \subseteq M(I,t)|_{\mathcal{U}}$, and this inclusion cannot be strict since any information in $M(I,t)$ not in I is colored $\mathbf{c}$ and thus not $\mathbf{u}$ because κ is simple. Using these two observations, it can be verified that $M(M(I,t_1),t_2) = M(I,t_1) \cup M(I,t_2) = M(M(I,t_2),t_1)$ for any pair $\{t_1,t_2\}$ of receivers. Hence, by Lemma 3.3, M is sequential-deterministic.

Only if: Assume κ is not simple; then the soundness of κ can be used to deduce that at least there is a node R colored (1) $\{\mathbf{u}, \mathbf{d}\}$, (2) $\{\mathbf{u}, \mathbf{c}, \mathbf{d}\}$, or (3) $\{\mathbf{u}, \mathbf{c}\}$, or an edge (R, a, A) colored (4) $\{\mathbf{u}, \mathbf{d}\}$, (5) $\{\mathbf{u}, \mathbf{c}, \mathbf{d}\}$, or (6) $\{\mathbf{u}, \mathbf{c}\}$. For each of these cases we will give a method of type $[R, A]$ which is not sequential-deterministic, having κ as its minimal coloring. Since κ is sound, we can start with the method associated to κ according to the proof of Proposition A.2. We then adapt this method to one of the six possible cases as follows:

1. If there are exactly two objects of type R, delete the receiving object.

2. As in the previous case, but if the test fails add two new objects to class R.

3. If there are not exactly two objects of type R, do nothing. Otherwise, if the receiving object is equal to some fixed object, add two new objects to class R; otherwise, add only one.

4. If there is an edge with label a between receiving and argument object, delete all other a-edges.

5. As in the previous case, but if the test fails add an a-edge between receiving and argument object and delete all other a-edges.

6. If there are no a-edges, add one between receiving and argument object. ■

Proof of Theorem 5.2. (Sketch) We first reduce expression equivalence to method order independence. Let S be an object-base schema, and let E_1 and E_2 be two expressions over S (E_1 and E_2 are assumed to have the empty result scheme, w.l.o.g.) Augment S with a class name C having two properties a and b of type C. The following update method of type $[C]$ is order independent if and only if E_1 and E_2 are equivalent:

```
$a := \emptyset$;
$b :=$ **if** $Ca = C \times \rho_{C\to a}(C)$
    **then if** $E_1$ **then** *self* **else** $\emptyset$
    **else if** $E_2$ **then** *self* **else** $\emptyset$.
```

It is well-known how if-then-else constructs can be simulated in the relational algebra.

We next reduce method order independence to expression equivalence. Let M be an update method with receiving class C, containing update assignments $a := E_a$, for each $a \in \mathcal{A}$ where $\mathcal{A}$ is some set of properties of C. If I is an instance and the unary singleton relations *self*, arg_1, ..., arg_k together hold a receiver t, then the relation Ca in the instance $M(I,t)$ can be expressed as

$$\pi_{C,a}(Ca \underset{C \neq self}{\bowtie} self) \cup \rho_{self\to C}(self) \times E_a.$$

Denote this expression by $E_a[t]$. Now denote by E'_a the expression obtained from $E_a[t]$ by replacing each occurrence of Cb, where $b \in \mathcal{A}$, by $E_b[t]$, and let *self'*, arg'_1, ..., arg'_k together hold a second receiver t'. Then the relation Ca in the instance $M(I, tt')$ can be expressed as

$$\pi_{C,a}(E_a[t] \underset{C \neq self'}{\bowtie} self') \cup \rho_{self'\to C}(self') \times E'_a.$$

Call this expression $E_a[tt']$. By reversing the process, we obtain an expression $E_a[t't]$. By Lemma 3.3, M is order independent iff for each $a \in \mathcal{A}$, the expressions $E_a[tt']$ and $E_a[t't]$ are equivalent. However, in testing the equivalence, care must be taken so as to consider only interpretations of the relations *self*, *self'*, $arg_1, \ldots, arg'_k$ which assign them *(i)* at most one element; *(ii)* at least one element; and *(iii)* different receivers t and t'. Requirement *(i)* is dealt with by imposing functional dependencies of the form $\emptyset \to self$; requirements *(ii)* and *(iii)* are dealt with by modifying the expressions so as to yield the empty result if they are not satisfied. ■

Proof of Theorem 5.6. (Sketch) Recall the reduction of method order independence to expression equivalence shown in the proof of Theorem 5.2. In this reduction, if the method to be checked for order independence is positive, then so are the expressions to be checked for equivalence. Our positive expressions can be viewed as conjunctive queries extended with union and non-equality. Testing for containment (whence equivalence) of conjunctive queries is well-known to be decidable [9, 2], and extensions incorporating union [18] or selection for inequalities [13] are equally well-known. These two generalizations can be combined, and it can be verified that the techniques for dealing with inequalities ($\leq$) carry over to non-equalities ($\neq$). Finally, we must keep in mind that our expressions have to be checked for equivalence only on object-base instances. Such instances satisfy the inclusion dependencies specified by the schema, as well as exclusion dependencies implied by the disjointness of classes. Furthermore, we must take into account the functional dependencies mentioned in Theorem 5.2. The functional and inclusion dependencies can be dealt with by a standard chase process [16, 2, 12] (the inclusion dependencies are unary and full and hence do not pose a problem in this respect). The exclusion dependencies are dealt with by employing a typed chase. It can be verified that this process works together with the extensions to deal with union and non-equality. ■

Example A.3 We will give counterexamples *dis*proving the following statement: let Q be a positive algebra query and M a positive algebraic method. Then M is Q-order independent if and only if M is order independent on any pair (I, T) where T is a two-element subset of $Q(I)$. Consider a scheme with a class name C having two properties a and b of type C.

For the if-direction, M (of type $[C, C]$) is the method deleting the argument object from the a-properties of the receiving object on condition that relation Ca contains at least two tuples; query Q equals **if** $\#Ca > 2$ **then** Cb **else** $\emptyset$.

For the only-if direction, M (of type $[C, C, C]$) assigns to a all b-properties of the receiving object, and adds the first argument object to the b-properties (the second argument object is not used). Query Q returns the three-fold Cartesian product of C with itself. ■

Proof of Proposition 6.2. (Sketch) Let E be an expression occurring as the right-hand side of some statement in M. Using the fact that T is a key set, one can show by induction on the structure of E that

$$\widetilde{E}(I, T) = \bigcup_{t \in T} t(\mathit{self}) \times E(I, t).$$

Moreover, if $T = \{t_1, \ldots, t_n\}$, one can show that $E(M(I, t_1 \ldots t_i), t_{i+1}) = E(I, t_{i+1})$. The intuitive reason is that since M is key-order independent, the outcome of the sequential application must be the same regardless of whether or not one starts with t_{i+1} as the first receiver. The formal proof is notationally rather intricate and tedious. These two properties immediately imply $M_{\text{par}}(I, T) = M_{\text{seq}}(I, T)$. ■