Applying an Update Method to a Set of Receivers

MARC ANDRIES
University of Antwerp
LUCA CABIBBO
Università di Roma Tre
JAN PAREDAENS
University of Antwerp
and
JAN VAN DEN BUSSCHE
Limburgs Universitair Centrum

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 that 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 update methods and compare and relate it to the sequential strategy.

Categories and Subject Descriptors: H.2 [**Information Systems**]: Database Management

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words and Phrases: Database update, order independence, parallel update, relational algebra, schema coloring

2   •   M. Andries et al.

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 article 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 that 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 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 [1989]. Methods in this framework can only update properties of the receiving objects. We observe that order independence of algebraic updates 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 update 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 that explains many practical cases.

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, which normally stands for a single receiver, by the whole set of receivers at once. In this approach, order independence is automatically guaranteed. Hence, it is interesting to ask whether every order-independent algebraic update 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 updates 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 [1993] 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 [1979] considered sequential and parallel execution strategies for looping constructs of the form **for each** t **in** R **do** in database manipulation languages that are closely analogous to the sequential and parallel strategies we consider in the present article. 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 [1981] 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 nondeterministic construct, e.g., in the work of Qian [1991] or in the language SETL [Schwartz et al. 1986]. 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 nondeterministic rule-triggering system introduced by Simon and de Maindreville [1988].

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 [1990] 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 [Breazu-Tannen and Subrahmanyam 1991; Breazu-Tannen et al. 1992] 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 $\bigcap_i D_i \cup \bigcup_i (D_i - D)$.

After this introduction, the remainder of our article is organized as follows. In Section 2, we introduce the simple object database model we will be working in and define the concept of update method in this context. In Section 3, we introduce the notion of sequential application and the associated notions of order independence. Section 4 contains the axiomatic framework for studying update behavior and order independence, and Section 5 contains the algebraic framework. In Section 6 we discuss parallel application. We conclude the paper with a discussion of the practical ramifications of our results.

2. UPDATE METHODS ON OBJECT DATABASES

In this section we present the basic definitions concerning databases and update methods.

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:[1]

Definition 2.1. An *object-base schema* is 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:

Definition 2.2. An *instance* of S is 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.

---

[1]Many of our results also hold for a more involved object data models featuring inheritance and a distinction between single- and multivalued properties [Cabibbo 1996].
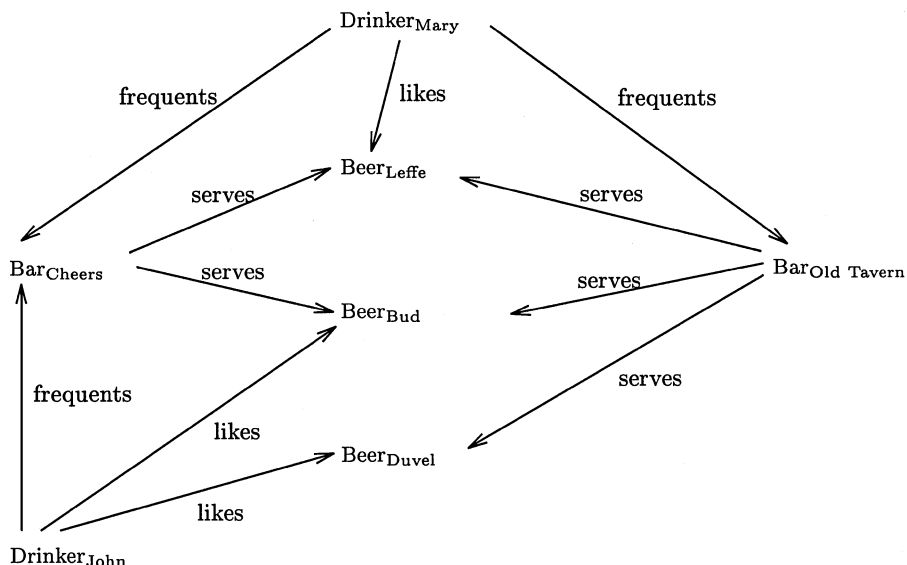
Fig. 1.   An object-base instance.

The set of all objects in an instance labeled by the same class name C is called *the class C*.

Example 2.3.   We use Ullman's well-known 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. An instance of this schema is shown in Figure 1. In this and subsequent figures, objects of some type C are denoted as C_1, C_2, and so on.

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:

Definition 2.4.   A *method signature* σ over schema S is a nonempty 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*.

Definition 2.5.   Given a method signature $\sigma = [C_0, \ldots, C_k]$ over S and an instance I of S, a *receiver* over I of type σ is a tuple of the form $[o_0, \ldots, o_k]$, where $o_0, \ldots, o_k$ are objects in I of types $C_0, \ldots, C_k$, respectively. The first object o_0 is called the *receiving object*; the remaining tuple $o_1, \ldots, o_k$ comprises the *arguments* of the receiver.

Definition 2.6.   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.
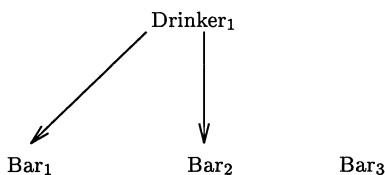
Drinker$_1$

Bar$_1$ Bar$_2$ Bar$_3$

Fig. 2.   Example instance I.

Drinker$_1$

Bar$_1$ Bar$_2$ Bar$_3$
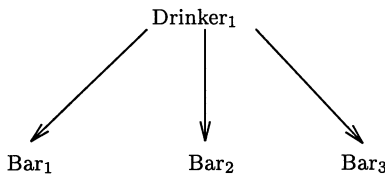
Fig. 3.   The instance $add\_bar(I, [\text{Drinker}_1, \text{Bar}_3])$, where update method add_bar is defined in Example 2.7 and instance I is shown in Figure 2.

Example 2.7.   On our example schema, consider the following two updates 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.

To illustrate these update methods, consider the simple instance I in Figure 2, consisting of a single drinker and three bars (two of which are frequented by the drinker). For simplicity, we have left out any beers from this example. Then $add\_bar(I, [\text{Drinker}_1, \text{Bar}_3])$ is shown in Figure 3 and *favorite_bar*$(I, [\text{Drinker}_1, \text{Bar}_1])$ is shown in Figure 4.

3. SEQUENTIAL APPLICATION

In this short 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 an update. In what follows, we fix a schema S, a signature σ over S, and an update 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 $s = t_1, \ldots, t_n$ is a sequence of distinct receivers, $M(I, s)$ equals I if $n = 0$, and equals $M(M(I, t_1), t_2, \ldots, t_n)$ if $n > 0$, 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 that $M(I, s) = M(I, s')$.[2] In this case we define the *sequential*

---

[2]If $M(I, s)$ is undefined for some s, then it must be undefined for every other s'.

Drinker$_1$

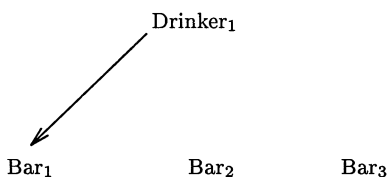Bar$_1$ Bar$_2$ Bar$_3$

Fig. 4.   The instance $favorite\_bar$(I,[Drinker$_1$, Bar$_1$]), where update method $favorite\_bar$ is defined in Example 2.7 and instance I is shown in Figure 2.
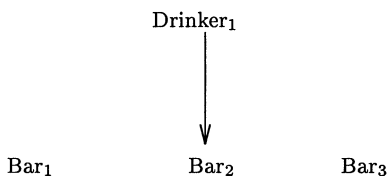
Drinker$_1$

Bar$_1$ Bar$_2$ Bar$_3$

Fig. 5.   The instance $favorite\_bar$(I,[Drinker$_1$, Bar$_1$], [Drinker$_1$, Bar$_2$]), where I is shown in Figure 2.

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:

(1) Absolute order independence: If M is order independent on any pair (I, T) then M is called *order independent*.

(2) Key-order independence: 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*.

(3) Query-order independence: Finally, let Q be a function that 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.   The update add_bar from the previous example is clearly order independent, but $favorite\_bar$ is not. Indeed, continuing Example 2.7,

$$favorite\_bar(I, [\text{Drinker}_1, \text{Bar}_1], [\text{Drinker}_1, \text{Bar}_2])$$

is shown in Figure 5, while

$$favorite\_bar(I, [\text{Drinker}_1, \text{Bar}_2], [\text{Drinker}_1, \text{Bar}_1])$$

equals simply $favorite\_bar(I, [\text{Drinker}_1, \text{Bar}_1])$ already shown in Figure 4.

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 update methods as general computable functions, as we did, all of the notions of order independence defined above are undecidable, by Rice's theorem [Hopcroft and Ullman 1979]. We will later show however that order independence is decidable for more restricted kinds of methods. Thereto,

we rely on the following lemma, which is quite obvious 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.*

This lemma 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 at then end of Section 5.

4. SCHEMA COLORINGS

In this section we present the beginnings of a theory of schema colorings. Such colorings, which could be provided by the programmer or be inferred from the specification, 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 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 that describe order-independent updates.

In what follows, we fix a schema S and a method signature over S.

4.1 Preliminaries

Since schemas and instances are graphs, it is useful to introduce the following terminology:

Definition 4.1. An *item* of a graph is either a node or an edge of that graph.

Consequently, a graph can be identified with the set of its items.

It is not difficult to formalize what it means for an update to create or delete information of a certain type:

Definition 4.2. 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.

Dually, *M deletes information of type X* if there exists and instance I and a receiver t over I such that I contains an item labeled X that is not in $M(I, t)$.

In order to define what it means for an update to use information of a certain type, we introduce the following auxiliary notion:

Definition 4.3. A *partial instance* is a subset of some instance (viewed as the set of its items).

So a partial instance is an instance from which some items have been removed. Partial instances are different from instances, in that they may contain "dangling edges": a node may be removed without removing all its incident edges.

The operator $\mathcal{G}$ eliminates all dangling edges from a partial instance:

Definition 4.4.   Let J be a partial instance. Then $\mathcal{G}(J)$ equals the largest instance contained in J.

Note that, by viewing a partial instance as a set of items, we can apply set-theoretic operations such as union and difference to partial instances.
We also need:

Definition 4.5.   Let $\mathcal{X}$ be a set of schema items and I be 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 $I|_{\mathcal{X}}$.

To end this preliminary section, we introduce the notion of schema coloring formally:

Definition 4.6.   A *coloring* of schema S is a function κ assigning to each item in S a subset of $\{\mathbf{u}, \mathbf{c}, \mathbf{d}\}$.

For some schema item X, if $\kappa(X)$ contains $\mathbf{u}$ then we say that X is colored $\mathbf{u}$ by κ (and similarly for the other colors).
Note that we can compare colorings according to the subset ordering in the canonical way, i.e., $\kappa \subseteq \kappa'$ if $\kappa(X) \subseteq \kappa'(X)$ for all schema items X.

4.2 Inflationary Colorings[3]

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 4.7.   Let M be an update method and let $\mathcal{X}$ be a set of schema items such that if an edge is in $\mathcal{X}$, then so are its incident nodes and such that each class name in M's signature is in $\mathcal{X}$. Then M is said to *use only information of type $\mathcal{X}$* if for any instance I and receiver t over I,

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

The conditions on $\mathcal{X}$ are necessary to guarantee that $I|_{\mathcal{X}}$ is always an instance and that t is in it, so that the expression $M(I|_{\mathcal{X}}, t)$ makes sense.
By the following theorem, we can associate to each update method a unique coloring that describes its behavior.

THEOREM 4.8.   *For each update 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 $\mathbf{c}$.*
(2) *If M deletes information of type X, then X is colored $\mathbf{d}$.*
(3) *If $\mathcal{U}$ is the set of items in S colored $\mathbf{u}$, then M uses only information of type $\mathcal{U}$.*

---

[3]The title of this subsection will become clear later.

10   •   M. Andries et al.

(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.*

PROOF. Note that the "full" coloring that assigns all colors to all items satisfies the conditions of the definition. 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 of the theorem, 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 **u** by κ_i and let $\mathcal{U}$ be the set of items colored **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, we have

$$M(I, t) = \mathcal{G}\big(M\big(I|_{\mathcal{U}_1}, t\big) \cup \big(I - I|_{\mathcal{U}_1}\big)\big) \tag{1}$$

$$= \mathcal{G}\big(M\big(I|_{\mathcal{U}_2}, t\big) \cup \big(I - I|_{\mathcal{U}_2}\big)\big). \tag{2}$$

It is straightforward to check that κ satisfies conditions 1, 2, 4, and 5. We can therefore concentrate on condition 3:

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

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

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

To prove that the graph denoted by the last expression above equals

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

we 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 $I - I|_{\mathcal{U}}$:

$$\begin{aligned} \mathcal{G}\big(\mathcal{G}\big(M(I|_{\mathcal{U}}, t) \cup \big(I|_{\mathcal{U}_1} - I|_{\mathcal{U}}\big)\big) \cup \big(I - I|_{\mathcal{U}_1}\big)\big) \\ = M(I|_{\mathcal{U}}, t) \cup \big(I|_{\mathcal{U}_1} - I|_{\mathcal{U}}\big) \cup \big(I - I|_{\mathcal{U}_1}\big) \\ = M(I|_{\mathcal{U}}, t) \cup (I - I|_{\mathcal{U}}) \\ = \mathcal{G}(M(I|_{\mathcal{U}}, t) \cup (I - I|_{\mathcal{U}})) \end{aligned}$$

For the edges, the crux of the proof is to establish the following equivalence: an edge e together with its incident nodes n and m belong to

$$\mathcal{G}\big(M(I|_{\mathcal{U}}, t) \cup \big(I|_{\mathcal{U}_1} - I|_{\mathcal{U}}\big)\big) \cup \big(I - I|_{\mathcal{U}_1}\big)$$

if and only if e, n, and m simply belong to

$$M(I|_{\mathcal{U}}, t) \cup \big(I|_{\mathcal{U}_1} - I|_{\mathcal{U}}\big) \cup \big(I - I|_{\mathcal{U}_1}\big).$$

Indeed, using this equivalence we can deduce:

$$e \in \mathcal{G}\big(\mathcal{G}(M(I|_{\mathcal{U}}, t) \cup \big(I|_{\mathcal{U}_1} - I|_{\mathcal{U}}\big)\big) \cup \big(I - I|_{\mathcal{U}_1}\big)\big)$$
$$\Leftrightarrow\ e, n, m \in \mathcal{G}\big(M(I|_{\mathcal{U}}, t) \cup \big(I|_{\mathcal{U}_1} - I|_{\mathcal{U}}\big)\big) \cup \big(I - I|_{\mathcal{U}_1}\big)$$
$$\Leftrightarrow\ e, n, m \in M(I|_{\mathcal{U}}, t) \cup \big(I|_{\mathcal{U}_1} - I|_{\mathcal{U}}\big) \cup \big(I - I|_{\mathcal{U}_1}\big)$$
$$\Leftrightarrow\ e, n, m \in M(I|_{\mathcal{U}}, t) \cup (I - I|_{\mathcal{U}})$$
$$\Leftrightarrow\ e \in \mathcal{G}(M(I|_{\mathcal{U}}, t) \cup (I - I|_{\mathcal{U}})).$$

To prove the needed equivalence, the only-if direction of this equivalence is trivial. To show the if-direction, we can concentrate on the case $e \in (I|_{\mathcal{U}_1} - I|_{\mathcal{U}})$. Indeed, the other cases are trivial: $M(I|_{\mathcal{U}}, t)$ is already an instance, so the operator $\mathcal{G}$ has no effect on it, and $(I - I|_{\mathcal{U}_1})$ is outside the scope of the application of $\mathcal{G}$ altogether. In particular, $e \in I|_{\mathcal{U}_1}$ and hence $n, m \in I|_{\mathcal{U}_1}$ since κ_1 satisfies condition 5. 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 $\mathcal{G}(M(I|_{\mathcal{U}}, t) \cup (I_{\mathcal{U}_1} - I|_{\mathcal{U}}))$, as had to be shown.

In conclusion, the intersection of all colorings of S satisfying the conditions of the theorem is the unique minimal coloring stated in the theorem. □

The unique coloring associated to an update M by Theorem 4.8 is simply referred as *the minimal coloring of M*. The minimal coloring is clearly a semantic property of an update; it is undecidable whether a given coloring is the minimal coloring of a given method.

A consequence of our axiomatization of "use" is that updates whose minimal coloring is "simple" are inflationary.[4] More precisely, we have the following definition and proposition:

Definition 4.9.   A coloring is called *simple* if each item has at most one color.

PROPOSITION 4.10.   *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.*

PROOF.   Let the minimal coloring of M be κ. We prove the following technical lemma:

LEMMA 4.11.   *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**.*

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

To establish the truth of the lemma, 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 the minimality of κ implies the existence of 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

---

[4]Whence the title of the present subsection.

X cannot will be in $I - I|_{\mathcal{U}}$, and hence in $\mathcal{G}(M(I|_{\mathcal{U}}, t) \cup (I - I|_{\mathcal{U}}))$, which equals $M(I, t)$ by condition 3 of Theorem 4.8; a contradiction.

To prove the second statement, assume there is an edge in S, labeled e, that 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; then n and m belong to $M(I, t)$ and thus to $M(I|_{\mathcal{U}}, t) \cup (I - I|_{\mathcal{U}})$. Hence, x is in $\mathcal{G}(M(I|_{\mathcal{U}}, t) \cup (I - I|_{\mathcal{U}}) = M(I, t)$; a contradiction. □

The intuition behind Lemma 4.11 is that by deleting a node, we have implicitly used it as well; this implication is a logical consequence of the way we axiomatized "use" in Definition 4.7. The same implication holds for edges, except that for edges there is one exception in which we can delete an edge without using it: if a node is deleted, then of course its incident edges must be deleted as well (otherwise the result would not be a proper graph), and such "automatic" deletions of edges are not considered uses by Definition 4.7.

Lemma 4.11 specifies in fact a necessary condition for a coloring to be "sound" in the following sense:

Definition 4.12.   A coloring is called *sound* if it is the minimal coloring of some update method.

It is then natural to ask what exactly are the conditions for a coloring to be sound. We can prove the following characterization:

PROPOSITION 4.13.   *A coloring is sound if and only if it has the following properties:*

(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**.

(2) *If an edge is colored* **c**, *then its incident nodes are colored* **u** *or* **c**.

(3) *If a node B is colored* **d**, *then, for each incident edge* (B, e, C) *or* (C, e, B) *in the schema that is neither colored* **d** *nor* **u**, *C is colored* **u**.

(4) *At least one node is colored* **u**.

(5) *If an edge is colored* **u**, *then so are its incident nodes.*

PROOF.   For the only-if direction, consider the minimal coloring of some update method M. Property 1 has already been proven in Lemma 4.11; properties 4 and 5 are clear.

To see property 2, consider a schema edge (A, e, B) colored **c**. Then there exists an instance I and a receiver t such that $M(I, t)$ contains an edge $x = (n, e, m)$ not in I, with n and m objects of type A and B, respectively. Since x is in $M(I, t)$, it is also in $\mathcal{G}(M(I|_{\mathcal{U}}, t) \cup (I - I|_{\mathcal{U}}))$, and hence in $M(I|_{\mathcal{U}}, t) \cup (I - I|_{\mathcal{U}})$. But x is not in I, so x is in $M(I|_{\mathcal{U}}, t)$. Hence, n and m must also be in $M(I|_{\mathcal{U}}, t)$. If A is not colored **u**, then n is clearly not in $I|_{\mathcal{U}}$. If A is not colored **c** either, then n cannot be in $M(I|_{\mathcal{U}}, t)$ either; a contradiction. We conclude that A must be colored **u** or **c**. An identical reasoning applies to B.

The proof of necessity for property 3 is a little bit more involved. Suppose B is a schema node colored **d** and suppose, for the sake of arriving at a contradiction, that an incident schema edge (B, e, C) exists that is neither colored **d** nor **u**, and such that C is not colored **u**.[5] Note that $C \neq B$ because B *is* colored **u** (by property 1, since it is colored **d**). There exists an instance I and a receiver t such that n is in $I - M(I, t)$ for some node n of type B. Note that, since B is colored **u**, n is also in $I|_\mathcal{U} - M(I|_\mathcal{U}, t)$.

Observe that there can be no e-labeled edges incident to n in I. Indeed, such an edge would not be in $M(I, t)$ (since n is not), which is impossible because e is not colored **d**. Moreover, there can be no C-labeled nodes in I. Suppose such a node m is present. Then consider the instance I' obtained from I by adding the edge (n, e, m). By the previous observation, we know that $n \in M(I', t)$. However, since e is not colored **u**, $I'_\mathcal{U} = I_\mathcal{U}$, whence $n \notin M(I'_\mathcal{U}, t) = M(I_\mathcal{U}, t)$. By the "axiom of use" $M(I', t) = \mathcal{G}(M(I'_\mathcal{U}, t) \cup (I' - I'_\mathcal{U}))$, n is thus not in $M(I', t)$ (but we know it is).

Now consider the instance I' obtained from I by adding an object m of type C (we just observed that I does not contain such objects). By the previous observation, we know that $n \in M(I', t)$. However, since C is not colored **u**, $I'_\mathcal{U} = I_\mathcal{U}$, whence $n \notin M(I'_\mathcal{U}, t) = M(I_\mathcal{U}, t)$. Again by the axiom of use we conclude that $n \notin M(I', t)$; a contradiction.

For the if-direction, let κ be a coloring having the properties of the proposition. Note that by these properties, $\kappa(X)$ for any schema node X cannot be $\{d\}$ or $\{c, d\}$. 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:

— For each node X in the schema, fix distinct objects $o_\mathbf{c}^X$, $o_\mathbf{d}^X$, and $o_\mathbf{u}^X$ of type X, and perform the following action depending on the value of $\kappa(X)$:
  (1) $\{\mathbf{c}\}$: Add $o_\mathbf{c}^X$.
  (2) $\{\mathbf{c}, \mathbf{u}\}$: Test if $o_\mathbf{u}^X$ is present; if so, add $o_\mathbf{c}^X$.
  (3) $\{\mathbf{d}, \mathbf{u}\}$: *Provisionally delete* $o_\mathbf{d}^X$. By this we mean that $o_\mathbf{d}^X$ and all its incident edges are removed on condition that the following two tests *fail* for each schema edge (X, e, Y) or (Y, e, X) incident to X:
    — if e is not colored **d** but is colored **u**, test for the presence of any edges labeled e incident to $o_\mathbf{d}^X$.
    — If e is neither colored **d** nor **u**, test for the presence of any Y-labeled nodes (note that by property 3, such nodes are colored **u**).
  (4) $\{\mathbf{c}, \mathbf{d}, \mathbf{u}\}$: Here we perform both actions of the cases $\{\mathbf{c}, \mathbf{u}\}$ and $\{\mathbf{d}, \mathbf{u}\}$.

— For each edge $X = (A, e, B)$ in the schema, fix distinct objects o_1^e and o_3^e of type A, and o_2^e and o_4^e of type B, and perform the following action, depending on the value of $\kappa(X)$:
  (1) $\{\mathbf{c}\}$: *Provisionally create the edge* (o_3^e, e, o_4^e). By this we mean that the edge is added, as well as o_3^e and o_4^e if not yet present, except when A is not colored **c** and o_3^e is not yet present, or B is not colored **c**, and o_4^e is not yet present; in that case we do nothing.

---

[5] The case of a schema edge (C, e, B) is completely analogous.

14   •   M. Andries et al.

(2) $\{\mathbf{d}\}$: In this case we know that at least A or B is colored $\mathbf{d}$. If A is colored $\mathbf{d}$, provisionally delete o_1^e; else, provisionally delete o_3^e.

(3) $\{\mathbf{c}, \mathbf{d}\}$: Here we perform both actions of the cases $\{\mathbf{c}\}$ and $\{\mathbf{d}\}$.

(4) $\{\mathbf{c}, \mathbf{u}\}$: Test if (o_1^e, e, o_2^e) is present; if so, provisionally create the edge (o_3^e, e, o_4^e).

(5) $\{\mathbf{d}, \mathbf{u}\}$: Remove the edge (o_1^e, e, o_2^e).

(6) $\{\mathbf{c}, \mathbf{d}, \mathbf{u}\}$: Here we perform both actions of the cases $\{\mathbf{c}\}$ and $\{d, u\}$.

— For each schema node X with $\kappa(X) = \{\mathbf{u}\}$, we have not yet prescribed an action. If some of the actions described so far that have to be performed test for the presence of certain objects of type X, we do nothing extra. Otherwise, test for the presence of $o_{\mathbf{u}}^X$; if not present, go into an infinite loop.

— Also for each scheme edge (A, e, B) with $\kappa(e) = \{\mathbf{u}\}$, we have not yet prescribed an action. If some of the actions described so far that have to be performed test for the presence of certain edges of type e, we do nothing extra. Otherwise, test for the presence of (o_1^e, e, o_2^e); if not present, go into an infinite loop.

Let us verify that the conditions of Theorem 4.8 are satisfied by M and κ. Conditions 1 and 2 are clear: M never creates (deletes) information of any type unless that type is colored $\mathbf{c}$ ($\mathbf{d}$). Condition 4 is equally clear, and condition 5 is a given property of κ. These remains condition 3, for which we have to verify that $M(I, t) = \mathcal{G}(M(I|_{\mathcal{U}}, t) \cup (I - I|_{\mathcal{U}}))$. We verify both inclusions.

For the inclusion from left to right, consider first a node n of type X in $M(I, t)$. We make the following case analysis:

— n does not equal $o_{\mathbf{c}}^X$, $o_{\mathbf{d}}^X$, or $o_{\mathbf{u}}^X$, and neither o_i^e for some edge label e and $i \in \{1, \ldots, 4\}$. In this case the inclusion is clear: n was already in I because M never adds a node like n; if n is not in $I - I|_{\mathcal{U}}$, it is in $I|_{\mathcal{U}}$, and thus in $M(I|_{\mathcal{U}}, t)$ because M never removes a node like n.

— $n = o_{\mathbf{c}}^X$. If n was already in I, the inclusion is again clear: if n is not in $I - I|_{\mathcal{U}}$, it is in $I|_{\mathcal{U}}$ and thus in $M(I_{\mathcal{U}}, t)$ because M never removes $o_{\mathbf{c}}^X$. If n is not in I, then X must be colored $\mathbf{c}$, and we distinguish between the following two possibilities for the value of $\kappa(X)$:

— $\{\mathbf{c}\}$: Then $o_{\mathbf{c}}^X$ is always added, and hence is in $M(I|_{\mathcal{U}}, t)$. There is, however, one possible caveat: $M(I|_{\mathcal{U}}, t)$ might have gone into an infinite loop because some node or edge x is not present in $I|_{\mathcal{U}}$ (cf., the last two items in the description of M's behavior). However, this is not the case, because $M(I, t)$ did terminate, so x is present in I; but then x is also present in $I|_{\mathcal{U}}$ because we have taken care in the definition of the behavior of M to test for the presence of items only if they are of a type colored $\mathbf{u}$.[6]

— $\{\mathbf{c}, \mathbf{u}\}$ or $\{\mathbf{c}, \mathbf{d}, \mathbf{u}\}$: Then $o_{\mathbf{c}}^X$ has only been added because $o_{\mathbf{u}}^X$ is in I. However, then $o_{\mathbf{u}}^X$ is also in $I|_{\mathcal{U}}$, and hence n is in $M(I|_{\mathcal{U}}, t)$ as well.

---

[6]The same possible caveat applies to various other places in this part of the proof; the reason why it does not pose a problem is always the same.

Strictly, we should also account for the case where $M(I, t)$ does not terminate. But then again this will be due to the absence of a certain node or edge, which will then certainly be absent in $I|_{\mathcal{U}}$ as well; hence $M(I|_{\mathcal{U}}, t)$ will not terminate either in this case.

—$n = o_{\mathbf{d}}^X$. Then n was already in I because M never adds $o_{\mathbf{d}}^X$. If X is not colored $\mathbf{d}$, the inclusion is again clear. If X is colored $\mathbf{d}$ (whence also $\mathbf{u}$), the presence of n in $M(I, t)$ implies the presence in I of one of the following two possibilities:

— An edge, incident to n, not labeled $\mathbf{d}$ but labeled $\mathbf{u}$. This edge will also be present in $I|_{\mathcal{U}}$, and hence n will also be in $M(I|_{\mathcal{U}}, t)$.

— A node labeled $\mathbf{u}$. Again this node will also be present in $I|_{\mathcal{U}}$, so that n is also in $M(I_{\mathcal{U}}, t)$.

—$n = o_{\mathbf{u}}^X$. Then n was already in I, since M never adds $o_{\mathbf{u}}^X$. Hence, if n is not in $I - I|_{\mathcal{U}}$, it is in $I|_{\mathcal{U}}$ and thus also in $M(I_{\mathcal{U}}, t)$, since M never deletes $o_{\mathbf{u}}^X$.

—$n = o_1^e$ or o_2^e for some label e of a schema edge (A, e, B). These cases are analogous to the case $n = o_{\mathbf{d}}^X$.

—$n = o_3^e$ or o_4^e. These cases are analogous to the case $n = o_{\mathbf{c}}^X$.

Continuing our verification of containment from left to right, consider now an edge x in $M(I, t)$, of type (A, e, B). We make the following case analysis:

—$x = (o_3^e, e, o_4^e)$. If e is not colored $\mathbf{c}$, x plays no special role for M, and the inclusion is clear. So now assume e is colored $\mathbf{c}$. If A and B are colored $\mathbf{c}$, x is always added by M, and the inclusion is trivial. If A (B) is not colored $\mathbf{c}$, then it must be colored $\mathbf{u}$ by property 2. If x was already in I, the inclusion is again clear because M never deletes x, o_3^e, or o_4^e. If x was not in I, its creation has succeeded because o_3^e (o_4^e) is already in I. But then o_3^e (o_4^e) is also in $I|_{\mathcal{U}}$, so that x will also be in $M(I|_{\mathcal{U}}, t)$.

—x is incident to o_1^e or o_2^e. Then x only plays a role in M when $\kappa(e)$ is $\{\mathbf{d}\}$ or $\{\mathbf{c}, \mathbf{d}\}$, and A (B) is colored $\mathbf{d}$ and x is incident to o_1^e (o_2^e). In that case, x has not been deleted because the provisional deletion of o_1^e (o_2^e) did nothing. In an earlier case, $n = o_{\mathbf{d}}^X$, we saw that then the provisional deletion will not do anything when working on $I|_{\mathcal{U}}$. Hence x is also in $M(I|_{\mathcal{U}}, t)$.

—x is incident to $o_{\mathbf{d}}^X$ for some class name X. This case is analogous to the previous one.

— All other kinds of edges x will neither be added not deleted by M, so for them the inclusion is clear.

For the inclusion from right to left, we can argue as follows. First, we already noted earlier that if $M(I|_{\mathcal{U}}, t)$ does not terminate, then neither does $M(I, t)$. Furthermore, if an item that plays a special role in M's behavior is present in $M(I|_{\mathcal{U}}, t)$, it has been added or it has not been deleted. Since the decisions made by M to add or to delete are based entirely on tests involving items colored $\mathbf{u}$, the outcomes of these tests will be the same, regardless whether M is applied to $I|_{\mathcal{U}}$ or to I. Hence, $M(I|_{\mathcal{U}}, t)$ is contained in $M(I, t)$. Finally, $I - I|_{\mathcal{U}}$ is also contained in $M(I, t)$, since items not colored $\mathbf{u}$ are never deleted by M. The only exception are edges labeled $\mathbf{d}$ but not $\mathbf{u}$; but such edges are only deleted by M because their incident nodes are deleted. Hence, the $\mathcal{G}$ operator will remove these edges.

To conclude the proof, we must argue that κ is indeed minimal for M. By inspecting M's behavior, we see that the color $\mathbf{u}$ cannot be omitted from $\kappa(X)$ for schema items X for which M performs tests on the presence of items

of type X. Indeed, the outcome of these tests would become negative, and certain deletions or additions made in $M(I, t)$ would not be made in $M(I|_\mathcal{U}, t)$, violating condition 3 of Theorem 4.8. Similarly, for schema items X colored $\{\mathbf{u}\}$ but not involved in such tests, M goes into an infinite loop in the absence of certain items of type X; by removing the color $\mathbf{u}$, $M(I_\mathcal{U}, t)$ would not terminate in cases where $M(I, t)$ would. Finally, the colors $\mathbf{c}$ and $\mathbf{d}$ clearly cannot be omitted either, as M creates (deletes) information of the types colored $\mathbf{c}$ ($\mathbf{d}$).   □

Properties 2 and 3 of Proposition 4.13 are quite intuitive. Property 2 states that one cannot create an edge without checking first for each incident node that it is already present (in this case the node is used), except when we create the node at the same time. Property 3 expresses that you cannot delete a node without deleting its incident edges; if we want to avoid deleting edges, we must delete the node only if a test for the presence of incident edges fails. But testing for the presence of incident edges implies that we use them; if we want to avoid this usage, we can more drastically test whether there are any C-nodes at all (in this case C is used), and do the deletion only if this test fails.

Let us now return to our original motivation: order independence. The colorings describing order-independent updates can be characterized as follows.

THEOREM 4.14. *Let κ be a sound coloring. All update methods having κ as their minimal coloring are order independent if and only if κ is simple.*

PROOF. For the if-implication, let M be an update method having κ as its minimal coloring. By Proposition 4.10, M is inflationary,

$$I \subseteq M(I, t) \tag{3}$$

for any instance I and receiver t. Furthermore,

$$I|_\mathcal{U} = M(I, t)|_\mathcal{U}. \tag{4}$$

Indeed, Eq. (3) implies $I|_\mathcal{U} \subseteq M(I, t)|_\mathcal{U}$, and this inclusion cannot be strict since any information in $M(I, t)$ but not in I is colored $\mathbf{c}$, and thus not $\mathbf{u}$ because κ is simple.

We now use these two observations to prove that M is order independent. By Lemma 3.3 it is sufficient to show that $M(M(I, t), t') = M(I, t) \cup M(I, t') = M(M(I, t'), t)$ for any pair $\{t, t'\}$ of receivers. We can deduce:

$$
\begin{aligned}
M(M(I, t), t') \\
&= \mathcal{G}(M(M(I, t)|_\mathcal{U}, t') \cup (M(I, t) - M(I, t)|_\mathcal{U})) \\
&= \mathcal{G}(M(I|_\mathcal{U}, t') \cup (M(I, t) - I|_\mathcal{U})) \\
&= \mathcal{G}(M(I|_\mathcal{U}, t') \cup M(I, t)) \\
&= M(I|_\mathcal{U}, t') \cup M(I, t) \\
&= M(I|_\mathcal{U}, t') \cup (I - I|_\mathcal{U}) \cup M(I, t) \\
&= \mathcal{G}(M(I|_\mathcal{U}, t') \cup (I - I|_\mathcal{U})) \cup M(I, t) \\
&= M(I, t') \cup M(I, t).
\end{aligned}
$$

For the only-if direction, assume κ is not simple. 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 give a method of type $[R, A]$ that is not order independent, having κ as its minimal coloring. We start with the method associated with κ according to the proof of Proposition 4.13. 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.
    To see that this update is not order independent, apply it to instance $\{n, m\}$, where n and m are objects of type R, and the set of receivers $\{n, m\} \times \{n, m\}$.

(2) As in the previous case, but if the test fails add two new objects to class R.
    To see that this update is not order independent, use the same instance and set of receivers as in the previous case.

(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.
    To see that this update is not order independent, use the same instance and set of receivers as in the previous case.

(4) If there is an edge with label a between receiving and argument object, delete all other a-edges.
    To see that this update is not order independent, apply it to an instance of the form $R \xrightarrow{a} A \xleftarrow{a} R$ on the set of receivers $\{[n, m] \mid (n, a, m) \in I\}$.

(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.
    To see that this update is not order independent, use the same instance and set of receivers as in the previous case.

(6) If there are no a-edges, add one between receiving and argument object.
    To see that this update is not order independent, apply it to an instance of the form $R\ A\ R$ on the set of receivers $\{[n, m] \mid n$ of type R, m of type $A\}$. □

Example 4.15.   Recall the example schema (Example 2.3). To illustrate Theorem 4.14, consider the update method of type [Drinker] that 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.

4.3 Deflationary Colorings

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

18   •   M. Andries et al.

I, and any item x in I whose label is *not* in $\mathcal{X}$, $M(\mathcal{G}(I - \{x\}), t) = \mathcal{G}(M(I, t) - \{x\})$.

Notice how conceptually different the above definition is from our first Definition 4.7. In a sense, the first definition is more global while the second is more local. The two definitions are also formally different, as shown in the following example. In a sense, the two definitions are each other's dual in the way they treat deletion and creation of information.

Example 4.17.   Consider the method that deletes all objects of a certain class X. If this method uses only information of type $\mathcal{X}$ according to Definition 4.7, X must be in $\mathcal{X}$, but this is not true under Definition 4.16.

On the other hand, consider the method that always adds some fixed object of type X. Now it is according to Definition 4.16 that X must be in $\mathcal{X}$, but no longer under Definition 4.7.

It turns out that one can repeat the entire development of the previous section under the new Definition 4.16. First, we have:

THEOREM 4.18.   *Exactly the same statement of Theorem 4.8 also holds when the meaning of the term "use" is that of Definition 4.16.*

We omit the proof; it is along the same lines as that of Theorem 4.8, but technically easier. For the remainder of this section, the meaning of "use" in formal propositions, theorems, and proofs is always that of the new Definition 4.16.

We also prove the verbatim analog of Theorem 4.14, and again formulate a soundness criterion. The curious duality alluded to above will have as an effect that simple colorings under the new definition describe *deflationary* rather than inflationary updates.

The following proposition is the analog of Proposition 4.10. While for the old axiomatization of "use", we see in Lemma 4.11 that, roughly, we cannot delete information without using it, we now see in Lemma 4.20 that for the new axiomatization, we cannot *create* information without using it. Indeed, the statement of Lemma 4.20 is exactly that of Lemma 4.11 with '**d**' replaced by '**c**'. This formalizes the duality already alluded to in Example 4.17.

PROPOSITION 4.19.   *Let M be an update method. If the minimal coloring of M (as given by Theorem 4.18) is simple, then M is* deflationary, *i.e., $M(I, t) \subseteq I$ for each instance I and receiver t over I.*

PROOF.   The proof is technically quite different from that of Proposition 4.10. Let the minimal coloring of M be κ. We prove the following technical lemma:

LEMMA 4.20.   *If a node in the schema is colored **c** by κ, then it is also colored **u**. If an edge is colored **c** by κ, then either it is also colored **u** or one of its incident nodes is colored **c**.*

This lemma clearly implies the proposition is proven: if κ is simple, it cannot color anything **c**, and hence M will never create any information, i.e., M is deflationary.

To prove the first statement of the lemma, assume X is a node in S colored **c** but not **u**. The minimality of κ implies the existence of an instance I and a receiver t such that $M(I, t)$ contains an object n of type X that is not in I. Since X is assumed not colored **u**, we have

$$M(\mathcal{G}(\bar{I} - \{\bar{n}\}), \bar{t}) = \mathcal{G}(M(\bar{I}, \bar{t}) - \{\bar{n}\})$$

for any instance $\bar{I}$, node $\bar{n}$ labeled X in $\bar{I}$, and receiver $\bar{t}$ over $\bar{I}$. Applying this to $I' := I \cup \{n\}$, n, and t, we obtain

$$M(I, t) = \mathcal{G}(M(I', t) - \{n\}).$$

But $M(I, t)$ contains n, while the right-hand side of the equation clearly does not; a contradiction.

To prove the second statement, assume there is an edge in S, labeled e, that is colored **c** but not **u**. Then there exists an instance I and a receiver t such that $M(I, t)$ contains an edge $x = (n, e, m)$ that is not in I. Since e is not colored **u**, we have $M(\mathcal{G}(\bar{I} - \{\bar{x}\}), \bar{t}) = \mathcal{G}(M(\bar{I}, \bar{t}) - \{\bar{x}\})$ for any instance $\bar{I}$, edge $\bar{x}$ labeled e in $\bar{I}$, and receiver $\bar{t}$ on $\bar{I}$. Applying this to $I' := I \cup \{n, m, x\}$, x, and t, we obtain $M(I \cup \{n, m\}, t) = \mathcal{G}(M(I', t) - \{x\})$. If n and m were in I, then the left-hand side of this equation would equal $M(I, t)$, which contains x, while x is obviously not contained in the right-hand side of the equation. Consequently, either n or m is not in I. But since these nodes are in $M(I, t)$, one of their labels must be colored **c**. $\square$

If the duality were perfect, we would now expect the soundness criterion under the new axiomatization of use to be the soundness criterion under the previous axiomatization (Proposition 4.13), where we replaced the first property (from Lemma 4.11) with the property in the dual, Lemma 4.20. However, the duality is not perfect. The point is that Lemma 4.20 already provides a weak form of property 2 required by Proposition 4.13; and it turns out that the latter property itself is no longer necessary. This is shown by the following example.

Example 4.21.   Consider a schema with two class names A and B and a property e of A of type B. Consider the coloring assigning {**u**, **c**} to A, {**c**} to e, and $\emptyset$ to B. This coloring is not sound under Definition 4.7 (it does not satisfy property 2 from Proposition 4.13), but it is sound under Definition 4.16. Indeed, as we can verify, it is the minimal coloring of the update that checks to see if some fixed object n^A of type A is already present; if not, it adds n^A, together with edges to all B-nodes present. Definition 4.7 considers these B-nodes to be used by the update, but Definition 4.16, by its more "local" nature, does not.

As a new soundness criterion, we get:

PROPOSITION 4.22.   *A coloring is sound if and only if it has the following properties:*

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

20   •   M. Andries et al.

(2) *If a node is colored* **d***, then all its incident edges are colored* **u** *or* **c***, or the other node incident to such an edge is colored* **u***.*

(3) *At least one node is colored* **u***.*

(4) *If an edge is colored* **u***, then so are its incident nodes.*

The proof of the only-if direction has already been given for the first property in Lemma 4.20; it is clear for the last two properties, and is analogous to the proof of Proposition 4.13 for the remaining property (which is identical in both propositions). The proof of the if-direction requires no new ideas beyond those of the proof of the if-direction of Proposition 4.13; the only extra complication is for edges colored **c**; these are dealt with as shown in Example 4.21.

We can conclude this section with the verbatim analog of Theorem 4.14:

THEOREM 4.23. *Let κ be a sound coloring. All update methods having κ as their minimal coloring are order independent, if and only if κ is simple.*

PROOF. The only-if implication is proven analogously to Theorem 4.14. Assume κ is not simple. Then, by Proposition 4.22, we only have to consider the same six possibilities as in the proof of Theorem 4.14, namely of a node or an edge colored $\{\mathbf{u}, \mathbf{c}\}$, $\{\mathbf{u}, \mathbf{d}\}$, or $\{\mathbf{u}, \mathbf{c}, \mathbf{d}\}$. The same six examples used in that proof also apply to the present setting. Again, these methods have to be adapted to the colors of other items in the scheme. Thereto, we merely have to replace the case of a node colored **c** by that of a node colored **d** in the obvious way.

For the if-implication, let M be an update having κ as its minimal coloring. By Lemma 3.3, it is sufficient to show that $M(M(I, t), t') = M(M(I, t'), t)$ for any pair $\{t, t'\}$ of receivers. Let $\{x_1, \ldots, x_n\}$ be the set of all items of the partial instance $I - M(I, t)$. The labels of all these items and edges are colored **d**, and therefore not **u** because κ is simple. Hence, $M(\mathcal{G}(I - \{x_1\}), t') = \mathcal{G}(M(I, t') - \{x_1\})$. Subtracting $\{x_2\}$ from both sides followed by applying $\mathcal{G}$ yields

$$\mathcal{G}(M(\mathcal{G}(I - \{x\}), t') - \{x_2\}) = \mathcal{G}(\mathcal{G}(M(I, t') - \{x_1\}) - \{x_2\}),$$

and thus

$$M(\mathcal{G}(I - \{x_1, x_2\}), t') = \mathcal{G}(M(I, t') - \{x_1, x_2\}).$$

We can repeat this reasoning for all other items x_i ($2 < i \leq n$), and eventually results in

$$M(\mathcal{G}(I - (I - M(I, t))), t') = \mathcal{G}(M(I, t') - (I - M(I, t'))),$$

which can be rewritten as

$$M(M(I, t), t') = M(I, t) \cap M(I, t').$$

Hence, by symmetry, $M(M(I, t), t') = M(M(I, t'), t)$, had to be proven. □

4.4 A Conclusion on Colorings

The specification of update behavior on a language-independent level is a notoriously difficult problem. Our results in this section show that it is not impossible

to solve when the updates have a uniform behavior (i.e., are inflationary or deflationary). We would also like to make clear that we do not intend to claim that colorings based on only three kinds of update behavior (creation, deletion, and usage) are rich enough for specification. Indeed, although we find our results rather satisfying from a theoretical point of view, their practical usefulness remains limited (but see Section 7 for some practical implications). A study of schema annotations, which can distinguish more kinds of update behavior, is a challenging and interesting issue for further research.

5. A MODEL OF 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 [1989].

5.1 Preliminaries

It is well-known (e.g., Lyngbaek and Vianu [1987] Hull and Su [1989]; Hull and Yoshikawa [1990]) that object-base schemas and instances can be viewed naturally as relational database schemas and instances. Formally, assume that all class names and property names are attribute names. Following the standard convention, we 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. Note that every relational instance of this schema will also satisfy the disjointness dependencies $C[C] \cap C'[C'] = \emptyset$ for each pair of different class names C and C', since we agreed in Section 2 that different class names have disjoint universes of objects.

The following proposition is clear:

PROPOSITION 5.1. *The object-base instances of S correspond precisely to the relational database instances of the relational database schema corresponding to S.*

Henceforth, we blur the distinction between an object-base schema or instance and its relational representation.

We can now use the relational algebra to derive relations from object-base instances. The algebra we use is the standard one [Ullman 1988], consisting of the binary operators union ($\cup$), difference ($-$), Cartesian product ($\times$), and the unary operators equality selection ($\sigma_{A=B}$), projection ($\pi_{A_1,\dots,A_p}$), and renaming ($\rho_{A \to B}$). We also use a weaker algebra, called the "positive" algebra:

Definition 5.2.   The *positive algebra* consists of the operators union, Cartesian product, equality selection, projection and renaming, plus the nonequality selection ($\sigma_{A \neq B}$).

22   •   M. Andries et al.

Note that the positive algebra does not include the difference operator.

Following standard practice, we use natural joins ($\bowtie$) and theta-joins ($\underset{\theta}{\bowtie}$) as abbreviations of the well-known combinations of Cartesian product, selection, and renaming.

It is well-known [Abiteboul et al. 1995] that equivalence (or satisfiability) of arbitrary relational algebra expressions is undecidable. However, the standard interpretation of "equivalence" is that two expressions over some schema are equivalent if they have the same value on *every possible* instance of that schema, disregarding any integrity constraint on instances that may be present. In our setting, such integrity constraints are indeed present: as explained above, object-base instances satisfy certain inclusion and disjointness dependencies. We regard two expressions as equivalent if they have the same value on every object-base instance; we are not interested in relational instances that do not represent an object-base instance.

The preceding discussion motivates the following lemma. It implies that equivalence over object-base instances is undecidable as well.

LEMMA 5.3. *Testing equivalence of relational algebra expressions over arbitrary relational instances can be reduced to testing equivalence over object-base instances.*

PROOF. Let S be an object-base schema and let $R = AB$ be a classical binary relation scheme where A and B are attribute names with the same infinite domain D. Consider the standard finite satisfiability problem: given a relational algebra expression E over R, does there exist a relation instance r such that $E(r) \neq \emptyset$? We reduce this problem to satisfiability over object-base instances of S. Since E_1 and E_2 are equivalent if and only if $(E_1 - E_2) \cup (E_2 - E_1)$ is not satisfiable, this reduction suffices to prove the lemma.

We first show how a binary relation can be represented by an object-base instance. Assume S contains class names C and D and edges (C, A, D) and (C, B, D).[7] A relation $r = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ can be represented by an object-base instance I of S where

— the nodes labeled D are $\{a_1, \ldots, a_n, b_1, \ldots, b_n\}$;
— the nodes labeled C are n abstract nodes $\{t_1, \ldots, t_n\}$;
— the edges are all those of the form (t_i, A, a_i) and (t_i, B, b_i) for $i = 1, \ldots, n$.

In such an instance I, the expression $\pi_{A,B}(CA \bowtie CB)$ evaluates to r. Hence, an expression E over R is satisfiable if and only if the expression E' over S obtained from E by replacing each occurrence of R by $\pi_{A,B}(CA \bowtie CB)$ is satisfiable. □

5.2 Algebraic Update Methods

We are now ready to define our update methods algebraic model. We consider methods that can only update the properties of the receiving object.

---

[7]The lemma can also be proven under the assumption of a single class name C and a single edge (C, e, C).

They cannot remove existing objects or create new ones. The 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, we have the following definitions. We fix an object-base schema S in what follows.

Definition 5.4

(1) Let $\sigma = [C_0, \ldots, C_k]$ be a method signature. An *update expression of type σ* is a unary relational algebra expression over the relation schemes in S and the special unary relation schemes *self* and arg_i for $1 \le i \le k$, where the domain of *self* is Δ_{C_0} and the domain of arg_i is Δ_{C_i} for $1 \le i \le k$.

(2) Let E be an update expression of type σ. Let I be an instance of S and let $t = [o_0, \ldots, o_k]$ be a receiver of type σ over I. Then $E(I, t)$ is defined as the result of evaluating E 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 singleton $\{o_i\}$ containing the ith argument, for $1 \le i \le k$.

(3) 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 an update expression of type σ.

(4) 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.

(5) Finally, if M is an algebraic update method of type σ, I is an instance of S, and t is a receiver of type σ over I, the *result of applying M to (I, t)* is defined as the instance obtained from I by replacing, for each statement $a := E$ in M, all edges labeled a leaving the receiving object by edges to all elements of $E(I, t)$.

Example 5.5.   In writing examples of algebraic methods, we abbreviate the class and property names from our example schema by their first letter (Bar and Beer are abbreviated as *Ba* and *Be*).

The method *favorite_bar* of Example 2.7 can be implemented simply as $f := arg_1$, and the method *add_bar* as $f := \pi_f(self \underset{self=D}{\bowtie} Df) \cup arg_1$. The method of Example 4.15 can be implemented as

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

In practice, syntactic sugar such as path expressions can be used to write algebraic update methods more concisely.

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 B is the type of property a, then we must have $E(I, t) \subseteq B(I)$ for any instance I and receiver t. Not surprisingly, in view of Lemma 5.3, this property of expressions E is undecidable. However, by using "many-sorted" expressions, well-definedness can be syntactically guaranteed, and this without loss of expressive power [Van den Bussche and Cabibbo 1998]. Another, pragmatical, solution is to use

24   •   M. Andries et al.

only expressions E of the form $E' \cap B$. Hence, well-definedness does not really 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.

THEOREM 5.6. *The problem of deciding equivalence between relational algebra expressions (over arbitrary relational instances) is reducible to the problem of deciding order independence of algebraic methods.*

Conversely, method order independence is reducible to expression equivalence under functional, inclusion, and disjointness dependencies.

PROOF. We first reduce expression equivalence to method order independence. By Lemma 5.3, it suffices to reduce equivalence over object-base instances to order independence.

Let S be an object-base schema and let E_1 and E_2 be two expressions over S. Without loss of generality, we assume E_1 and E_2 to have the empty result scheme. Augment S with a class name C having two properties a and b of type C. The following update method M of type $[C]$ is order independent if and only if E_1 and E_2 are equivalent:

$$
\begin{aligned}
a &:= \emptyset; \\
b &:= \textbf{if } Ca = C \times \rho_{C \to a}(C) \\
&\qquad \textbf{then if } E_1 \neq \emptyset \textbf{ then } \textit{self} \textbf{ else } \emptyset \\
&\qquad \textbf{else if } E_2 \neq \emptyset \textbf{ then } \textit{self} \textbf{ else } \emptyset.
\end{aligned}
$$

In proof, assume $E_1(I)$ is empty but $E_2(I)$ is not for some instance I. Let I' be obtained from I by adding two objects o and o' in class C with all 8 possible a- and b-edges between them. Then in $M(M(I', o), o')$, there is no b-edge leaving o, but in $M(M(I', o'), o)$ there is. Hence, M is not order-independent.

Conversely, if E_1 and E_2 are equivalent, then the update to b simplifies to

$$b := \textbf{if } E_1 \neq \emptyset \textbf{ then } \textit{self} \textbf{ else } \emptyset,$$

which makes M order-independent, since E_1 does not depend on C and its properties.

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 $\textit{self}, \textit{arg}_1, \ldots, \textit{arg}_k$ together hold a receiver t, then the relation Ca in the instance $M(I, t)$ can be expressed as

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

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 $\textit{self}', \textit{arg}'_1, \ldots, \textit{arg}'_k$ together hold a second receiver t'. Then the relation Ca in the instance $M(I, tt')$ can be expressed as

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

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.

Indeed, only object-base instances must be considered. This is dealt with by imposing the inclusion and disjointness dependencies corresponding to the object-base schema.

Moreover, only interpretations of the relations $self, self', arg_1, \ldots, arg'_k$ must be considered 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 the functional dependencies $self \colon \emptyset \to self$ and $arg_i \colon \emptyset \to arg_i$ (and similarly for $self'$ and arg'_i). Requirements *(ii)* and *(iii)* are dealt with by modifying the expressions to yield the empty result if they are not satisfied. This can be done by taking the Cartesian product with

$$\pi_\emptyset(self \times arg_1 \times \cdots \times arg_k \times self' \times arg'_1 \times \cdots \times arg'_k) \times$$
$$\bigcup_{i=1}^{k} \pi_\emptyset\left(arg_i \underset{arg_i \neq arg'_i}{\bowtie} arg'_i\right) \cup \pi_\emptyset\left(self \underset{self \neq self'}{\bowtie} self'\right). \quad \square$$

The first part of the above theorem implies the following undecidability results. In the next section, we use the second part of the theorem to obtain decidability results in the special case of "positive" methods.

COROLLARY 5.7. *The following problems are undecidable:*

(1) *Given an algebraic method M, is M order independent?*
(2) *Given an algebraic method M, is M key-order independent?*
(3) *Given a relational algebra query Q over the object-base schema and an algebraic method M, is M Q-order independent?*

PROOF.   Problem 1 follows immediately from the first part of Theorem 5.6 and the undecidability of equivalence of relational algebra expressions. The proof of Theorem 5.6 shows that the reduction from expression equivalence to method order independence also works for key-order independence. Hence, problem 2 follows as well. We leave it as an exercise for the reader to reduce order independence to query-order independence. Problem 3 follows from this. $\square$

To conclude the present section, we present a sufficient condition for key-order independence in the general case.

PROPOSITION 5.8. *An algebraic method M is key-order independent if each of its update expressions does not access the relations corresponding to the properties updated by M.*

Instead of proving this proposition formally, which is straightforward, we note that the condition is only sufficient for key-order independence, not for absolute order independence. Indeed, the update $a := arg$ satisfies the condition, but is not order independent (only key-order independent).

Example 5.9.   The method *favorite_bar* satisfies the condition of Proposition 5.8 (see Example 5.5), and is indeed key-order independent. Observe that *add_bar* does not satisfy the condition (it both accesses and modifies relation *Df*) but is still order independent; this shows that the condition is only sufficient.

PROPOSITION 5.8, trivial as it may be, covers many practical cases; more examples of its applicability are given in Section 7.

5.3 Positive Update Methods

An important special kind of algebraic method is the "positive" one:

Definition 5.10.   An algebraic method is called *positive* if all relational algebra expressions used in it are positive (in the sense of Definition 5.2).

Since positive algebra expressions always express monotone queries, positive methods always express monotone updates, i.e., if $I \subseteq J$ then $M(I, t) \subseteq M(J, t)$.

The following example shows that positive methods can still delete information.

Example 5.11.   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

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

Our main positive result concerning algebraic update methods is the following:

THEOREM 5.12.   *Order independence and key-order independence of positive algebraic methods are decidable.*

PROOF.   The proof is based on the following lemma, whose complete proof is given in Appendix A.

LEMMA 5.13.   *Containment* (*whence also equivalence*) *of positive relational algebra expressions under functional dependencies and full inclusion dependencies is decidable.*

The theorem is implied by this lemma and the reduction from order independence to equivalence of relational algebra expressions given in the proof of the second part of Theorem 5.6. To see this, note the following facts concerning this reduction:

(1) The reduction preserves positivity: if the method to be checked for order independence is positive, then so are the generated expressions to be checked for equivalence.

(2) The reduction can be readily adapted for key-order independence. It suffices to omit, in the large final expression of the proof of Theorem 5.6, the term $\bigcup_{i=1}^{k} \pi_\emptyset(arg_i \bowtie_{arg_i \neq arg'_i} arg'_i)$, so that the expressions to be evaluated become empty from the moment the two receivers have the same receiving object. (Recall that Lemma 3.3 also holds for key-order independence.)

(3) The dependencies involved in the reduction are covered by those mentioned in Lemma 5.13. Hence, the lemma yields the decidability of order independence and key-order independence.   □

It remains open whether the following problem is decidable:

Open problem.   Given a positive relational algebra query Q over the object-base schema and a positive algebraic method M, is M Q-order independent?

The reason our techniques fail to solve this problem is that they crucially rely on Lemma 3.3, which fails for query-order independence. More precisely:

PROPOSITION 5.14.   *The following statement does* not *hold for every positive algebra query Q and positive algebraic method M:*

> *M is Q-order independent iff M is order independent on any pair (I, T) where T is a two-element subset of $Q(I)$.*

In fact, none of the two implications (if and only if) holds in general.

PROOF.   Consider a schema with a class name C having two properties a and b of type C. We give counterexamples disproving the two implications.

We first disprove the if-direction. Consider the update M of type $[C, C]$ deleting the argument object from the a-properties of the receiving object, on condition that relation Ca contains at least two tuples. We can express M as a positive algebraic method as follows:

$$a := \textbf{if } \#Ca \geq 2 \textbf{ then } \pi_a\big(self \underset{self=C}{\bowtie} Ca \underset{a \neq arg}{\bowtie} arg\big) \textbf{ else } \emptyset.$$

Furthermore, consider the query

$$Q := \textbf{if } \#Ca \geq 3 \textbf{ then } Cb \textbf{ else } \emptyset.$$

Note that a query of the form **if** $\#Ca \geq 2$ **then** E **else** $\emptyset$ (or $\#Ca \geq 3$) can indeed be expressed positively; for example, the former as

$$\pi_\emptyset\big(\pi_C(Ca) \underset{C \neq C'}{\bowtie} \rho_{C \to C'}\pi_C(Ca) \cup \pi_a(Ca) \underset{a \neq a'}{\bowtie} \rho_{a \to a'}\pi_a(Ca)\big) \times E.$$

Under these assumptions, it follows that $M(I, t_1 t_2) = M(I, t_2 t_1)$ for any instance I and pair of distinct receivers $t_1, t_2 \in Q(I)$. Indeed, if $Q(I)$ is nonempty, then $\#Ca$ is at least 3. Hence, applying M to (I, t_1) amounts to deleting t_1 from Ca, after which $\#Ca$ is still at least 2, so that applying M to $(M(I, t_1), t_2)$ amounts to deleting t_2 from Ca. Clearly, this is equivalent to first deleting t_2 and only then t_1, so that $M(I, t_1 t_2) = M(I, t_2 t_1)$.

However, M is not Q-order independent. Indeed, consider an instance I where relation Ca equals $\{(c_1, a_1), (c_2, a_2), (c_3, \alpha)\}$ and Cb equals $\{(c_1, a_1), (c_2, a_2), (c_3, \beta)\}$ with $\alpha \neq \beta$. In $M(I, (c_1, a_1)(c_2, a_2)(c_3, \beta))$, object c_3 has no a-properties, while in $M(I, (c_3, \beta)(c_1, a_1)(c_2, a_2))$, it still has α as an a-property. Hence, these two sequential applications differ.

28   •   M. Andries et al.

We next disprove the only-if direction. Consider the update M of type $[C, C, C]$, which assigns to a all the b-properties of the receiving object and adds the first argument to the b-properties (the second argument is not used). We can express M as a positive algebraic method as follows:

$$a := \pi_b\big(self \underset{self=C}{\bowtie} Cb\big);$$

$$b := \pi_b\big(self \underset{self=C}{\bowtie} Cb\big) \cup arg_1.$$

Consider furthermore the three-fold Cartesian product of C with itself as a query Q.

Then M is Q-order independent. Indeed, if I contains less than two objects, then $Q(I)$ returns no more than one receiver, and the application is trivially order independent. If I contains more than one object, applying M to $(I, Q(I))$ sequentially (regardless of order) will result in the instance in which every object has all other objects as a- and b-properties.

However, there is an instance I and a pair of distinct receivers $t_1, t_2 \in Q(I)$ such that $M(I, t_1 t_2) \neq M(I, t_2 t_1)$. Indeed, consider the instance I containing two objects o_1 and o_2 having no a- or b-properties. Consider the following pair of receivers $t_1, t_2 \in Q(I)$: $t_1 = (o_1, o_1, o_1)$ and $t_2 = (o_1, o_2, o_1)$. In $M(I, t_1 t_2)$, relation Ca equals $\{(o_1, o_1)\}$, while in $M(I, t_2 t_1)$ it equals $\{(o_1, o_2)\}$. Hence, $M(I, t_1 t_2) \neq M(I, t_2 t_1)$.   □

6. PARALLEL APPLICATION OF ALGEBRAIC UPDATE METHODS

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

An update expression E occurring in an algebraic update method 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 '$self$' with '$\pi_{self}(rec)$' and 'arg_i' with '$\pi_{arg_i}(rec)$.'

Using this relation rec suggests a natural semantics for applying the update 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.

This motivates the following definition:

Definition 6.1.   Let E be an update expression. Then $par(E)$ is the relational algebra expression over the relation schemes in the object-base schema plus the relation scheme $rec = self\,arg_1 \ldots arg_k$ obtained by modifying E as follows:

—Each relation scheme R occurring in E 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 is modified in a natural join on the common attribute *self*.

Note that the result scheme of $par(E)$ is that of E to which the attribute *self* is added. For instance, if C is the receiving class and the update expression E is over a property a of type B of C, then the result scheme of $par(E)$ is *self a*, whose domains are C and B, respectively.

The result of applying a method M in parallel to an instance I and a set T of receivers over I is now defined in the obvious way:

Definition 6.2.

(1) Let E be an update expression. Then $par(E)(I, T)$ is defined as the result of evaluating $par(E)$ on I, where the relation *rec* is interpreted by T.

(2) The *result of applying M in parallel to* (I, T), denoted $M_{\mathrm{par}}(I, T)$, is defined as the instance obtained from I by replacing, for each statement $a := E$ in M and 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 $par(E)(I, T)$.

The parallel semantics just defined coincides with the ordinary semantics in the case of a single receiver, as stated in the following proposition. The proof is straightforward.

PROPOSITION 6.3. *For any algebraic update method M, instance I and receiver t, $M_{\mathrm{par}}(I, \{t\}) = M(I, t)$.*

The following example illustrates parallel application and contrasts it against sequential application.

Example 6.4.   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

$$tc := \pi_e\left(self \underset{self=C}{\bowtie} Ce\right) \cup \pi_e\left(self \underset{self=C}{\bowtie} Ctc \underset{tc=C'}{\bowtie} \rho_{C \to C'}(Ce)\right).$$

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, $par(E)$ (E being the expression assigned to tc) equals

$$\pi_{self,e}\left(\pi_{self}(rec) \underset{\substack{self=self \\ self=C}}{\bowtie} (\pi_{self}(rec) \times Ce)\right)$$

$$\cup\, \pi_{self,e}\left(\pi_{self}(rec) \underset{\substack{self=self \\ self=C}}{\bowtie} (\pi_{self}(rec) \times Ctc) \underset{\substack{self=self \\ tc=C'}}{\bowtie} \rho_{C \to C'}(\pi_{self}(rec) \times Ce)\right),$$

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

The above example suggests 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 (which cannot express transitive closure).[8]

When we restrict attention to key sets of receivers, however, parallel and sequential application are equivalent, as stated by the following theorem.

THEOREM 6.5.   *If M is key-order independent, then $M_{\text{seq}}(I, T) = M_{\text{par}}(I, T)$ for any instance I and key set of receivers T.*

PROOF.   Let E be an update expression occurring on the right-hand side of one of the statements in M. We refer to Lemmas 6.6 and 6.7, stated and proven below.

To see how the theorem follows from these lemmas, consider a statement $a := E$ in M, and assume that C is the receiving class. By Lemma 6.6, relation Ca in $M_{\text{seq}}(I, T)$ equals

$$Ca(I) - \pi_{Ca} \bigcup_{t \in T} \left(\{t(self)\} \underset{self=C}{\bowtie} Ca(I) \right) \cup \rho_{self \to C} \bigcup_{t \in T} \{t(self)\} \times E(I, t),$$

which, by Lemma 6.7, equals relation Ca in $M_{\text{par}}(I, T)$.

LEMMA 6.6.   *If $T = \{t_1, \ldots, t_n\}$, then for each i with $1 < i \leq n$,*

$$E(M(I, t_1 \ldots t_{i-1}), t_i) = E(I, t_i).$$

PROOF.   Let $a := E$ be the statement in M having E as its right-hand side and let C be the receiving class. The objects to which the object $t_i(self)$ is linked by a-edges in $M_{\text{seq}}(I, T)$ are the same as those in $M(I, t_i)$ and in $M(I, t_1 \ldots t_i)$, since T is a key set and M is key-order independent. In $M(I, t_i)$, they can be computed as $E(I, t_i)$; in $M(I, t_1 \ldots t_i)$, they can be computed as $E(M(I, t_1 \ldots t_{i-1}), t_i)$. Hence, $E(I, t_i) = E(M(I, t_1 \ldots t_{i-1}), t_i)$ as had to be proven.   □

LEMMA 6.7. $par(E)(I, T) = \bigcup_{t \in T} \{t(self)\} \times E(I, t).$

PROOF.   A straightforward induction on the structure of E. By way of example we show how the difference operator is handled.

$$
\begin{aligned}
par(E_1 - E_2)(I, T) &= par(E_1)(I, T) - par(E_2)(I, T) \\
&= \bigcup_{t \in T} \{t(self)\} \times E_1(I, t) - \bigcup_{t \in T} \{t(self)\} \times E_2(I, t) \\
&= \bigcup_{t \in T} \{t(self)\} \times (E_1(I, t) - E_2(I, t)).
\end{aligned}
$$

The first equality holds by definition; the second by induction. To prove the third, the inclusion from left to right is straightforward. For the inclusion from right to left, let

$$s \in \bigcup_{t \in T} \{t(self)\} \times (E_1(I, t) - E_2(I, t)).$$

Then $s = t_0(self) \times s_0$, with $s_0 \in E_1(I, t_0) - E_2(I, t_0)$, for some $t_0 \in T$. So s is clearly in $\bigcup_{t \in T} \{t(self)\} \times E_1(I, t_0)$. Now assume s is also in $\bigcup_{t \in T} \{t(self)\} \times E_2(I, t_0)$. Then $s_0 \in E_2(I, t_0')$, for some $t_0' \in T$ with $t_0'(self) = t_0(self)$. But since T is a key set, the latter can only hold when $t_0' = t_0$, which, implies $s_0 \in E_2(I, t_0)$; a contradiction. □

The parallel application of algebraic update methods can be implemented much more efficiently than the sequential application. Indeed, the result of the parallel application is defined in terms of one single relational algebra expression per property to be updated; this expression can be optimized and is then executed only once. This is not possible in the sequential application, where the application to a set of n receivers results in the evaluation of n separate relational algebra expressions. Hence, we believe Theorem 6.5 is an interesting result.

7. PRACTICAL IMPLICATIONS

In this section, we show that our theory can be applied in a practical SQL context and that it can explain a variety of update phenomena in that context.

We begin by noting that the object-based data model we have been using can be applied very well in the classical relational setting as well. A tuple t in some relation R can be interpreted as an object of type R. An attribute $t.A$ can then be interpreted as a property of t. We can also interpret a relation P whose attributes include the primary key of relation R as a foreign key, as well as the primary key of another relation S, as a property (R, P, S). So a tuple (k_1, k_2) in P would be interpreted as an edge (t_1, P, t_2), where t_1 is the tuple in R identified by key k_1 and t_2 is the tuple in S identified by key k_2.

Now consider a relation *Employee*(*EmpId, Salary, Manager*) holding information about the salary and the manager of employees and a list *Fire*(*Amount*) of amounts. Suppose we want to delete all employees whose salary occurs in *Fire*. We can do this in two different ways:

(1) Using a standalone, set-oriented SQL statement:

delete from *Employee* **where** *Salary* **in table** *Fire*

(2) Using a cursor-based delete in programmed SQL:[9]

for each t **in** *Employee* **do**
  if *Salary* **in table** *Fire*
    delete t **from** *Employee*

These two solutions work in entirely different ways. The set-oriented statement, in a first phase, identifies all tuples to be deleted; only in a second phase are they effectively removed. The cursor-based update removes a tuple directly to be deleted before inspecting the next one. Because the cursor-based update is *order independent*, the two end results are the same.

---

[9]In this and the following examples we do not worry about the precise syntax for cursor manipulation, and use an abstract syntax instead.

We can readily see that the cursor-based update is order independent using schema colorings (since we are deleting information we would use the deflationary semantics of colorings). Indeed, the relation *Employee* from which we delete is not used in the **if**-condition, so *Employee* has color {**d**}. No items other than *Employee* objects are deleted, so no other schema item is colored **d**. Nothing at all is created (inserted), so no schema item is colored **c**. Of course, some schema elements are used, more specifically the class *Fire*, the property *Salary*, and the the class *D* to represent the type of this property, so these items have color {**u**}. We thus have a simple coloring, which guarantees order independence by our Theorem 4.23.

An example in which *Employee* would be colored both **d** and **u** is when we try a cursor-based update to delete all employees for which *the salary of their Manager* occurs in *Fire*:

for each t **in** *Employee* **do**
  if exists (**select** $*$
        from *Employee* $E1$
        where $E1.EmpId = Manager$ **and** $E1.Salary$ **in table** *Fire*)
  delete t **from** *Employee*

We now delete from *Employee*, and at the same time use *Employee* information in the **if**-condition of the update. The resulting double color for *Employee* means we can no longer use Theorem 4.23 to conclude order independence; in fact, the above update is order *dependent*, because an employee will not be deleted if his manager was visited and deleted before him. The cursor-based solution is thus wrong for this case.

In contrast, the set-oriented statement

delete from *Employee*
where exists (**select** $*$
        from *Employee* $E1$
        where $E1.EmpId = Manager$ **and** $E1.Salary$ **in table** *Fire*)

is still correct, as it will again first identify the employees to be deleted, and only then remove them all together.[10] In effect, this statement actually uses an extremely simple underlying update that is trivially order independent: this update merely removes its parameter t from relation *Employee*. The whole point is that the complete set T of parameters is computed before the actual deletions are applied. Thus we see that the set-oriented **delete** statement in SQL is very nicely explained by our theory as the application of a trivial, order-independent, removal update to a precomputed set of receivers.

Analogous examples can be given with insertions instead of deletions. Once we move to modifications, however, we can no longer use our coloring framework to analyze update behaviors, since modifications both delete and insert

---

[10]A referee pointed out that some of the earlier SQL implementations did not in fact follow this two-phase semantics, using an order-dependent semantics equivalent to that of the cursor-based deletion instead. We have tested the SQL implementation of two current (1998) DBMSs, and fortunately they no longer have this problem.

at the same time, and our results on simple colorings apply to updates that are either deletion-only (deflationary) or insertion-only (inflationary) only. Hence, we move to the algebraic framework.

As a first example of a modification, consider again the relation *Employee*, along with a relation *NewSal*(*Old, New*) specifying new salaries. Suppose we want to give each employee a new salary as specified in *NewSal*. This can be achieved by the standalone set-oriented update statement

update *Employee*   (*A*)
set *Salary* = (**select** *New*
         from *NewSal*
         where *Old* = *Salary*)

or by the cursor-based update

for each *t* **in** *Employee* **do**   (*B*)
  update *t*
  set *Salary* = (**select** *New*
           from *NewSal*
           where *Old* = *Salary*)

Both solutions are correct; in particular, the cursor-base update is key-order independent. To see how this naturally falls out of our theory, in the algebraic framework, we model this update as consisting of the following single algebraic update statement working on a receiver [*self*, arg_1]:

$$Salary := \pi_{New}\big(arg_1 \underset{arg_1 = Old}{\bowtie} NewSal\big) \qquad (B')$$

This update is then applied to the set of receivers $\{[t(EmpId), t(Salary)] \mid t \in Employee\}$. Note that this is a key set, and thus Proposition 5.8 can be applied to guarantee order independence, since relation *Employee* (in which property *Salary* is stored) does not occur in the expression on the right-hand side of the statement.

An example of a cursor-based modification that is order *dependent* is when we try to give each employee the new salary *his manager* would have gotten by the previous update, as follows:

for each *t* **in** *Employee*   (*C*)
  update *t*
  set *Salary* = (**select** *New*
           from *Employee E*1, *NewSal*
           where *E*1.*Eid* = *Manager* **and** *Old* = *E*1.*Salary*)

This solution is order dependent (and thus wrong) because we get different end results for the new salary of some employee, depending on whether or not we have already visited his manager. In the algebraic framework, this update is now modeled by the following algebraic update statement applied to each receiver [*t*(*EmpId*)] with *t* ∈ *Employee*:

$$Salary := \pi_{New}\big(self \underset{self=EmpId}{\bowtie} Employee \underset{Salary=Old}{\bowtie} NewSal\big) \qquad (C')$$

34   •   M. Andries et al.

Significantly, both algebraic updates (B') and (C') above are positive, and thus the algorithmic procedure from our Theorem 5.12 is able to discriminate correctly between update (B) being order independent and update (C) being order dependent.

Note that a correct, solution to the above-specified update problem is to use the following set-oriented statement:

update *Employee*
set *Salary* $=$ (**select** *New*
    from *Employee E*1, *NewSal*
    where *E*1.*Eid* $=$ *Manager* **and** *Old* $=$ *E*1.*Salary*)

This solution is correct, again because the changes are made only after all the new salaries are computed.

In effect, the algebraic update statement underlying the above SQL statement is extremely simple: it is simply *Salary* $:=$ arg_1, which is trivially key-order independent. The key set of receivers to which it is applied is computed by the SQL query

select *EmpId*, *New*
from *Employee*, *Employee E*1, *NewSal*
where *E*1.*Eid* $=$ *Manager* **and** *Old* $=$ *E*1.*Salary*

Thus we see that the set-oriented **update** statement in SQL is explained very nicely by our theory as the application of a trivial, key-order independent, modification update to a precomputed key set of receivers.

To conclude this section, we situate our parallel semantics for algebraic updates within the SQL context. Recall updates (A) and (B) above. Both have the same end result, but update (A) is much more efficient because it computes the changes to be made in one global query, which can be optimized and executed once. In contrast, update (B) performs a separate query for each individual tuple. Now recall that update (B) is key-order independent. Our Theorem 6.5 states that we can equivalently use the parallel semantics. Now the nice observation is that this parallel semantics corresponds to update (A). To see this, recall the algebraic update statement for (B):

$$Salary := \pi_{New}\left(arg_1 \underset{arg_1 = Old}{\bowtie} NewSal\right)$$

Then the parallel version of the expression on the right-hand side is

$$\pi_{self,New}\left(\pi_{self,arg_1}(rec) \underset{\substack{self=self \\ arg_1=Old}}{\bowtie} (\pi_{self}(rec) \times NewSal)\right).$$

Since, in this case, relation *rec* is nothing but the *Employee* relation, where *self* corresponds to *EmpId* and arg_1 to *Salary*, the above expression simplifies to

$$\pi_{EmpId,New}\left(Employee \underset{Salary = Old}{\bowtie} NewSal\right),$$

or in SQL,

select *EmpId*, *New*
from *Employee*, *NewSal*
where *Salary* = *Old*

This is exactly the key set of receivers computed by update (*A*)!

Thus we see that our Theorem 6.5 provides a "code improvement" tool which, given a cursor-based update program that is key-order independent and is equivalent to a set-oriented update statement, can automatically find this statement (which is much more efficient).

APPENDIX A

This section (included at the explicit request of one of the referees) is devoted to the proof of Lemma 5.13, the main result needed to characterize decidability of a number of problems in the context of positive update methods. Since positive expressions can be viewed as conjunctive queries extended with union and nonequality, we consider a generalization of the classical problem of containment of conjunctive queries, with three different extensions: (i) union, (ii) nonequality, and (iii) functional and full inclusion dependencies. Testing for containment of conjunctive queries is well-known to be decidable [Chandra and Merlin 1977; Aho et al. 1979], and extensions incorporating union [Sagiv and Yannakakis 1980], selection for inequalities [Klug 1988], or the presence of functional and inclusion dependencies [Johnson and Klug 1984] are equally well-known. However, we need to prove that these generalizations can be combined and that the comprehensive problem remains decidable. For more information on relational database theory, we refer the reader to Abiteboul et al. [1995].[11]

Before we proceed, we need to introduce some terminology and notation. In what follows, we fix a relational scheme $S = \{R_1, \ldots, R_n\}$. We also fix a finite set Σ of *functional* and *full inclusion dependencies*, of the following forms. Functional dependencies (*fd*) have the form $R_i : X \rightarrow A$, where X is a set of attributes and A is a single attribute. Full inclusion dependencies (*ind*) have the form $R_i[A_1 \ldots A_k] \subseteq R_j[B_1 \ldots B_k]$, where R_j is a k-ary relation, that is, $B_1 \ldots B_k$ is *exactly* the scheme of R_j.

The notion of *containment* and *equivalence*, (denoted $\subseteq$ and $\equiv$, respectively) of queries are as usual. For the set Σ of dependencies, we write $\subseteq_\Sigma$ ($\equiv_\Sigma$) to denote containment (equivalence) of queries *under* the set Σ. We say that $q \subseteq_\Sigma q'$ if $q(I) \subseteq q'(I)$ for every instance I that satisfies the dependencies in Σ. Similarly, $q \equiv_\Sigma q'$ if $q(I) = q'(I)$ for every instance I that satisfies Σ.

Since we are interested in querying relational database schemas corresponding to object-base schemas, we consider *typed* relational schemes and *typed* positive queries. Specifically, we assume that the database is defined with respect to a number of disjoint domains, and that each attribute of each relation is

---

[11]A result similar to Lemma 5.13, supporting only a weak form of union but allowing a weak form of negation, was presented by Chan [1992]. 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.

36   •   M. Andries et al.

associated with a domain. In writing queries, variables are associated with domains; variables of a domain Δ can only occur in the positions for the attributes of the same domain.

Positive expressions are defined as follows. For each domain Δ, we assume the existence of two pairwise disjoint sets V_Δ^d and V_Δ^u of *distinguished* and *undistinguished variables* associated with Δ. Moreover, we let $V_\Delta = V_\Delta^d \cup V_\Delta^u$ and define on it a total order $\preceq$ such that the variables in V_Δ^d always precede those in V_Δ^u. A *conjunctive query* q is specified by means of the functions s, d, u, v, c, and n, as follows.

—$s(q)$ is a finite tuple $\langle x_1 \ldots x_n \rangle$ of distinct distinguished variables, called the *summary* of q; we write $d(q)$ to denote the set $\{x_1, \ldots, x_n\}$;

—$u(q)$ is a finite set $\{y_1, \ldots, y_m\}$ of undistinguished (existentially quantified) variables; we write $v(q)$ to denote the set $d(q) \cup u(q)$;

—$c(q)$ is a finite set of *conjuncts*, each conjunct a literal of the form $R(z_1, \ldots, z_h)$, where R is a h-ary relation in S and, for each $1 \le i \le h$, the variable z_i is in both $v(q)$ and V_{Δ_i}, where Δ_i is the domain associated with the ith attribute of R;

—$n(q)$ is a finite set of *nonequalities*, each nonequality is of the form $z_i \neq z_j$, where z_i, z_j are variables both in $v(q)$ and in the same set V_Δ for a domain Δ.

Following Klug [1988], we say that q is an *equality* conjunctive query if $n(q) = \emptyset$ (no nonequalities in q). A *positive query* Q is a finite set of conjunctive queries having the same summary, interpreted as the union of these queries. The functions s, d, u, and v are extended in the natural way to positive queries.

The *result* of applying a conjunctive query q to an instance I, denoted $q(I)$, is defined as usual, referring here to "typed valuations." A *typed valuation θ for* q is a mapping from $v(q)$ to values, with the condition that variables in V_Δ are associated with values in the domain Δ. It is clear that a valuation contributes to $q(I)$ if and only if it allows satisfying the conjuncts in $c(q)$ and the nonequalities in $n(q)$. We say that a valuation θ for q *gets* the tuple $t = \theta(s(q))$ in $q(I)$ if: (i) for each conjunct $R(z_1, \ldots, z_h) \in c(q)$, it is the case that $\theta(z_1, \ldots, z_h) \in I(R)$; and (ii) for each $z_i \neq z_j \in n(q)$, it is the case that $\theta(z_i) \neq \theta(z_j)$. A similar terminology and notation can be used for a positive query Q.

The problem of equivalence of two positive queries Q_1, Q_2 is easily reduced to the problem of containment of a conjunctive query in a positive query. Indeed, $Q_1 \equiv_\Sigma Q_2$ if and only if $Q_1 \subseteq_\Sigma Q_2$ and $Q_2 \subseteq_\Sigma Q_1$. Moreover, if $Q_1 = q_1^1 \cup \ldots \cup q_1^k$ (where each q_i is conjunctive), then $Q_1 \subseteq_\Sigma Q_2$ if and only if $q_1^i \subseteq_\Sigma Q_2$ for $1 \le i \le k$. Hence we concentrate on this simpler problem.

First of all, we face the problem of union and nonequality.

The problem of testing containment of equality conjunctive queries was solved by Chandra and Merlin [1977]. Their homomorphism theorem says that, given two equality conjunctive queries q_1, q_2, it holds $q_1 \subseteq q_2$ if and only if there is a *homomorphism* from q_2 to q_1; that is, a mapping ψ from $v(q_2)$ to $v(q_1)$ such that $\psi(c(q_2)) \subseteq c(q_1)$ and $\psi(s(q_2)) = s(q_1)$. The intuition is that the conjuncts in q_1 can be seen as tuples in a "magic" canonical instance I_1 where each variable

corresponds to some (distinct) constant and its summary to a "magic" tuple t_1. Then, q_1 is contained in q_2 if and only if $t_1 \in q_2(I_1)$.

As pointed out by Klug [1988], the homomorphism theorem fails with respect to conjunctive queries with inequalities, since looking at a single canonical instance does not provide a correct test for containment. However, containment can still be decided by looking at a set of "representative" instances in place of a single one. Here we develop the framework with respect to nonequalities ($\neq$) rather than inequalities ($\leq$).

Consider a conjunctive query q and a valuation θ. We say that θ is *nonequality preserving for q* if, for each $z_i \neq z_j \in n(q)$, it is the case that $\theta(z_i) \neq \theta(z_j)$. Then two nonequality-preserving valuations θ_1, θ_2 for q are said to be *equivalent* if, for each pair z_i, z_j of variables in $v(q)$, it is the case that $\theta_1(z_i) = \theta_1(z_j)$ if and only if $\theta_2(z_i) = \theta_2(z_j)$. By choosing an arbitrary representative from each equivalence class, we obtain a set Θ_q of *representative* nonequality-preserving valuations for q. Note that, if n is the number of distinct variables in $v(q)$, it is possible to define the set Θ_q referring to just n distinct values from the corresponding domains. Hence, only a finite number of different valuations have to be considered. Once this set is chosen, the *representative instances for q* are the "magic" instances given by $\theta(c(q))$, one for each $\theta \in \Theta_q$. The *representative set $r(q)$ for q* is the following set of representative instance-tuple pairs:

$$r(q) = \{(\theta(c(q)), \theta(s(q)) \mid \theta \in \Theta_q)\}.$$

THEOREM A.1 (KLUG [1988]). *Let q_1, q_2 be two conjunctive queries with nonequalities. Then, $q_1 \subseteq q_2$ if and only if $s \in q_2(I)$ for each pair (I, s) in the representative set $r(q_1)$ of q_1.*

The problem considered by the above theorem is decidable, since it requires evaluating only a finite number of conjunctive queries.

Sagiv and Yannakakis [1980] considered the problem of testing containment of equality positive queries (that is, unions of equality conjunctive queries), proving that equality conjunctive queries are contained in a trivial way only, that is, that an equality conjunctive query q is contained in an equality positive query Q if and only if there is an equality conjunctive query $q' \in Q$ such that $q \subseteq q'$. This implies that a single "magic" canonical instance suffices for testing containment in this case. In the presence of inequalities as well, Klug [1988] proved that this technique can be combined with that of the representative set. Specifically, he proved that, given a conjunctive query q and a positive query Q, $q \subseteq Q$ holds if and only if, for each pair (I, s) in the representative set $r(q)$ of q, there is a query $q' \in Q$ such that $s \in q'(I)$. Again, the problem remains decidable.

We now consider the management of dependencies in this framework. The technical tool we use is a *typed chase* process: the standard chase process [Maier et al. 1979; Aho et al. 1979; Johnson and Klug 1984] accounts for functional and inclusion dependencies, while a typed management of variables accounts for the disjointness of domains. The process of chasing a query consists in successive modifications to its conjuncts: intuitively, the canonical instance associated with the query is modified to "enforce" the satisfaction of the dependencies. In the presence of nonequalities, a contradiction is sometimes reached (e.g., a

38   •   M. Andries et al.

nonequality of the form $z \neq z$), meaning that the query is unsatisfiable over instances satisfying the dependencies; this fact is denoted $\perp$. The *chase* is based on the successive applications of the following rules.

fd rule. Let $\sigma = R : X \rightarrow A$ be a functional dependency over R, and let $R(u), R(v)$ be conjuncts in $c(q)$ such that $u[X] = v[X]$ and $u[A] \neq v[A]$. Let x be the least variable in $\{u[A], v[A]\}$ under the ordering $\preceq$, and y be the other one. Call θ the substitution that maps y to x and is the identity elsewhere. The *result of applying σ to $R(u), R(v)$ in q* is the query $\theta(q)$ if $x \neq y \notin n(q)$, and $\perp$ otherwise.

ind rule. Let $\sigma = R[X] \subseteq S[Y]$ be a full inclusion dependency over R, let $R(u)$ be a conjunct in $c(q)$, and suppose that $c(q)$ does not contain the conjunct $S(v)$, where $v = u[X]$. The *result of applying σ to $R(u)$ in q* is the query q' such that $s(q') = s(q)$, $u(q') = u(q)$, $n(q') = n(q)$, and $c(q') = c(q) \cup \{S(v)\}$.

We denote the result of chasing a conjunctive query q with respect to a set Σ of dependencies by $chase_\Sigma(q)$. It is worth noting that the chase process, with respect to functional and full inclusion dependencies, always terminates. Moreover, the process satisfies the Church-Rosser property, meaning that the results of different terminal chasing sequences are identical [Abiteboul et al. 1995]. Finally, we note that, given a valuation θ for a conjunctive query q, it is the case that $\theta(c(chase_\Sigma(q)))$ represents an instance that satisfies the dependencies in Σ.

Our main result, concerning the containment of a conjunctive query in a positive query under a set of dependencies, relies on the two following lemmas. Intuitively, we want to reduce the problem of containment constrained by a set of dependencies to an unconstrained problem of containment, eventually referring to chased queries. The following lemma specializes a result by Johnson and Klug [1984] to functional and *full* inclusion dependencies, but also generalizes it to conjunctive queries containing nonequalities.

LEMMA A.2.   *Let q be a conjunctive query and Σ be a set of functional and full inclusion dependencies. Then, $q \equiv_\Sigma chase_\Sigma(q)$.*

PROOF.   We proceed by induction on the length n of a terminal chasing sequence on q with respect to Σ. In such a sequence, we denote by $chase_\Sigma^i(q)$ the partial chase obtained after the ith application of a chase rule. We claim that, for $1 \leq i \leq n$, it is the case that $chase_\Sigma^{i-1}(q) \equiv_\Sigma chase_\Sigma^i(q)$ (where $chase_\Sigma^0(q) = q$ and $chase_\Sigma^n(q) = chase_\Sigma(q)$).

The induction hypothesis clearly holds for $n = 0$ (meaning $chase_\Sigma(q) = q$). Suppose it holds for $chase_\Sigma^{i-1}(q)$. There are two cases, depending on whether the ith chase rule application involved an fd or an ind.

Suppose it was the fd $\sigma = R : X \rightarrow A$ to $R(u), R(v)$, with $u[X] = v[X]$ and $u[A] \neq v[A]$. Let x be the least variable in $\{u[A], v[A]\}$ under the ordering $\preceq$ and y be the other one. Call θ a substitution that maps y to x and is the identity elsewhere. Then, if $x \neq y \in n(chase_\Sigma^{i-1}(q))$, then $chase_\Sigma^i(q)$ was the unsatisfiable query $\perp$, otherwise it was the query $\theta(chase_\Sigma^{i-1}(q))$. In the former case, it can be shown that $chase_\Sigma^{i-1}(q)$ is unsatisfiable as well on instances that satisfy Σ, and

the equivalence holds. In the latter case, consider an instance I that satisfies Σ and a valuation ν that gets a tuple t in $chase_\Sigma^{i-1}(q)(I)$. Then, since $\nu(u), \nu(v) \in I(R)$ and I satisfies σ, it is also clearly the case that $\nu(\theta(u)), \nu(\theta(v)) \in I(R)$, and hence ν gets the tuple t in $chase_\Sigma^i(q)(I)$ as well. To prove the converse inclusion, consider a valuation ν that gets a tuple t in $chase_\Sigma^i(q)(I)$; it is then clear that the valuation ν' obtained from ν by also mapping y to $\nu(x)$ gets t in $chase_\Sigma^{i-1}(q)(I)$.

On the other hand, suppose that the ith chase rule application involved the ind $\sigma = R[X] \subseteq S[Y]$ to $R(u)$, and let v be the tuple over S such that $v[Y] = u[X]$. In this case, $chase_\Sigma^i(q)$ has been obtained from $chase_\Sigma^{i-1}(q)$ by introducing the conjunct $S(v)$. Now consider an instance I that satisfies Σ and a valuation ν that gets a tuple t in $chase_\Sigma^{i-1}(q)(I)$. Since $\nu(u) \in I(R)$ and I satisfies σ, it is also clearly the case that $\nu(v) \in I(S)$, and hence ν gets t in $chase_\Sigma^i(q)(I)$ as well. To prove the converse inclusion, consider a valuation ν that gets a tuple t in $chase_\Sigma^i(q)(I)$; the same valuation ν can be used to get t in $chase_\Sigma^{i-1}(q)(I)$. $\square$

LEMMA A.3. *Let q be a conjunctive query, Q a positive query, and Σ a set of functional and full inclusion dependencies. Then, $q \subseteq_\Sigma Q$ if and only if $chase_\Sigma(q) \subseteq Q$.*

PROOF. The *if* part follows from Lemma A.2. For the converse inclusion, by Theorem A.1, it suffices to show that, for each pair (I, s) in the representative set $r(chase_\Sigma(q))$, it is the case that $s \in Q(I)$. Consider the pair (I_ν, s_ν) obtained by a valuation $\nu \in \Theta_{chase_\Sigma(q)}$. It is clear that I_ν satisfies Σ. By the assumption, $q(I_\nu) \subseteq Q(I_\nu)$, and hence $s_\nu \in Q(I_\nu)$. $\square$

Lemma 5.13 now follows. Indeed, Lemma A.3 suggests the reduction from the problem of containing a conjunctive query q in a positive query Q under a set of dependencies Σ to the problem of containing a conjunctive query in a positive query. The latter problem is decidable by Theorem A.1. The observation that $chase_\Sigma(q)$ is indeed computable, since Σ contains only a finite set of functional dependencies and full inclusion dependencies [Abiteboul et al. 1995], concludes the proof.

REFERENCES

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison-Wesley.

ABITEBOUL, S. AND VIANU, V. 1990. Procedural languages for database queries and updates. *J. Comput. Syst. Sci. 41*, 2, 181–229.

AHO, A., SAGIV, Y., AND ULLMAN, J. 1979. Equivalences among relational expressions. *SIAM. Comput. 8*, 2, 218–246.

AHO, A. AND ULLMAN, J. 1979. Universality of data retrieval languages. In *Proceedings of the Conference Record, 6th ACM Symposium on Principles of Programming Languages* (1979), 110–120.

BREAZU-TANNEN, V., BUNEMAN, P., AND NAQVI, S. 1992. Structural recursion as a query language. In *Database Programming Languages: Bulk Types and Persistent Data*. P. Kanellakis and J. Schmidt Eds., Morgan Kaufmann, 9–19.

BREAZU-TANNEN, V. AND SUBRAHMANYAM, R. 1991. Logical and computational aspects of programming with sets/bags/lists. In *Automata, Languages, and Programming*, LNCS 510, Springer, 60–75.

40   •   M. Andries et al.

CABIBBO, L. 1996. Querying and updating complex-object databases. Ph.D. thesis, Università di Roma "La Sapienza."

CHAN, E. 1992. Containment and minimization of positive conjunctive queries in OODB's. In *Proceedings 11th ACM Symposium on Principles of Database Systems*, 202–211.

CHANDRA, A. 1981. Programming primitives for database languages. In *Proceedings of the Conference Record, 8th ACM Symposium on Principles of Programming Languages*, 50–62.

CHANDRA, A. AND MERLIN, P. 1977. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th ACM Symposium on the Theory of Computing*, 77–90.

HOPCROFT, J. AND ULLMAN, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

HULL, R. AND SU, J. 1989. On accessing object-oriented databases: Expressive power, complexity, and restrictions. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, J. Clifford, B. Lindsay, and D. Maier Eds. *SIGMOD Rec. 18*, 2 147–158.

HULL, R. AND YOSHIKAWA, M. 1990. ILOG: Declarative creation and manipulation of object identifiers. In *Proceedings of the 16th International Conference on Very Large Data Bases* D. McLeod, R. Sacks-Davis, and H. Schek Eds., 455–468. Morgan Kaufmann.

JOHNSON, D. AND KLUG, A. 1984. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci. 28*, 167–189.

KLUG, A. 1988. On conjunctive queries containing inequalities. *J. ACM 35*, 1, 146–160.

LAASCH, C. AND SCHOLL, M. 1993. Deterministic semantics of set-oriented update sequences. In *Proceedings of the Ninth International Conference on Data Engineering*, IEEE Computer Society Press, 4–13.

LYNGBAEK, P. AND VIANU, V. 1987. Mapping a semantic database model to the relational model. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*, U. Dayal and I. Traiger Eds. *SIGMOD Rec. 16, 3* 132–142.

MAIER, D., MENDELZON, A., AND SAGIV, Y. 1979. Testing implications of data dependencies. *ACM Trans. Database Syst. 4*, 455–469.

QIAN, X. 1991. The expressive power of the bounded-iteration construct. *Acta Informatica 28*, 631–656.

SAGIV, Y. AND YANNAKAKIS, M. 1980. Equivalence among relational expressions with the union and difference operators. *J. ACM 27*, 4, 633–655.

SCHWARTZ, J. ET AL. 1986. *Programming with Sets: An Introduction to SETL*, Springer.

SIMON, E. AND DE MAINDREVILLE, C. 1988. Deciding whether a production rule is relational computable. In *ICDT'88*, LNCS 326 M. Gyssens, J. Paredaens, and D. Van Gucht Eds. Springer, 205–222.

ULLMAN, J. 1988. *Principles of Database and Knowledge-Base Systems*, Vol. I. Computer Science Press.

VAN DEN BUSSCHE, J. AND CABIBBO, L. 1998. Converting untyped formulas into typed ones. *Acta Informatica 35*, 8, 637–643.