In: *Proc. 9th Int'l Conf. on Data Engineering (ICDE), Vienna, Austria, April, 1993, pages 4–13.*

Deterministic Semantics of Set-Oriented Update Sequences

Christian Laasch, Marc H. Scholl

Department of Computer Science, Databases and Information Systems
University of Ulm, D-W 7900 Ulm, Germany
e-mail: {laasch, scholl}@informatik.uni-ulm.de

Abstract

An iterator is proposed that allows to apply sequences of update operations in a set-oriented way with deterministic semantics. Because the mechanism is independent of a particular model, it can be used in the relational and in object-oriented ones. Thus, the deterministic semantics of embedded SQL cursors, and of triggers that are applied after (set-oriented) SQL updates can be checked. Furthermore, the iterator can be used to apply object-oriented methods, which are usually update sequences defined on a single object, also to sets in a deterministic way. It turns out that the criteria that guarantee determinism are also used in semantic or multi-level concurrency control.

1 Introduction

The paper deals with the general problem of defining update languages that are comparable in expressive power to typical query languages. When designing such an update language, one should pursue the following objectives:

• **genericity:** update operations should be applicable to all types of objects in a database schema (in contrast to type-specific updates, e.g., methods in object-oriented models).

• **safety:** update operations should respect model-inherent constraints.

• **determinism:** the effects of updates should be clearly (formally) defined in a deterministic way.

• **set-orientation:** like retrievals, we should be able to issue bulk-updates. On the one hand, this makes update operations applicable to query results or named collections (such as relations or classes), thus eliminating the gap between *one–tuple/object–at–a–time* updates and *set–oriented* queries. On the other hand, set-oriented updates allow for more efficient update processing (for example, parallel execution of updates), because the system gets more flexibility in executing the update request.

• **orthogonality:** update operations should be combinable into complex sequences in order to realize non-trivial requests. These update sequences can be used to define more powerful update units that respect (application-specific) integrity constraints. Examples for update sequences are *methods* in object-oriented models or any update operation with its *triggered* operations in commercial relational systems. The orthogonality of sequences and set-orientation allows to apply update sequences not only to single objects, but also to arbitrary sets of objects.

• **ad-hoc usability:** The possibility to use update operations in an interactive way, similar to queries and SQL updates, simplifies database handling, because simple updates need not be coded into a host language program (as was the case in CODASYL systems).

In this paper, we focus on the problem of combining determinism and set-orientation with orthogonality. We analyze the well-known problems in combining set-orientation and updates in a deterministic way and try to give a general, model-independent solution. Afterwards, the proposed solution can be combined with elementary update operations that are defined with respect to genericity, safety, and ad-hoc usability (see for example [20, 23]).

Section 2 reviews known problems with set-oriented updates in the relational context. It is shown how this generalizes to object-oriented models, and that the problem is due to sharing. In Section 3, we develop a generic set-iterator for update operations that allows bulk-updates with sequences of update operations. It turns out that set-oriented updates can only be given deterministic semantics under certain restrictions: there must not be conflicts. This notion of conflicts is essentially the same as the one in semantic concurrency control, an analogy that is elaborated in Section 3.2. Finally, we give an example application of the framework for the object-oriented model BCOOL in Section 4 before we conclude.

2 Problems with set-oriented updates

In this section, we discuss the update facilities of the relational, and some object-oriented data models. We argue that none of these models offer set-oriented update sequences with deterministic semantics. In addition, we show that most of the problems with set-oriented update sequences originate in sharing of data (i.e., tuples or objects).

Let us first focus the update operations that can be used in the interactive mode of relational system. All operations (**delete**, **insert**, and **update**) can be applied in a set-oriented way to relations or query results. However, sequencing and set-orientation are not orthogonal: sequences of set-oriented updates can be expressed, but it is not possible to define update sequences that are executed for all elements of a set.

With respect to deterministic semantics, the most interesting updates in SQL are so called *self-referencing* updates. These are update operations that change tuples depending on their values. Let us consider, for example, the following update that sets the salary of all employees in the relation *Emp* to the average salary of all employees:

update *Emp* **set** *sal* = **select avg**(*sal*) **from** *Emp*;

Since salaries of employees are used to compute new salaries, it is crucial in which state the retrieval operation on the right hand is evaluated. If it is evaluated, for each employee, immediately before the assignment, the new salary depends on the execution order (because the global average changes with each assignment). Therefore, the solution in the SQL2 standard [18] is to evaluate all retrieval expressions in the update statement in the old database state (that is, **avg**(*sal*) is logically evaluated before any update takes place)[1]. According to this semantics, the values of attributes can be swapped by the following update, because both assignments are logically executed simultaneously, rather than as a sequence of assignments:

update *Table* **set** $a = b, b = a$;

However, this approach can not be adopted to *sequences* of update operations. In *sequences*, retrieval expressions might occur in different statements of the sequence. Evaluating all retrieval expressions in advance would contradict our intention that one statement of a sequence is applied after the other.

In embedded SQL, the cursor concept can be used to implement set-oriented updates. Additionally, it is possible to execute a sequence of retrieve and update operations within the loop that is executed for each element of the set. Thus, set-oriented update sequences can be defined. However, this is no clear deterministic semantics of such updates. In fact, it is possible to specify non-deterministic updates as illustrated in Example 1.

[1] Up to now, in most of the current SQL implementations this evaluation order is explicit such that the above update must be split into two operations.

Example 1 Let us consider the relation *Emp* with the following three tuples:

Emp	eno	sal	mgr
	e_1	40K	m_1
	e_2	60K	m_1
	m_1	*	*

The update sequence consists just of a single update operation that is executed for each employee e and sets the salary of e's manager to e's salary:

```
declare e cursor for select sal, mgr from Emp
                     where sal < 100K;
open e;
loop
  fetch e into :sal, :mgr;
  update Emp set sal = :sal where eno = :mgr;
endloop
```

In this case ambiguities arise, because e_1 and e_2 share the same manager m_1 but have different salaries: The salary of the manager is assigned to different values in different iterations of the loop. ◇

Thus, the result of the set-oriented update depends on the order of the set iteration. This problem originates in *sharing*, because multiple, inconsistent updates are applied to the *same* data item.

Similarly, we have to cope with the retrieval of a data item in one iteration of a sequence and an update to the same data item in another iteration of a set-oriented update:

Example 2 Assume that the relation *Emp* contains the additional attribute *bdg* containing the budget. Let us apply the following update sequence to all employees, in order to raise the salary and to initialize the budget by adding the new salary and the salaries of the subordinate employees:

```
declare e cursor for select eno, sal from Emp;
open e;
loop
  fetch e into :eno, :sal;
  update Emp set sal = 1.05 * :sal
  where eno = :eno;
  update Emp set bdg = :sal + (select sum(sal)
     from Emp where mgr = :eno)
  where eno = :eno;
endloop
```

Of course, this example could be syntactically simplified using for example the **current of** construct of Ingres [22] in addition to the *direct* update mode of the **open** cursor statement. Nevertheless, if we apply the update to the following relation

Emp	eno	sal	mgr	bdg
	e_1	40K	m_2	*
	m_2	60K	m_1	*
	m_1	80K	*	*

the budget of the manager m_1 depends on the execution order: If m_1 is updated before m_2, the old salary of m_2 is used to evaluate the budget. In case that m_2 has already been updated, m_1's budget gets higher. $\diamond$

Notice that the same argument holds for triggers and rules executing statements for each tuple that has been inserted, deleted, or updated. Such triggers and rules have been integrated into commercial relational systems in order to allow checking of application-specific integrity constraints by query and update sequences that are executed after an update operation. In both cases set iteration is independent of the executed operations: The iterator of a trigger is specified by the set-oriented update operation, on which the trigger is defined. This corresponds to a cursor, which is defined on a query expression. The trigger procedure, which is the body of the set iteration, can manipulate arbitrary data within cursor loops. Thus, the lack of orthogonality has been alleviated, but at the expense of deterministic semantics.

As shown in the above examples, non-deterministic update semantics arises by sharing of data items. This problem does not occur for the interactive relational update operations, since these operations are restricted to be applied exactly to the tuples retrieved by the set: First, since only the elements of the set can be manipulated, no multiple updates to the same data item can occur. Secondly, because sequences are not allowed within the set iteration, all retrieval expressions can be evaluated in advance, which prevents conflicts between retrievals and updates on the same data item. In case of embedded SQL or triggers, the execution of update operations is decoupled from the set iteration. This allows updating and retrieving other data than the elements of the set.

Consequently, if we allow to update objects that are not identified by the set expression itself, we have to cope with object sharing. Otherwise, we have to restrict the update operations.

In object-oriented models, updates are usually carried out by applying methods (see [3, 6]). That is, integrity-preserving updates are offered to clients. In most object-oriented models (see e.g., [8, 14, 21]) these updates are built upon predefined C++ or Smalltalk functions (like *create, destroy*) and assignments, which are usually applied like in programming languages: one–object–at–a–time. In order to apply methods to sets, iterators on sets can be used. However, since these iterators are not necessarily defined with respect to sharing problems, the above discussion of the embedded SQL operations still holds.

Some object-oriented models (such as IRIS [31]) use SQL extensions as query and update language. Thus, similar to the interactive update operations of the relational model, set-oriented updates can be executed, but they can not consist of sequences of elementary update operations.

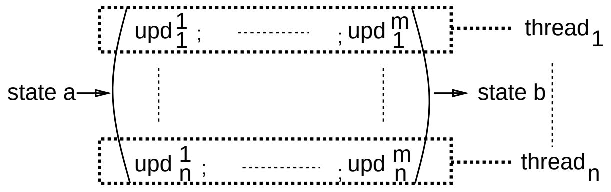

Figure 1: Transformation of state a to b by the update **apply_to_all** $[upd^1; ...; upd^m](\{o_1, ..., o_n\})$

3 Formalizing set-oriented updates

In this section we first introduce a set iterator that gives deterministic semantics to set updates. The proposed mechanism copes with object sharing: ambiguities such as multiple inconsistent updates or retrieving and updating the same objects are prohibited. Then we compare our approach with semantic or multi-level concurrency control.

3.1 Definition of a set iterator

In order to define the iterator (called **apply_to_all**) independent of a special model, we only require that an update is regarded as a transformation of states: That is, we assume the existence of a function U that maps a state σ according to an update operation *upd* onto a new state $U[\![upd]\!]_\sigma$. Following [13], we call the application of update sequence to each individual element of the set a *thread* (illustrated as a row in Fig. 1) and the particular element the *thread object.* The j-th operation in an update sequence is denoted as upd^j, and the j-th operation within $thread_i$ (that is associated to the thread object o_i) as upd_i^j.

The core problem of formalizing set-oriented update sequences is to collect the state transformations of all threads. The standard semantics for an iterator **apply_to_all** (also called *filter, map,* or *replace* [1, 7]) is the union of the operations applied to each element of the input set: In our case, the union of state transitions performed by applying the thread $ups(o)$ to each element o, in the initial state σ [2]:

$$U[\![\mathbf{apply\_to\_all}[ups](set)]\!]_\sigma = \bigcup_{o \,\in\, E[\![set]\!]_\sigma} U[\![ups(o)]\!]_\sigma$$

In general, unions are only adequate for "increasing" state transitions (e.g., creation of objects). Assignments would have to be expressed by the *difference* between the state before and after the update, or by (recursively) overriding unions. For example, consider a state where $f(p) = p'$, and $f(q) = q'$, and the update operation $\mathbf{apply\_to\_all}\,[\mathbf{set}\,[f := o](o)](o : \{p, q\})$, which sets the function f to the identity function for the objects p, q. If we look at the final states of both threads separately, one contains the pairs $\langle p, p\rangle$ and $\langle q, q'\rangle$ and the other one the pairs $\langle p, p'\rangle$ and $\langle q, q\rangle$. Therefore, the union of these sets would not be a function in the first argument anymore, because the union of both states would contain pairs with a common first component. Therefore, we need a sophisticated definition that merges all changes of the different threads. On the other hand, operations such as *delete* need something like the *intersection* of states, since we remove information. For example, consider the deletion of the two objects p and q, in a state where only p, and q exist: $\mathbf{apply\_to\_all}\,[\mathbf{delete}\ (o)](o : \{p, q\})$. According to the above definition, the value of $U[\![\mathbf{delete}\ (p)]\!]_\sigma$ contains q but not p and the value of $U[\![\mathbf{delete}\ (q)]\!]_\sigma$ includes p but not q, vice versa. The *union* of these states would still contain both objects, p and q.

Thus, we conclude that merging the states needs different mechanisms depending on the update operation. Such a differentiation, however, would lead to the definition of set-oriented elementary update operations, but not to a definition of an iterator independent of the elementary update operations. This is one explanation for the SQL approach, where only set-oriented *single* updates are possible. Furthermore, it shows that set-orientation and updates do not go together too well. We now follow another approach to "set-oriented" updates.

Since it seems impossible to combine threads uniformly using set-operations, we take the freedom to define a "set-oriented" update as one where the order of application to the elements of the set is not significant. That is, we could sequentialize in any order (e.g., depending on execution plans) or we could even parallelize. Hence, our formalization of set-oriented update sequences is based on the intuition that several objects can be manipulated in one operation, if the effects of different threads are *independent* of each other. Then, the final state of the set-oriented update sequence does not depend on the execution order of the threads. That is, all possible execution orders yield the same result. Therefore, the semantics of **apply_to_all** can be defined as the composition of threads, if all threads commute. Commutativity is guaranteed, if the elementary update operations do not conflict with each other, which can be derived by a model-specific conflict relation containing the dependencies between the elementary update operations. Let us now present this idea in more detail.

Formally, we define the semantics of the iterator in three levels. The highest level defines the semantics of a set-oriented update depending on the commutativity of its threads; the second level defines commutativity of threads depending on the conflict relation; and the lowest level defines the conflict relation according to data model specific operations:

Def. 1 (Set-oriented updates) If and only if all threads of the set-oriented application of an update sequence $ups \equiv upd^1; \ldots; upd^m$ are pairwise commutative, $\mathbf{apply\_to\_all}[ups](set)$ is defined by the composition ($\prod$):

$$U[\![\mathbf{apply\_to\_all}\,[ups](set)]\!]_\sigma = \prod_{o \,\in\, E[\![set]\!]_\sigma} U[\![ups(o)]\!]_\sigma .$$

Notice that this pairwise commutativity always results in executions with no conflicts; that is, *all* sequential executions of threads are equivalent.

Def. 2 (Commutativity of threads) Two threads $thread_i$, $thread_j$ are commutative if, in every possible database state, no elementary update operation upd_i^k conflicts with any upd_j^l ($k, l \in [1..m]$ where m is the number of operations in the update sequence).

The conflicts between elementary update operations have to be defined by a conflict relation for the update operations on the particular underlying data model. As an example, this is done for the generic update operations of the BCOOL model in Section 4. Because this is the only place, where model-specific information is used, the **apply_to_all** iterator can be used for arbitrary models, provided that a conflict-relation is given.

Once a set-oriented update sequence passed the semantics test, there is no need to execute the threads in a serial way. Instead, each thread can be executed as a parallel (sub)transaction, provided we have this choice in our DBMS. Thus, we can easily take advantage of intra-transaction parallelism for the execution of such updates as described in [17, 30].

[2] Since the value of an expression depends on the state in which the expression is evaluated, we use, like in denotational semantics [27], the function E that returns the evaluated value $E[\![expr]\!]_\sigma$ of the expression $expr$ at the current state σ.

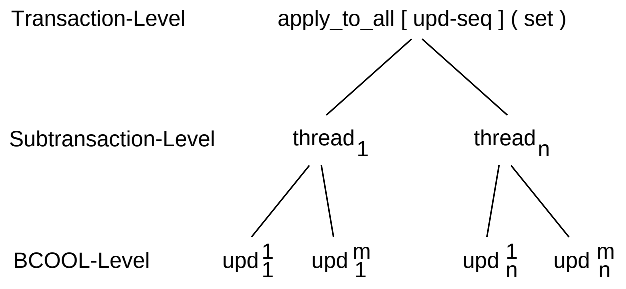

Figure 2: Mapping set-oriented update sequences to multi-level transactions

3.2 Relationship to concurrency control

There are strong similarities between our mechanism for set-oriented update sequences and semantic or multi-level concurrency control (see [11, 15, 28, 29]). In fact, the notions of conflict and commuting operations are exactly the same. In semantic concurrency control (SCC), commuting operations are said not to be in conflict with each other, that is, they can be admitted in parallel. Thus, the **apply_to_all** mechanism as well as semantic concurrency control use the same conflict relation. Therefore, one can think of the set-oriented update sequence **apply_to_all** [*ups*](*set*) as an operation on the transaction level that consists of one subtransaction for each thread, in which the elementary update operations are applied sequentially (see Fig. 2).

However, there are three fundamental differences as to how conflicts are used in SCC as opposed to set-oriented updates: (1) SCC schedules operations of *different* transactions, whereas the **apply_to_all** mechanism controls the semantics of operations within *one single* transaction; (2) in SCC we look for serializable schedules: i.e., the conflict graph has to be acyclic. The **apply_to_all** mechanism, however, demands the pairwise commutativity of all threads so as to guarantee deterministic semantics. Therefore, the conflict graph must be empty. If this restriction is weakened, such that the conflict graph is not empty but only acyclic, the execution order of the threads becomes important. That is, the order of choosing elements of the set would influence the semantics of the set-oriented update, which results in non-deterministic update semantics. (3) in contrast to SCC, which takes place at runtime, the **apply_to_all** mechanism is used as a *static* criterion for defining update semantics. Therefore, the conflict relation between operations is defined independent of the database state. Thus, the conflicts that are derived at compile-time need not necessarily lead to real conflicts; instead they are *potential conflicts* that might or might not occur dependent on the current database state.

In comparison to SCC, where the burden of defining conflicts between the model-specific generic update operations (if any) as well as the methods specified in a database schema is imposed on the type implementor, our approach alleviates this burden, because conflicts can be derived automatically. Thus, the type implementor can focus on cases where potential conflicts between methods have been derived by our approach. In case that these methods semantically commute, nevertheless, he/she can change the conflict relation. Then, the methods can be applied in a set-oriented way even though potential conflicts have been derived. This reflects the difference between the static criterion for commutativity according to the conflict relation, which is state–independent, and SCC, which may exploit state–dependent commutativity.

3.3 Coping with potential conflicts

Because the conflict relation is state-independent, it describes *potential conflicts*. These are real conflicts if sharing occurs, which can not be checked without information on the state. In order to reduce the number of potential conflicts, additional information about the semantics of the application could be used. This information might be given by the type implementor (such as commutativity of methods as described above) or as part of the schema definition. For example, if functions are defined as unique, sharing cannot occur by the application of this function to a set. The same is true for functions that yield (non-shared) subobjects, which are often used in models where an internal state is related to an object.

In case that "real" conflicts arise at runtime, although the type implementor ensured commutativity, two alternatives can be offered: Either the operation **apply_to_all** is not executed (i.e., rollback of all threads), or the operation is executed with non-deterministic (run–time dependent) semantics; e.g. with the semantics determined by the serialization of parallel (sub)transactions. Notice, however, that both alternatives require information from the concurrency control component at run-time. Therefore, it would be necessary that the (dynamic) conflict graph is made visible to the update processor, or that at least the scheduler can be asked whether conflicts occurred.

4 Application to an OO model

In this section we integrate **apply_to_all** into an object-oriented data model, called BCOOL. Therefore, we first sketch the basic concepts of the model and its elementary query and update operations. Afterwards, we discuss the conflict relation of these update operations.

4.1 The BCOOL model

BCOOL serves as the formal basis of COOL, the database language of the COCOON model [24, 25]. The key objectives in designing BCOOL have been the following: *static type-checking* in order to improve the efficiency by making use of type inference; *hiding object-identity* to avoid similar problems as they has arisen with database keys in CODASYL systems. Thus, BCOOL consists of only a few concepts, which can be found in (almost) all object-oriented models:

Objects are fully encapsulated. They can be used and manipulated only by means of their interface, a set of functions.

Data are instances of concrete types (such as numbers, strings) and constructed types (such as sets). The distinction from objects is the same as in [10].

Functions are described by a name and signature, they are the interface operations of type instances. We can distinguish retrieval (stored or computed) and update functions. Generic update operations can only be used for stored retrieval functions, which are uniform abstractions of "attributes" and "relationships" of classical data models, since directly updating derived or computed functions requires type-specific methods. Indirect updates (i.e., updates to values used in the derivation) are automatically propagated.

Types describe the common interface of all instances of that type, the set of applicable functions. A type definition normally consists of two parts: a set of functions and a type name. The following example defines an abstract object type *Pers* with the functions *name, age,* and the set–valued function *children*:

type *Pers* **isa** *Object* = *name* : **string**,
age : **integer**,
children : **set of** *Pers*;

Subtyping. If a type is defined as a *subtype* of another, then every instance of the subtype is also an instance of its supertype. Subtyping between abstract object types is defined by the inclusion of the sets of applicable functions. For example,

type *Empl* **isa** *Pers* = *sal* : **integer**,
courses : **set of string**;

defines the *Empl* type as a subtype of *Pers*: the functions of *Pers* ({*name, age, children*}) are inherited by *Empl* (which has {*name, age, children, sal, courses*} as applicable functions). The root of the type hierarchy is *Object*, without any user-defined function.

Variables. Variables are used as temporary names ("handles") for instances of any type, i.e., data (e.g. integer), objects, sets of any type, or functions. They have to be declared with their type in the database language, for compile-time type checking. For example,

var *kids* : **set of** *Pers*,

declares a variable of type **set of** *Pers*. It can, for example, keep the result of a selection on persons (see below).

4.2 Query operations and expressions

We use a set-oriented algebra, where the inputs and outputs of the operations are sets of objects. Hence, query operators can be applied to all instances of a type T (denoted as the active domain of type: $ad(T)$), to set-valued function results, to query results, or to set variables. The effects of each operator are described separately for type and value. (Only the set operations $\cup$, and $\cap$ have an effect on both.)

Selection (**select** $[P](set)$) returns a subset of the input set of objects denoted by *set*, namely those satisfying the predicate P. The type of the set is unchanged.

Projection (**project** $[f_1, ..., f_n](set)$). The output of a projection is a set with a usually new type, a supertype of the input type: fewer functions are defined, namely only those listed in the projection. All objects of the input set *set* are also elements of the output set (*object preservation*).

Extend (**extend** $[f_1 := expr_1, ..., f_n := expr_n](set)$). Projection eliminates functions, extend defines new derived ones. Obviously, each function name f_i must be different from all existing functions of the input's type. The expression $expr_i$ can be any legal arithmetic-, boolean-, or set-expression. The result set contains exactly the same objects as the input, but a new type, a subtype of the input type, is associated to it (all the old functions plus the new ones are defined on it). Notice, however, that derived functions can not be **set** by generic update operations, but updated only by methods and generic updates to the values from which functions are derived.

Set operations. We provide the usual set operations ($\cup$, $\cap$, and $\backslash$). With a polymorphic type system, we need no restrictions on operand types of set operations (ultimately, they are all objects). The result type, however, depends on the input types: for the union ($\cup$) it is the lowest common supertype (in the lattice) of the input types. The intersection ($\cap$) results in the greatest common subtype; finally, difference ($\backslash$), which can be reduced to a selection, yields a subset of its first argument with the same type.

Pick (**pick** (set)) is provided to convert a singleton set into the only element. The result type is the element type of the set.

These are the basic *object preserving* query operators of our algebra. Other operators, such as join can be derived from them [25, 26].

4.3 Update operations

There are the following three groups of update operations (a formal definition can be found in [20]:

Methods are defined by combining the generic query and update operations. They allow to specify more powerful updates with respect to application specific integrity constraints.

Assignments ($:=$ and **set**) for changing values of variables and functions. Besides global assignments to variables (e.g., $v := expr$) it is possible to assign function values for given arguments. For example, a salary can be assigned to an *Empl* object denoted by the variable e:

set $[sal := 30000](e)$;

We call this "partial assignment", because it has no effect on the salary of other employees.

Operations for object evolution: besides being **create**d and **delete**d, objects might also **gain** or **lose** types dynamically, (similar to *add/remove type* in the IRIS model [9], and to *specialize* in [16]). For example, we create a new instance of type *Pers* by:

create $[Pers](p)$;

Here, p has to be a variable of type *Pers* or a supertype. The object type is instantiated and the new object is assigned to the variable, which can then be used in partial assignments, for example. After a while, the person denoted by the variable p might be hired. The following update makes him/her an instance of the *Empl* type, such that the additional functions *sal* and *courses* become applicable:

gain $[Empl](p)$;

Conversely, retirement can be expressed by

lose $[Empl](p)$,

which makes the functions *sal* and *courses* inapplicable again. Notice however, that due to object sharing the retired person still might occur in variables, sets or function values that are associated with the type *Empl* (or a subtype of it). In order to make the functions *sal* and *courses* inapplicable, each occurrence of the object denoted by the variable p has to be removed from variables, sets and functions that range over the type *Empl* or a subtype.

This semantics guarantees that static type checking is sufficient despite operations that change types of objects dynamically[3]. More intuitively, this implements the point of view that type information is part of the constraints that every valid database state has to fulfill. Once such constraints fail to hold, the state is changed by removing the objects from variable values.

In case of destruction, an object o is removed from all variables, sets, classes and function values in the database and application program. However, we do not need an additional operation, since this functionality is already subsumed by removing the relationship between the object o and the most general type *Object* by the operation **lose** $[Object](o)$.

4.4 BCOOL's conflict relation

In this section, we discuss the properties of each generic update operation. This includes the definition of conflicts between the operations depending on restrictions to which object the operations are applied.

Conflicts between operations can be derived by analyzing the semantics of the update operations defined in Section 4.3. Because this analysis takes no state–dependent information into account, such as the values of types, variables, and functions, the resulting conflict relation can be used at compile-time. This analysis is carried out for each update operation in the following. Notice however, that our notion of "conflict" does not necessarily mean real conflicts that arise at run–time, but only potential conflicts that *might* occur.

4.4.1 Global assignment

Certainly, global assignments (such as $v := e$) to the same variable v in different threads conflict with each other, because the assigned expression e might depend on the thread object. Thus, if we allow global assignments within **apply_to_all** operations, there will always be conflicts that restrict the commutativity of threads. Instead, in order to avoid the conflicts, we forbid the application of global assignments within the **apply_to_all** operation. This seams reasonable, since there is no need to change the value of a variable in a set-oriented way; this could be executed before the update.

4.4.2 Partial assignments

Also in this case we have to avoid assignments of different values for the same function argument, which might occur due to object sharing (see Example 1 in Section 2). Again, we have a choice to cope with the conflicts or to restrict the operation. Since we want to allow partial assignments (e.g. in order to change the "internal state" of an object), we restrict their applicability to thread objects. This restriction ensures that no ambiguities arise between partial assignments of different threads at the expense of flexibility. For example, consider a unique function g and the update operation **set** $[f := e](g(o))$ that occurs in an

[3] That is, if objects are regarded as references, dangling references can not occur.

update sequence, where o denotes the elements of the set. Despite the fact that no object sharing can occur (because of the uniqueness), this update operation is prohibited by the above restriction.

There are conflicts between the operations **set** $[f := e]$ and **lose** $[T]$ of different threads in case they refer to the same object and if the function f is not inherited to, but defined on type T. The operation **lose** states that the function f must not be applied, whereas the **set** operation makes use of this function. This conflict could be eliminated, if the application of the **lose** operation is also restricted to the thread object. However, as explained below, we are not interested in this restriction. There are no conflicts between **set** and the operations **create** and **gain**, since these change only the active domain of types. However, a partial assignment conflicts with retrieving the function value in another thread, except if the function is applied to the thread object (see Example 2).

4.4.3 Create

The operation **create** in an **apply_to_all** statement yields a newly generated object and assigns it to a local variable for further reference. The scope of these local variables is restricted to one thread. That is, local variables are distinct in each thread even if they all use the same name. For example, the update operation **create** $[T](v)$ yields a new instance of type T, for each $thread_i$, that can be referenced in subsequent expressions by the local variable v, whose scope is $thread_i$. Since the generated objects are not known in other threads, the operation **create** has no conflicts with any update operation. Particularly, there is no conflict between two **create** operations in different threads. These operations need synchronization on the implementation level of the BCOOL operations according to the invention of object identities, but semantically they commute. That is, since object identity is not visible on the BCOOL level, we do not need to care about different database states that are isomorphic up to renaming of object identities (see the notion of O-isomorphisms in [2]). The operation **create** $[T](v)$ conflicts, however, with retrievals on the active domains of the type T and its supertypes, to which the generated object is added.

Notice, that two **create** operations semantically commute, although there are conflicts between the lower operations that implement them. This is a general observation. For example, type implementors might state the commutativity of two methods, even if there are conflicts between the BCOOL operations used in their implementations.

4.4.4 Gain

Similar to the **create** operation, **gain** only changes the active domain of types. Therefore, **gain** conflicts with retrieval operations on these active domains. Because **gain** is applied to already existing objects (in contrast to **create**), it might also conflict with the **lose** operation. The conflict between **gain** $[T_1]$ and **lose** $[T_2]$ arises, if T_1 is a subtype of T_2 ($T_1 \preceq T_2$) and both operations are applied to the same object in different threads. We could eliminate this conflict, which occurs in object sharing, if the **gain** and **lose** operations are restricted in the same way as the **set** operation; i.e., the operations are only applicable to the thread object. However, then we would lose the idempotency of either operation. This is the property that either operation can be applied to the same object repeatedly without conflicts, since either operation only adds (removes) this object into (from) the active domains, which again is an idempotent operation.[4] For example, firing the manager of employees e in $empset$ by

apply_to_all [**lose** $[Emp](mgr(e)](e : empset)$

is feasible, since the result does not depend on how often a manager that is "shared" by different employees in $empset$ is fired, if he/she is fired at least once.

Thus, we do not restrict the application of **gain** and **lose** in general, but cope with the conflict.

4.4.5 Lose

As already mentioned in Section 4.3, the operation **lose** $[T]$ conflicts with all variables, functions, and active domains that are related to type T or a subtype of T, in addition to the conflicts we discussed above. However, even though different **lose** operations might influence the values of the same variables and functions, they commute.

For example, assume that a variable v of type $\{T\}$ contains the objects o_1, o_2 and, o_3. The update **apply_to_all** [**lose** $[T](o)](o : \{o_1, o_2\})$ is transformed into two single update operations according to the two elements o_1 and o_2. Both manipulate v by excluding one object from the set, nevertheless they are commutative: the result will be $\{o_3\}$ in any possible sequence.

However, the deletion of objects in an **apply_to_all** operation, i.e., the operation **lose** $[Object]$, is restricted to the thread object, in order to guarantee the existence of other thread objects.

4.4.6 Summary of conflicts

Let us summarize the conflict relation of BCOOL's generic update operations in a table. An entry '+' means that two operations do not conflict, and the variable *self* denotes the thread object. Since the conflict relation is symmetric, the values of the conflict relation for empty cells are specified by changing row and column. The table also describes the conflicts between update and query operations. We state the

[4]Notice that deletion is restricted in Sect. 4.4.5.

	set $[f := ..]('self')$	**create** $[T_1]$	**gain** $[T_1]$	**lose** $[T_1]$
v	$+$	$+$	$+$	v defined on T_1 or its subtypes
$f(v)$[5]	$v \neq self$	$+$	$+$	f defined on T_1 or its subtypes
$ad(T_2)$	$+$	$T_1 \preceq T_2$	$T_1 \preceq T_2$	$T_2 \preceq T_1$
select $[P]$	$+$ [6]	$+$	$+$	$+$
project $:: T_2$[7]	$+$	$+$	$+$	$+$
extend $:: T_2$[7]	$+$	$+$	$T_1 \preceq T_2$	$T_2 \preceq T_1$
$\cup$	$+$	$+$	$+$	$+$
$\cap :: T_2$[7]	$+$	$+$	$T_1 \preceq T_2$	$T_2 \preceq T_1$
pick	$+$	$+$	$+$	$+$
set $[f := ..](o_j)$	$+$	$+$	$+$	f is defined on T_1
create		$+$	$+$	$+$
gain $[T_2]$			$+$	$T_2 \preceq T_1$
lose $[T_2]$				$T_1 = 'Object'$

Table 1: Summary of conflicts

query operations independent of their arguments, because the conflict between update operations and the arguments of query expressions are recursively contained in the table. Notice that query expressions do not conflict with each other.

5 Summary

We proposed an iterator that allows to avoid updates with a non–deterministic semantics. The iterator can be used to make single update operations as well as update sequences applicable to sets. Because the mechanism is independent of a particular choice of single-object update operations or data models, it can be used, for example, in the relational and in object-oriented models. Thus, our approach can be used to check the deterministic semantics of triggers that are applied after (set-oriented) update operations, and of embedded SQL programs. Furthermore, the iterator can be used to apply object-oriented methods, which consist of update sequences, in a set-oriented way.

According to the set-orientation, our approach is related to [12]. There, the idea to get semantics for parallel updates is to use partial interpretations of the new database for each single update. A parallel update, which is the conjunction of single updates, is well defined, if the partial interpretations are consistent. Thus, Chen's partial interpretations correspond to the threads in our approach, and the consistency of partial interpretations to the commutativity of threads. However, in contrast to [4, 12] we discuss updates whose effect is not restricted to the elements of the set the update is applied to. In addition, we generalize the approach to update sequences in order to take triggers in the relational model or methods of object-oriented ones into consideration.

In comparison to [5], our notion of deterministic semantics of set-oriented update sequences roughly corresponds to their confluence and observable determinism. However, this is a rough comparison not only, because their notion applies to all rules defined in the database whereas our notion refers to single set-oriented update sequences, but also since the underlying execution models of both approaches are different: their granularity for defining rules (that consist of a sequence of SQL-statements that is applied once for all changed tuples) are operations on data (in form of tables), whereas we regard functions (coded by methods) applicable to objects.

We discussed the similarities between our mechanism and semantic concurrency control. It turned out that the conflict relation of the update operations, which is needed for semantic concurrency control, can also be used for the **apply_to_all** iterator as the only

[5] v is a meta variable that denotes a variable: I.e., if the function f is applied to the variable *self* no conflicts arise.

[6] If f occurs in P, the conflicting operations are **set** and the function application $f(o_j)$

[7] The notation $:: T_2$ indicates that the result of the query operation is of type T_2.

model-specific information.

Finally, we presented the conflict relation for an object-oriented model, in which coping with sharing problems becomes more involved, because object sharing is an essential feature [6]. In doing so, we discussed the trade-off between the functionality of update operations and the number of conflicts.

An additional advantage of our definition of set-oriented update semantics is that it gives a direct hint for an efficient, namely a parallel, implementation.

References

[1] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical Report 846, INRIA, Paris, May 1988.

[2] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD*, Portland, June 1989.

[3] S. Abiteboul, P.C. Kanellakis, and E. Waller. Method schemas. In *Proc. ACM PODS*, Nashville, TN, April 1990.

[4] S. Abiteboul and V. Vianu. A transaction language complete for database update and specification. In *Proc. ACM PODS*, San Diego, CA, March 1987.

[5] A. Aiken, J. Widom, and J.M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proc. ACM SIGMOD*, San Diego, CA., June 1992.

[6] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In Kim et al. [19], pages 40–57.

[7] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. VLDB*, Brighton, September 1987.

[8] J. Banerjee, H.-T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou, and H.-J. Kim. Data model issues for object-oriented applications. *ACM TOIS*, 5(1):3–26, January 1987.

[9] D. Beech. A foundation for evolution from relational to object databases. In *Advances in Database Technology — EDBT'88*. LNCS 303, Springer Verlag, Heidelberg, March 1988.

[10] C. Beeri. Formal models for object-oriented databases. In Kim et al. [19]. Revised version in "Data & Knowledge Engineering", Vol. 5, North-Holland.

[11] C. Beeri, P.A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(1), 1989.

[12] W. Chen. Database updates: Constructors and quantifiers. In *Proc. DOOD*, Munich, Germany, December 1991. LNCS 566, Springer Verlag, Heidelberg.

[13] K. Denninghoff and V. Vianu. The power of methods with parallel semantics. In *Proc. VLDB*, Barcelona, Spain, 1991.

[14] O. Deux et al. The story of O_2. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):91–108, March 1990. Special Issue on Prototype Systems.

[15] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS*, 8(2):186–213, June 1983.

[16] G. Ghelli. A class abstraction for a hierarchical type system. In *Proc. ICDT*, Paris, December 1990. LNCS 470, Springer Verlag, Heidelberg.

[17] C. Hasse and G. Weikum. A performance evaluation of multi-level transaction management. In *Proc. VLDB*, Barcelona, Spain, 1991.

[18] ISO. Information technology - Database language SQL2. ISO/IEC 9075:1992.

[19] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proc. DOOD*, Kyoto, December 1989. North-Holland.

[20] C. Laasch and M.H. Scholl. Generic update operations keeping object-oriented databases consistent. In *Proc. GI Workshop Information Systems and AI*, Ulm, Germany, February 1992. IFB 303, Springer Verlag, Heidelberg.

[21] Ontologic Inc., Burlington (MA). *ONTOS - Object Database Documentation, Release 2.0*, 1990.

[22] Relational Technology Inc., Alameda, Ca. *INGRES/-EMBEDDED SQL User's Guide and Reference Manual, Release 6.3*, January 1990.

[23] K.-D. Schewe, J. W. Schmidt, and I. Wetzel. Identification, genericity and consistency in object-oriented databases. In *Proc. ICDT*, Berlin, Germany, October 1992. LNCS 646, Springer Verlag, Heidelberg.

[24] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proc. DOOD*, Munich, December 1991. LNCS 566, Springer, Heidelberg.

[25] M.H. Scholl and H.-J. Schek. A relational object model. In *Proc. ICDT*, Paris, December 1990. LNCS 470, Springer Verlag, Heidelberg.

[26] M.H. Scholl and H.-J. Schek. A synthesis of complex objects and object-orientation. In *Proc. IFIP TC2 Conf. on Object Oriented Databases (DS-4)*, Windermere, UK, July 1990. North-Holland.

[27] J. E. Stoy. *The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge (Mass.), 1977.

[28] W.E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. on Comp.* , 37(12), 1988.

[29] G. Weikum. Prinicples and realization strategies of multi-level transaction management. *ACM TODS*, 16(1):132–180, March 1991.

[30] G. Weikum and C. Hasse. Multi-level transaction management for complex objects: Implementation, performance, parallelism. TR 162, ETH Zürich, Dept. of Computer Science, July 1991.

[31] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris architecture and implementation. *IEEE T-KDE*, 2(1):63–75, March 1990.