Reasoning about Set-Oriented Methods in Object Databases

Extended Abstract

Giovanna Guerrini *Isabella Merlo*

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso, 35 - I16146 Genova, Italy
{guerrini,merloisa}@disi.unige.it

Abstract. Most existing object-oriented database systems exploit a general purpose imperative object-oriented programming language as method definition language. The use of such languages makes it difficult to analyze method properties and to optimize them. We believe that the use of an ad-hoc, set-oriented language can offer some advantages, at least at a specification level. In particular, such a language can offer an appropriate framework to reason about method properties. In this paper we propose a set-oriented rule-based method definition language for object-oriented databases which supports set-oriented update sequences with a deterministic semantics and we characterize some relevant properties of methods, such as behavioral refinement in subclasses and conflicts among method specifications in sibling classes, in the framework of the proposed language.

1 Introduction

One relevant innovation introduced in the database field by the object-oriented paradigm is the overcoming of the distinction between data and operations manipulating them. In relational database systems, two languages are provided: a data definition language (DDL) and a data manipulation language (DML), including as a sublanguage the query language (QL). The DML provides a small number of update primitives, to express atomic data manipulations (such as insertion or deletion of a tuple, or modification of a field of a tuple). The DML is thus simple, but, however, not computationally complete. As a consequence, complex data manipulations must be expressed by embedding DML primitives in a host language, which is usually a *general purpose*, computationally complete language. A well known example is the use of SQL embedded in host languages, such as C or COBOL. In an object-oriented database, objects encapsulate the state and the operations manipulating the state. Thus, information related to data manipulation are not dispersed in application programs, rather they are recorded in the database. In object-oriented database systems, we can then distinguish three different languages: the data definition language, the query language, and the method definition language. Most object-oriented database systems use existing object-oriented programming languages or extensions of existing object-oriented programming languages, as languages for defining methods. We argue, by contrast, that using a simpler, set-oriented language for defining methods rather than an object-oriented programming language can offer some advantages. Indeed, object-oriented programming languages have not been designed to manipulate large amounts of data. Data manipulation code written in such languages is often impossible to optimize and not easily readable. Moreover, predefined primitives, well-suited for data manipulation, implemented at system-level, are completely missing. Another missing feature is set-orientedness. In the database context, the common case is to execute a given update operation on a set of objects rather than on a single object. The method definition languages of some existing systems [13, 16] provide an *iterate* operator to solve this problem.

An important advantage of a special purpose set-oriented language is that such language is well-suited for reasoning about methods. That is, useful semantic properties can be easily characterized for methods specified in that language. An important example of those properties is the behavioral refinement of methods in subclasses. Moreover, the use of such a language, specifically designed to support a set-oriented access to large amounts of data, would substantially increase efficiency, because access and query optimization techniques could be exploited. Such a language could also be used as a method specification language, from which methods containing SQL-like statements would be automatically derived.

In this paper we propose a set-oriented rule-based method definition language. An important issue in defining the semantics of a language supporting set-oriented updates is related to non-determinism [3, 14]. Because of non-determinism, set-oriented updates may lead to different correct database states. Therefore, an important goal of our work is to define a language with a deterministic semantics. Then, we show how, relying on such a semantics, relevant properties of methods can be formally stated and investigated. In particular, we consider behavioral refinement of methods in subclasses [15], that is, how to ensure that, when overriding a method, the method in the subclass refines, from a behavioral viewpoint, the method it overrides. Another relevant problem we address is that of conflicting method implementations in sibling classes. This problem arises in models supporting multiple class membership [6]. For those properties, some sufficient static conditions have been devised for our language.

The contributions of the paper can be summarized as follows: (i) definition of a set-oriented, rule-based method definition language for object-oriented databases with a deterministic semantics; (ii) use of the semantics of the proposed language as a formal basis for reasoning about some relevant properties of object-oriented methods, namely, refinement of method implementations and conflicting method implementations.

The paper is organized as follows. Section 2 presents an overview of our language and of its semantics. Section 3 discusses semantic properties of methods.

Finally, Section 4 concludes the paper.

2 A set-oriented method definition language and its semantics

A method implementation in our language is an expression of the form:

$$\begin{aligned} m(P_1, \ldots, P_n) : \mathit{condition}_1 &\rightarrow u_1^1; \ldots; u_n^1 \\ &\vdots \\ \mathit{condition}_m &\rightarrow u_1^m; \ldots; u_k^m \end{aligned}$$

where $P_1, \ldots, P_n$ are variables representing the method parameters. For the sake of simplicity, we assume that the former j parameters $P_1, \ldots, P_j$ $(0 \leq j)$ in a method implementation are the formal input parameters, and the latter $P_{j+1}, \ldots, P_n$ $(j \leq n)$ are the formal output parameters[1]. Each $\mathit{condition}_i$, $i \in [1, m]$, is a declarative formula expressing control upon method execution, and each u_j^i is an update operation. An expression $\mathit{condition}_i \rightarrow u_1^i; \ldots; u_n^i$ in the above method implementation is called *rule*. Moreover, $\mathit{condition}_i$ is called the *condition part* of the rule, whereas $u_1^i; \ldots; u_n^i$ is called the *action part* of the rule.

In our language conditions are conjunctions of atomic formulas and are interpreted as predicate calculus expressions over typed variables. Our language provides a classical set of terms, defined as follows. Constants (except *oids*) and variables are terms; moreover, structured terms can be built by applying set, list and record constructors. Path expressions (built making use of the dot notation) are terms, too. In addition we support a set of quite standard predefined operators that can be used to build terms. These predefined operators include arithmetic operators, set operators and list operators. Atomic formulas can be of three types, being t_1, t_2 terms and X a variable: *comparison formulas* (e.g. $t_1\ op\ t_2$, $op \in \{<, >, \geq, \leq, =, ==, ==_d\}$[2]); *membership formulas* (e.g. t_1 *in* t_2 or t_1 *in* c where c is a class name); *class formulas* (e.g. $c(X)$, where c is a class or type name). Complex formulas (or simply formulas) are obtained from atomic formulas and negated atomic formulas by means of conjunctions. Formally, if F is an atomic comparison or membership formula[3], then $\neg F$ is a (complex) formula; if F_1 and F_2 are formulas, then F_1, F_2 is a (complex) formula, where the symbol "," denotes the *and* logical connective. Each formula is required to contain exactly one class formula for each variable, specifying the type of the variable. All variables are assumed to be implicitly quantified. In addition, formulas are

[1] Note that a method can have no input parameters $(j = 0)$, or no output parameters $(j = n)$.

[2] Notice that our language provides three different types of equality: $=$ denotes equality by identity, $==$ denotes shallow value equality, while $==_d$ denotes deep value equality.

[3] Class and type formulas cannot be negated.

required to be range restricted, to avoid formulas that are satisfied by an infinite set of instances.

An update operation can be a `create`, `delete`, `modify`, `specialize` or `generalize` operation, or a method call. A method invocation (or method call) is an expression of the form:

$$O.m(t_1, \ldots, t_n)$$

where O is an object-denoting variable. The actual parameters, $t_1, \ldots, t_n$, are terms. In particular, the actual output parameters, $t_{j+1}, \ldots, t_n$, are variables.

As it is usual in database systems, a method is invoked with respect to a database state, denoted by S, and a set B of bindings (that is, assignments of values to variables). S and B represent the situation with respect to which the method is invoked. In particular, B represents the state of the variables of the program within which the method is invoked. This state, together with the method behavior, completely defines the semantics of the method execution[4].

As we have just stated a method call is implemented by several rules, where each rule has the form: *condition* $\rightarrow u_1; \ldots; u_n$. Hence first we would have to formalize the semantics of conditions. Method conditions can be seen as an extension of first order formulas. Informally, we can say that the result of the evaluation of a condition C is a set of substitutions, each grounding for C and such that the instantiation of C with respect to each of them, evaluates to true in state S according to first order logic. Based on first order logic, the semantics of conditions can be easily formalized. The formal definition of condition semantics can be found in [7].

Given the semantics of conditions, we will first formalize the semantics of atomic updates, then the semantics of update concatenation and, finally the semantics of each rule which implements the method body; to complete the definition of the semantics of methods we will then specify how such rules interact among each other.

In what follows we first describe the semantics of set-oriented update sequences in our language, and we then discuss the semantics of method execution.

2.1 Set-oriented update sequences

Consider an update sequence of the form u_1;...;u_n to be performed on a set of variable bindings B, a set of objects, in a particular database state, where each u_i is an update operation (either atomic or a method call). The first approach that could be taken to execute these updates is the deductive database approach [8] where an update sequence is considered as a set of updates and each single operation is executed according to the set-oriented semantics, that is, on the set

4 We assume that concurrency control is used to synchronize accesses on the underlying database. Therefore, there is no interference between concurrent executions.

of bindings B. Because updates u_1;...;u_n are considered as a set, the execution order must be irrelevant. This approach thus needs a notion of consistent updates. Consistency is important because it prevents a set of updates containing both an insertion and a deletion of the same object from being executed. One disadvantage of this approach is that not all syntactically correct update sequences are semantically correct. Moreover the sharp distinction between the phase in which bindings are collected and the update phase is not convenient for our language in which updates can be method calls.

Another possible approach is that of object-oriented update sequences [14]. In this approach u_1;...;u_n is considered as a sequence, in the real meaning of the word (one update after the other). Such sequence must be executed, in the established order, on each single object in B. In this case the order in which the objects in B are chosen to perform the updates must be irrelevant, otherwise the semantics is non-deterministic. To make the semantics deterministic, only update sequences which commute on objects in B are accepted as valid. Like the deductive approach, not all syntactically correct update sequences are semantically correct. This semantics eliminate the sequences which cause problems, by considering them invalid, without solving the problem of non-determinism. Moreover, several problems concerning aggregate functions and data sharing arise [7].

Our approach combines the principles of the previous approaches. The updates u_1;...;u_n are seen as a sequence; each update is performed on the set of bindings B. Therefore, each u_i is performed as in the set-oriented approach. Suppose to start from a set of bindings B and a database state S and to have to perform the updates u_1;...;u_n. After the execution of the update u_1 the pair $\langle B, S\rangle$ will be transformed in a pair $\langle B', S'\rangle$ with respect to which the update u_2 will be performed and so on. Our approach seems preferable to the other ones because it better models the intuitive semantics. Moreover note that, in contrast with the deductive approach and like the object-oriented approach, the update execution order is established by the sequence itself. So we avoid consistency problems. We also avoid problems related to aggregate functions. Indeed, if an aggregate function appears in one update of the sequence, we compute the aggregate function result in the database state immediately before the update and then we perform the update with such a result on all objects in B.

The following example shows how the three approaches differ and how our approach solves problems arising with the others.

Example 1. Consider a class `employee` containing two attributes: `salary` (with domain `integer`) and `manager` (with domain `employee`), representing the salary and the manager of an employee, respectively. Consider a method `raise_salary` of class `employee` that, when invoked on an employee sets its salary to the salary of his manager and increases the salary of his manager of an amount specified as input. Thus, the signature of method `raise_salary` is `employee` $\times$ `integer` $\rightarrow$, whereas its implementation is specified by the following rule:

```
raise_salary(Amount):M = Self.manager →
```

```
modify(employee.salary, Self, M.salary);
modify(employee.salary, M, M.salary + Amount)
```

Consider the invocation of method `raise_salary` with respect to the following set of bindings $B = \{\theta_1, \theta_2\}$ with $\theta_1 = \{\texttt{Self}/o_1, \texttt{Amount}/100\}$ and $\theta_2 = \{\texttt{Self}/o_2, \texttt{Amount}/100\}$, and a database state S such that the value of attribute `manager` of object o_1 is o_2 and the value of attribute `manager` of object o_2 is o_2 (that is, o_2 is manager of himself).

According to the deductive approach such a method invocation fails. Indeed the first `modify` operation sets the salary of o_2 to his salary in S, while the second `modify` operation sets the salary of o_2 to his salary in S increased by 100. These two modifications are collected in a set and since they are not consistent (different values are assigned to the same attribute) the method invocation fails.

According to the object-oriented update sequence approach, the semantics is non-deterministic, indeed, different database states are obtained depending on the order in which the two substitutions are considered. If substitution θ_1 is considered first, the salary of o_1 is set to the salary of o_2 in S, whereas the salary of o_2 is set to his salary in S increased by 200. By contrast, if substitution θ_2 is considered first, the salary of o_1 is set to the salary of o_2 in S increased by 100, whereas the salary of o_2 is set to his salary in S increased by 200. Therefore, o_1 could receive different salaries depending on the order in which the substitutions are considered. The semantics is thus non-deterministic since it depends on that order.

With our approach, by contrast, no problems arise and the salary of o_1 is set to the salary of o_2 in S, whereas the salary of o_2 is set to his salary in S increased by 100. $\diamond$

Note that our approach does not solve all the problems related to non-determinism. Consider indeed the case of an update that sets the salary of managers to the salary of their employees. Of course a manager can have more than one subordinate employee, thus the final manager salary depends on the execution order. The problem with this kind of updates is that they are intrinsically ambiguous. Therefore, they are not assigned any semantics in our model, rather an error message is issued as result of the execution of the update sequence.

Before formalizing atomic update semantics, we need to introduce the preliminary definitions of *database state* and *set of bindings*. In the following we consider: a set $\mathcal{V}$ of values; a set OID of all possible *oids*; a set C of classes; a set Var of variables.

We consider a simple and rather standard [1] definition of database state.

Definition 1. A *database state* (database for short) is a pair $S = (\pi, \nu)$ where:

- $\pi : C \rightarrow 2^{OID}$ is the function which associates with each class the set of *oids* of its proper instances[5];

[5] We remark that $\pi(c)$ denotes the set of *oids* of those objects for which c is the most

– for each $i \in OID$, object identifier, $\nu(i)$ returns the state of the object, that is, the value of its attributes; if the attributes names are $a_1, \ldots, a_n$ and $v_1, \ldots, v_n$ the corresponding values, $\nu(i) = [a_1 : v_1, \ldots, a_n : v_n]$[6]. □

For what concerns sets of bindings, we model them as sets of ground substitutions.

Definition 2. A *substitution* ϑ is a partial function from Var to $\mathcal{V}$; $\vartheta : Var \to \mathcal{V}$. A set of bindings B is a set of substitutions $\{\vartheta_1, \ldots, \vartheta_m\}$. □

Given a set *Update* of possible action parts of rules[7], a set *Bind* of possible sets of bindings, a set *State* of possible database states, the update semantic function is as follows:

$$\mathcal{U} : Update \to ((Bind \times State) \to (Bind \times State)).$$

Function $\mathcal{U}$ performs the semantic evaluation of a sequence of updates, that is, an action part, $u_1;\ldots;u_n$ of a rule. The semantics of update concatenation is quite intuitive: at each step the first update of the sequence is evaluated, such evaluation gives as result a new set of bindings and a new state with respect to which the remainder of the sequence is evaluated. Formally:

$$\mathcal{U}\,[\![\, u_1; \ldots; u_n \,]\!]\, BS = \mathcal{U}\,[\![\, u_2; \ldots; u_n \,]\!]\, (\mathcal{U}\,[\![\, u_1 \,]\!]\, BS)$$

For what concerns the evaluation of a single update, function $\mathcal{U}$ is specified for each different type of atomic update, that is, `create`, `delete`, and so on, and for method calls. Here we only specify, as an example, the semantics of the `delete` operation. The full definition of function $\mathcal{U}$ can be found in [7]. Given a set of binding B and a variable $\bar{O}$, let $\bar{O}_B$ be the interpretation of $\bar{O}$ in B, that is, the set of *oids* to which $\bar{O}$ is bounded in B, then:

$$\mathcal{U}\,[\![\, \mathtt{delete}(\bar{c}, \bar{O}) \,]\!]\, BS = \langle B, S' \rangle$$

where $S' = \langle \pi', \nu' \rangle$, and:

$$\pi'(c) = \begin{cases} \pi(c) \setminus \bar{O}_B & \text{if } c = \bar{c} \\ \pi(c) & \text{if } c \neq \bar{c} \end{cases} \qquad \nu'(i) = \begin{cases} \nu(i) & \text{if } i \notin \bar{O}_B \\ \perp & \text{if } i \in \bar{O}_B \end{cases}$$

---

specific class in the inheritance hierarchy.

6 To denote the values of single attributes we use the following notation: $\nu(i).a_j = v_j$, $j \in [1, n]$.

7 *Update* corresponds to the set of well-formed update sequences of the language.

2.2 Method execution semantics

In specifying the semantics of method execution, four different phases can be distinguished:

1. Selection of method implementation.
2. Generation of the set of bindings in which the rule conditions are evaluated.
3. Evaluation of the method implementation.
4. Generation of the set of bindings to return as result.

The first problem arising in the definition of the semantics of method calls is to establish which method has to be invoked for each *oid* bound to the variable, on which the method is invoked. Our language supports *late binding*, that is, at run-time, for each object the method related to its most specific class is invoked. Late binding, however, complicates the semantics of set-oriented method execution. According to the set-oriented approach, given the invocation $O.m(t_1, \ldots, t_n)$ to be executed in the state S with respect to the set of bindings B, the body of method m should be executed in the state S on each object bound to O in B and on each parameter evaluation corresponding to a substitution in B. Note that each of these body executions transforms the database from one state to another. Therefore, when different method implementations are executed for different sets of objects, the database transformation into a new state is not atomic. Each implementation may perform completely different updates on the database. The basic idea to solve that problem is to partition the set of objects to which variable O is instantiated by substitutions in B into disjoint subsets of *oids*, say $O_B^1, \ldots, O_B^t$, according to the most specific class of each object. Then, the corresponding method implementation is executed on each set of objects and the final states $S_f^1, \ldots, S_f^t$ are "merged" in a single final state S_f. If those states cannot be deterministically merged (e.g. since they have updated the same attribute of the same object in different ways) an error is raised.

For what concerns the other phases, the semantics of the execution of method m is informally defined as follows. When the method is invoked, first of all, the actual parameters $t_1, \ldots, t_j$ are evaluated and a new set of bindings B_P is generated. The generation of this set of bindings establishes the correspondence between the formal input parameters $P_1, \ldots, P_j$ and the values resulting from the evaluation of $t_1, \ldots, t_j$. Note that no bindings for the formal output parameters $P_{j+1}, \ldots, P_n$ are included in B_P. Then, each condition is evaluated against state S starting from the set of bindings B_P. We get as result the sets of bindings $B_P^1, \ldots, B_P^m$ which can be considered as extensions of the initial set B_P.

Starting from these sets of bindings and from the state S, the update sequence of the body of the i-th rule is evaluated in state S with respect to the set of bindings $B_P^i \setminus (B_P^1 \cup \ldots \cup B_P^{i-1})$, rather then with respect to B_P^i. One of the main issues in giving the semantics of a method implemented by several rules, indeed, is that the sets of bindings returned by condition evaluations are not, in general, disjoint. Intuitively we want that a method performs different operations corresponding to sets of different objects. We could impose the sets of bindings resulting from the evaluation of the conditions of different rules be disjoint, that is, $B_P^1 \cap \ldots \cap B_P^m = \emptyset$ and return an error whenever such restriction is not fulfilled. In our opinion such restriction is too strict. To capture the desired semantics we have chosen to establish a sort of implicit order among the rules in the implementation of a method. This is the reason why the evaluation of the body of the i-th rule takes place in $B_P^i \setminus (B_P^1 \cup \ldots \cup B_P^{i-1})$, that is, all the substitutions in B_P^i from which all the substitutions already computed in the previous rules are eliminated, if they are present.

Starting from the first rule, updates $u_1^1; \ldots; u_n^1$ are performed on state S with the set of bindings B_P^1, after the execution of u_n^1 the database state will be S_n^1. Likewise the second rule body is executed in state S_n^1, with set of bindings $B_P^2 \setminus B_P^1$ and so on. After all rules are executed, the database state is in the final state, S_f which is returned as result of the method invocation. The final set of bindings B_f resulting from method invocation is obtained from the sets of bindings resulting from the evaluation of the different rules. The substitutions in those sets, however, refer to the local variables of the method. Therefore, they are joined to the set B of bindings on which the method has been invoked, and a correspondence is established between the formal and the actual output parameters.

To conclude, we notice that, if an update operation u_i is a method invocation, its evaluation makes the database go through a sequence of intermediate states and gives as result a final set of bindings and a final state. Moreover, rules in a method implementation are executed sequentially, according to the order in which they appear in the implementation and according to the described semantics; rule r_{i+1} is then executed after rule r_i has completed its execution. We refer the interested reader to [7] for a complete and formal definition of method semantics.

3 Semantic properties

In this section we show how some relevant semantic properties of methods can be reasoned about in our language. We consider behavioral refinement of methods in subclasses and conflicts among method implementations in sibling classes. Both these properties can be formally characterized in terms of the semantics of our language. Note that these properties depend on the specific database state, and are thus undecidable in general. We have however devised some sufficient static conditions ensuring the verification of those properties.

3.1 Behavioral refinement

In object-oriented languages, polymorphic code can be written. However, this makes reasoning about methods difficult. There might be several method implementations that could be executed upon a method invocation. In most object-oriented programming languages, moreover, method overriding is always uncontrolled. The only restrictions imposed on method redefinitions in subclasses concern method signatures, but no restriction is imposed on method implementations to ensure that semantics is preserved. Intuitively, if a class c' is a subclass of a class c, every object instance of c' must behave *like* some object of class c. This notion of semantic method refinement is known as *behavioral subtyping* [15]. Consider as an example an *init* method initializing the attributes of an object. The implementation of such method in each class initializes to the proper values the attributes of that class. An *init* implementation in class c' that only extends the *init* implementation of class c with a new piece of code to initialize the additional attributes of c' is an example of behavior refinement. To obtain a correct behavioral subtyping, *super* calls or the *inner* mechanism could be exploited.

A method m in a class c' (denoted by $m^{c'}$) is a behavioral refinement of method m in a class c (denoted by m^c) if the portion of state manipulated by $m^{c'}$ includes the portion of state manipulated by m^c and if the portion of state modified by both is modified in the same way[8]. To formally define this notion, we must first model the changes made by a method execution. Given a method m, let $\mu(m)$ be the set of classes manipulated by m (either through atomic updates or methods calls)[9]. Given a method m and a class c, let $\delta_m(c)$ be the set of objects deleted from class c and $\iota_m(c)$ the set of objects inserted in class c as a consequence of the execution of method m; moreover, given an *oid* i and an attribute name a, let $\nu_m(i).a$ be defined if and only if the execution of method m has modified the value of attribute a of the object identified by i, and, if defined, let it contain the new value of the attribute.

Definition 3. Given a method implementation m, such that $\mu(m)$ is the set of classes manipulated by m, a database state $S = \langle \pi, \nu \rangle$ and a set of bindings B such that $\mathcal{U}[\![m]\!] BS = \langle B', S' \rangle$ (where $S' = \langle \pi', \nu' \rangle$), consider the following functions:

- $\forall\, c \in \mu(m) : \delta_m(c) = \pi(c) \setminus \pi'(c)$;
- $\forall\, c \in \mu(m) : \iota_m(c) = \pi'(c) \setminus \pi(c)$;
- $\forall\, c \in \mu(m)$, $\forall\, i \in \pi'(c) : \nu_m(i).a = v$ if and only if the execution of method m has modified the value of the attribute a of object i setting it equal to v.

The triple $\langle \delta_m, \iota_m, \nu_m \rangle$ models the portion of state manipulated by the method invocation. □

Definition 4. Given a class c' subclass of a class c, method $m^{c'}$ is a *behavioral refinement* of method m^c if $\mu(m^c) \subseteq \mu(m^{c'})$ and if, for each database state, the execution of m^c and $m^{c'}$ restricted to the objects in c' satisfies the following conditions:

- $\forall \bar{c} \in \mu(m^c)$: $\delta_{m^c}(\bar{c}) \subseteq \delta_{m^{c'}}(\bar{c})$ and $\iota_{m^c}(\bar{c}) \subseteq \iota_{m^{c'}}(\bar{c})$;

[8] Note that a method invoked on an object (set of objects) instance(s) of a class may manipulate objects of other classes.

[9] They can be deduced syntactically.

– $\forall \bar{c} \in \mu(m^c), \forall a$ attribute of $\bar{c}$, $\forall$ oid i instance of $\bar{c}$: if $\nu_{m^c}(i).a$ is defined, then $\nu_{m^{c'}}(i).a$ is defined and $\nu_{m^c}(i).a = \nu_{m^{c'}}(i).a$. □

Unfortunately, given two method definitions, behavioral refinement is undecidable [7]. This is a consequence of the fact that behavior refinement depends on the specific database state.

We have however devised some sufficient static conditions ensuring that a method $m^{c'}$ is a behavioral refinement of a method m^c. These conditions can be checked at method definition time, so that the overriding of a method in a subclass can be disallowed if the overriding method is not a behavioral refinement of the overridden one. These conditions (referred to as *refinement conditions*) requires that for each rule r in m^c a rule r' in $m^{c'}$ must exist refining r[10]. The implementation for $m^{c'}$ may also contain additional rules. A rule r' refines a rule r if the condition of r' subsumes the condition of r, restricted to the class of r', and the action of r' refines the action of r. Containment among conditions can be tested by exploiting algorithms for deciding subsumption among object-oriented queries [9, 12][11]. Action refinement basically means that the action of the refined rule must do at least what the action of the original rule does. This means that, for each update in the sequence constituting the action of the original rule there must be a corresponding update in the refined action. However, since rule actions are update sequences, the corresponding update could be discarded by some complementary update executed after it in the sequence. Consider as an example the case of a rule r creating an object in its action, overridden by a rule r' whose action firstly creates a corresponding object and then deletes it. In this case the overriding rule is not a refinement of the overridden one, since in the state resulting from the evaluation of rule r objects have actually been created, while such objects do not appear in the state resulting from the evaluation of rule r', since they have been deleted.

We consider, therefore, the net effect of the updates in the sequence and we state that for each update in the original rule there must be a corresponding update in the net effect of the refined rule. Note that the notion of net effect employed here is purely syntactical and relies only on complementary database operations. Net effect computation consists of composing the effects of those actions whose effect was compensated by a subsequent action on the same object. Classical compensations are performed as follows: a sequence of `create` and `delete` primitives on the same object, possibly with an arbitrary number of intermediate `modify` primitives on that object, has a null net effect; a sequence of `create` and several `modify` primitives on the same object has the net effect of

[10] Actually, a rule could also be refined by two or more rules, the disjunction of whose condition must subsume the condition of the original rule. However, we do not consider this case here due to space limitations.

[11] Actually, containment must be tested modulo variable renaming, since the methods could exploit different parameter names and different local variables. Moreover, the condition of the refined rule may also share some more variables with its actions. These issues have been considered in [7].

a single create operation; a sequence of several `modify` and a `delete` primitive on the same object has the net effect of a single delete operation on that object; a sequence of several `modify` primitives on the same object has the net effect of a single modify operation on the old object which modifies it in the newest. In addition to those classical compensations, we consider also compensations involving object migrations along the hierarchy.

Given a sequence of updates U, let $Net(U)$ denote the net effect of the sequence. The net effect of the sequence is computed at a syntactic level, by considering compensating actions on the same object-denoting term, contained in the sequence. Moreover, we consider a notion of refinement on update operations, denoted by $\leq_b$, such that if class c' is a subclass of class c, a create operation on c' is a refinement of a create operation on c, a delete operation from c' is a refinement of a delete operation on c, a mofify operation on an attribute $c'.a$ is a refinement of a modify operation on $c.a$, and so on.

Definition 5. A sequence of updates U' is a *refinement* of a sequence of updates U if the following conditions hold:

- $Net(U) = u_1; \ldots; u_n$, $Net(U') = u'_1; \ldots; u'_m$ and $m \geq n$;
- for each u_i, $i \in [1, n]$, in $Net(U)$, u'_j, $j \in [1, m]$, exists in $Net(U')$ such that $u'_j \leq_b u_i$, that is, u'_j is a refinement of u_i; let function $\xi : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$, such that $\xi(i) = j$, model this correspondence;
- if u_i precedes u_k in $Net(U)$, $u'_{\xi(i)}$ precedes $u'_{\pi(k)}$ in $Net(U')$. □

Definition 6. Given the definition of method $m^{c'}$, consisting of the rules $r'_1, \ldots, r'_n$, and the definition of method m^c, consisting of the rules $r_1, \ldots, r_k$, m^c and $m^{c'}$ meet the *refinement conditions* if for each i, $i \in [1, k]$, j exists, $j \in [1, n]$, such that, $r'_j = c'_j \rightarrow U'_j$, $r_i = c_i \rightarrow U_i$ and the following conditions are verified:

- c'_j subsumes c_i;
- U'_j is a refinement of U_i according to Definition 5. □

The following result holds.

Proposition 1 *Given two methods m^c and $m^{c'}$:*

- *we can decide whether m^c and $m^{c'}$ meet the refinement conditions;*
- *if m^c and $m^{c'}$ meets the refinement conditions, then $m^{c'}$ is a behavioral refinement of m^c.* □

Proof Sketch: The proof of the first item is based on the fact that both condition subsumption and action refinement are decidable.

The proof of the second item is more complex. $\mu(m^c) \subseteq \mu(m^{c'})$ follows from the fact that for each update rule R_i in m^c implementation a rule R'_j exists in $m^{c'}$ implementation, such that R'_j refines R_i. Since this means that for any upate

u^* in the net effect of R'_j action a corresponding update u^{**} in the net effect of R_i action exists, $\mu(m^c) \subseteq \mu(m^{c'})$.

Let us consider now the first condition of Definition 4, that is $\forall \bar{c} \in \mu(m^c)$: $\delta_{m^c}(\bar{c}) \subseteq \delta_{m^{c'}}(\bar{c})$. Again, this follows from the fact that for each update rule R_i in m^c implementation a rule R'_j exists in $m^{c'}$ implementation, such that R'_j refines R_i. This means that the set of objects selected by R'_j condition contains the set of objects selected by R_i condition (restricted to instances of c'), and for each delete operation in the net effect of R'_j action, a corresponding deletion is in the net effect of R_i action.

The same arguments can be applied for the other conditions of Definition 4, namely those concerned with insertions and modifications.

3.2 Conflicting method definitions

Some object-oriented data models [2, 6, 11] allow an object to be a direct member of multiple classes. This possibility has several advantages from a modeling viewpoint. However, it poses a number of problems with respect to method dispatching. Each class in the inheritance hierarchy can specify a different implementation for a certain method. For each method invocation, an implementation must be chosen among the most specific ones for the receiver object. In a model supporting multiple class direct membership, however, the choice of a method implementation that "most closely matches" an invocation is not obvious. In [6] we have addressed the problem of dispatching in object systems supporting multiple class direct membership. The approach is based on selecting the method implementation from *one* of the most specific classes of the object. This approach is, however, not always adequate. When different implementations for a method are defined in the most specific classes of an object, there are cases in which the most intuitive approach is executing all of them. Consider the *init* method above, initializing the attributes of an object. Suppose that different classes provide different implementations for the *init* method. Remember that the implementation of the *init* method in each class initializes to the proper values the attributes of that class. If, when the *init* method is invoked on an object, all the *init* implementations of all the classes of which the object is a direct member, are executed, the desired result is obtained, that is, all the attributes of the object are properly initialized. Problems due to different assignments to the same attribute may, however, arise. This approach causes the problem that some attributes can be assigned different values by different implementations of the method. To solve these issues we introduce the notion of non-conflicting method implementations. Different method implementations are non-conflicting if each method modifies a portion of data distinct from the one modified by the others and if data modified by two methods is modified consistently. This notion has been formally stated in terms of the semantics of our language. We do not present this definition here due to space limitations, it can be found in [7].

The notion of non-conflicting method definitions has been proved undecidable [7]. However we have devised some static conditions ensuring the absence of conflicts among two method definitions. A first static condition ensuring that two method implementations are non-conflicting is that the actions of the rules of the two method implementations do not contain updates on the same attributes or on the same classes. This condition can be checked through a simple syntactic analysis of the actions of the rules of which the method implementation consists[12]. Though this condition ensures that two method implementations are non-conflicting, it is too restrictive. Indeed, also methods modifying the same attributes of different objects are non-conflicting. We have also developed some sufficient conditions ensuring that two rule conditions are *exclusive*, that is, they will never be satisfied by the same objects. Then, only the actions of rules with non-exclusive conditions must be analyzed for the presence of conflicts. For these rules, first the net effect of those actions is computed and the updates corresponding to a common more general action (if any) are extracted. Then, if the remaining updates modify different portions of the schema, that is, different attributes or classes, the methods are non-conflicting. This means that we have devised a *syntactic* notion of non-conflicting method definitions, which has been proven decidable and has been proved being a sufficient condition for (semantic) absence of conflicts among method executions.

4 Conclusions

In this paper we have proposed a set-oriented method definition language for object databases. As we have discussed in the introduction, the use of an ad-hoc set-oriented language offers several advantages compared to the use of general purpose imperative programming languages. By contrast, one could argue that the use of a general purpose imperative programming language is the best way to overcome the impedance mismatch. Though our language is not computationally complete (because the language is restricted to non-recursive method definitions), it is expressive enough to model sequences of statements and if-then-else control constructs. Moreover, we believe that our framework could be easily extended to accommodate recursive method definitions. Note, moreover, that the object-oriented approach as opposed to other traditional approaches (such as embedding SQL into a programming language) improves the problem of impedance mismatch. Under the object-oriented approach, application programs in most cases can simply invoke methods, which encapsulate update commands. Methods are written through rules and need not to be interfaced with other languages.

Our set-oriented language has a deterministic semantics. In the paper, we have presented how it models set-oriented update sequences and contrasted our approach with other approaches in literature. Differently from other proposals

[12] This kind of analysis brings some similarities with the one proposed by [4] for detecting whether a method may potentially violate an integrity constraint.

dealing with sets (like, for instance, works on set comprehension [10]) we do not see the set on which the method operate as a complex value, rather we keep into account that it could be an heterogeneous set of objects. This aspect is particularly relevant with respect to dispatching and late binding of method invocations on a set of objects. In particular, the semantics we have defined is the first addressing the problem of method dispatching for set-oriented method execution.

The language we have defined has been exploited as the basis for reasoning about some semantic method properties, namely behavioral refinement and conflicting method definition, particularly relevant in the object-oriented context. We believe that this analysis has value beyond the specific language we have considered, since it can be generalized, for instance, to sequences of SQL-like select-update statements.

The work we have reported in this paper can be extended along several dimensions. Let us mention the possibility of reasoning about methods written in our language to decide whether they may potentially violate an integrity constraint written in a declarative style, in the same spirit of [4, 5]. We believe that our language is well suited for this kind of reasoning.

Acknowledgments We wish to thank Elisa Bertino, who suggested us to investigate the topic dealt with in the paper and who provided us several useful comments on this work.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An Object Data Model with Roles. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. Nineteenth Int'l Conf. on Very Large Data Bases*, pages 39–51, 1993.
3. M. Andries, L. Cabibbo, J. Paredaens, and J. van den Bussche. Applying an Update Method to a Set of Receivers. In *Proc. of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1995.
4. V. Benzaken and A. Doucet. Thémis: a Database Programming Language with Integrity Constraints. In *Proc. Fourth Int'l Workshop on Database Programming Languages*, Springer WICS, pages 243–262, 1993.
5. V. Benzaken and X. Schaefer. Static Integrity Constraint Management in Object-Oriented Database Programming Languages via Predicate Trasformers. In M. Aksit, editor, *Proc. Eleventh European Conference on Object-Oriented Programming*, 1997. To Appear.
6. E. Bertino and G. Guerrini. Objects with Multiple Most Specific Classes. In W. Olthoff, editor, *Proc. Ninth European Conference on Object-Oriented Programming*, number 952 in Lecture Notes in Computer Science, pages 102–126, Aarhus (Denmark), August 1995.
7. E. Bertino, G. Guerrini, and I. Merlo. A Set-Oriented Method Definition Language for Object Databases. Technical Report DISI-TR-97-11, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1997.

8. E. Bertino, G. Guerrini, and D. Montesi. Deductive Object Databases. In M. Tokoro and R. Pareschi, editors, *Proc. Eighth European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, pages 213–235, Bologna (Italy), July 1994.
9. M. Buchheit, M. Jeusfeld, W. Nutt, and M. Staudt. Subsumption Between Queries to Object-Oriented Databases. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Proc. Fourth Int'l Conf. on Extending Database Technology*, number 779 in Lecture Notes in Computer Science, pages 15–22, 1994. Extended version in *Information Systems*, 19(1).
10. P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, 149:3–48, 1995.
11. C. Chambers. Predicate Classes. In O. Nierstrasz, editor, *Proc. Seventh European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 268–296, 1993.
12. E. Chan. Containment and Minimization of Positive Conjunctive Queries in OODBs. In *Proc. of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 202–211, 1992.
13. O. Deux et al. The Story of o_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.
14. C. Laasch and M. H. Scholl. Deterministic Semantics of Set-Oriented Update Sequences. In *Proc. Ninth IEEE Int'l Conf. on Data Engineering*, 1993.
15. G. Leavens and W. Weihl. Reasoning about Object-Oriented Programs that use Subtypes. In *Proc. Fifth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications joint with Fourth European Conference on Object-Oriented Programming*, pages 212–223, 1990.
16. Servio Logic Development Corporation. *Programming in OPAL*, 1990. Version 2.0.

This article was processed using the LaTeX macro package with LLNCS style