Conflict Resolution using Derived Classes

Jürgen Schlegelmilch

Computer Science Department, Database and Information Systems,
University of Rostock
18051 Rostock, Germany
E-mail: schlegel@informatik.uni-rostock.de

Abstract. *Common object models select method implementations based on the class of the receiver. If an object belongs to several classes, more than one implementation is applicable for a method call. We present a conflict resolution strategy to get exactly one implementation per call. This is achieved by adding derived classes with method redefinitions.*

KEY WORDS: *method lookup, multiple inheritance, conflict resolution*

1. Introduction

In object-oriented systems, the object is the unit of discussion: it encapsulates some data called attributes together with the operations, called methods, that manipulate these attributes. Despite this, there are only few programming languages that are based on single objects, e.g. Self (Ungar and Smith, 1991). In contrast, class-based languages group objects with identical structure and behaviour into classes. There, classes are the unit of description, and an object is defined by the classes it belongs to; an example for a class-based language is Eiffel (Meyer, 1993). Mixed approaches are Smalltalk (Goldberg and Robson, 1983), that is object-based but has class objects providing the methods, and Oberon (Reiser and Wirth, 1992), that is class-based but binds implementations to objects.

In class-based languages, a method call to an object is performed by looking up its class membership and executing the implementation that the class associates with that method. This assumes that there is only *one* class that the method lookup has to consider. However, this is not always the case, and there are several ways to still use class-based method lookup.

1.1. Considering Inheritance

Most class-based programming languages use the inheritance hierarchy for incremental definition and substitution. Classes inherit structure definitions and methods as well as implementations for the methods from their superclasses. Inheriting from more than one class may cause conflicts because of either name clashes, or implementation clashes. The first means, attributes or methods from different superclasses have the same name, the latter describes the situation where a class inherits different implementations for one method from different superclasses. These languages require an object to belong to exactly one class that defines the structure and behaviour of its objects. We can therefore abstract from single objects to the classes, and resolve conflicts on the class level rather than the object level.

Another view on these data models is that the object belongs to a set of classes, with two conditions: First, if it is in a class, it also is in all the superclasses, and second, there is one most specific class in the set. This is equivalent to the one-class view, since this most specific class inherits from all the classes in the set, and therefore offers the same set of attributes and methods, as if the object belongs to all the classes individually. Even the implementations used for method calls to the object are the same: Either static binding is used, which executes the implementation defined by the class the variable is declared for (not the class of the object that is bound to the variable), or late binding, which selects the implementation from the most specific class anyway. Static binding is considered inferior because here the context dictates the behaviour of objects, instead of the object itself; we therefore assume late binding in the following.

1.2. Limitations of Simple Models

These simple models are very limiting: they require the user to specify a lot of classes to model common situations, because he has to define classes for all possible combinations of classes that an object might belong to. A typical example is a class hierarchy with class **Person** and several incomparable subclasses describing professions (**Employee**, **Student**, **Lawyer** . . .), hobbies (**Surfer**, **Golfer**, **IronMan** . . .) or other categories (**Patient**, **Customer**, **Politician** . . .). To model an ill person that is a successful lawyer with a golf handicap of 4, we need a class **LawyerGolfer-Patient** . . .

Some object models try to circumvent this by splitting an object into many role objects that are unrelated to each other. However, this only moves the problem into the role hierarchy: instead of being in several classes with conflicting method implementations, an object has several role objects with conflicting method implementations.

In most application models, the number of such classes is small enough, so this deficiency does not show up. This explains why programming languages use this simple model: Their applications handle only a tiny number of such categories so that the combinatorial explosion can be handled without too many problems. In contrast, databases often store data for several applications and therefore their schemas have to include all the (often orthogonal) categories of these applications. So, in the database model we have to cope with the combinatorial explosion that each single application does not encounter.

1.3. Our Approach

The context of this work is the OSCAR[1] database management system (Heuer *et al.*, 1990), which is based on the object-oriented database model EXTREM[2] (Hörner and Heuer, 1991). In this model, we drop the second condition mentioned in section 1.1: an object may still belong to more than one class, but there does not have to be one most specific class; it still has to be in all superclasses of a class it belongs to. This may cause a conflict if an object has more than one most specific class, and these classes associate different implementations with a method. This conflict has to be resolved, and we do so by introducing an additional derived class that redefines the method. This new class is derived by intersecting the conflicting classes, and therefore all concerned objects will automatically fall into it, and use the implementation defined there.

[1]**O**bject Management **S**ystem for **C**omplex **A**pplications, Approach: **R**elational
[2]**EXT**ended **RE**lational **M**odel

The paper is organized as follows: to prepare the grounds, we first describe the EXTREM model in Section 2, derived classes in Section 3 and the method lookup mechanism in Section 4. The conflict resolution strategy is described informally in Section 5, and in full detail in Section 6, including its application to an example. In Section 7 we review existing solutions in programming languages and database models, and in Section 8 we summarize our results and conclude with an outlook.

2. The EXTREM Database Model

In this section we informally introduce the database model EXTREM (Hörner and Heuer, 1991). In this model, we describe the universe of discourse with *objects* and *values*. Objects are grouped into classes, while values are partitioned into types. *Types* are either atomic, like **integer**, or structured using type constructors like **tuple** or **set**, and offer a set of functions to be applied to their values. The set of all types *Types* is organized in a lattice: A type K is a *subtype* of type L, $K \leq_T L$, iff K has more attributes or functions than L[3].

Classes are defined by a set of objects, called the *domain*. Objects are characterized by a unique identity and have to be created explicitly; the set of living objects of a class is therefore a subset of its domain, called its *extent*. All classes are placed into an inclusion hierarchy: The domain of a *specialization* class is the intersection of the extents of its superclasses, while that of a *generalization* class is the union of the extents of its subclasses. The set of all classes *Classes* is partitioned into abstract and free ones: *abstract* classes are pairwise disjoint and define each a domain of objects, while *free* classes are constructed by generalization and specialization of other classes. A free class may be subclass of only one abstract class, otherwise it would be empty.

Each class has a set of functions called *attributes*, that map its objects to values, and a set of *methods* that may be applied to its objects. The set of attributes corresponds to a tuple-structured type called the *state type*. Due to the inclusion hierarchy, objects in subclasses inherit all the attributes and methods of the superclasses. For generalizations we demand that the domain and the extent are identical, so that the class hierarchy and the type hierarchy of the state types coincide. We call this combined hierarchy the *class graph*.

To avoid name clashes, all attributes and methods are uniquely named. However, a method may be redefined in subclasses: this does not introduce a new method but only provides a new implementation for the existing one, that is to be used for objects in the subclass instead of the inherited implementation. A redefinition has to be substitutable for the implementations of its method in superclasses. Besides semantic issues, this poses restrictions on the signature: if an implementation $i : C_1 \times \cdots \times C_n \rightarrow C_0$ is redefined in a subclass by $j : D_1 \times \cdots \times D_m \rightarrow D_0$, then the contravariance rule must hold:

$$m = n \quad \wedge \quad D_0 \preceq_T C_0 \quad \wedge \quad \forall k \in [1:n] : C_k \preceq_T D_k$$

where $\preceq$ is the reflexive closure of $\prec$, the subclass relation on classes as defined in Section 4.1, and $\preceq_T$ is $\preceq \cup \, id|_{Types}$, i.e. there is no order on types.

[3]Although $\leq_T$ defines a substitutability relation, values cannot be substituted by those of subtypes due to their lack of an identity; a discussion of this is out of the scope of this paper.

Classes have a set of methods with corresponding implementations. Due to inheritance, a method is defined on more than one class, and redefinitions change for their class the implementation that is associated with the method. We can therefore view methods as entities having a mapping from classes to implementations.

3. Derived Classes

Up to now, the state type and extent of a class have been defined explicitly. However, using query languages, one can retrieve sets of objects from the database. In EXTREM, objects sharing some common structure are grouped in classes, and query results also group objects depending on shared properties, so query results are seen as special classes. In contrast to classes defined in the schema, both the structure and the domain of these classes are defined by the sequence of operators applied to other classes. We call explicitly defined classes *base classes*, and query results *derived classes*. OSCAR offers algebraic and SQL-like query languages as well as an object calculus as query languages; any of them can be used for the derived classes that we need in this paper.

Unlike base classes, derived classes do not fit into the class graph without problems because the algebra operators can change the set of objects and the state type independently. For base classes, the class and type hierarchy are identical, but not for derived classes. The solution is to place derived classes in clusters next to the base classes they are derived from, without establishing a hierarchy among them; details can be found in (Heuer and Sander, 1991). For derived classes that *can* be integrated into the class graph, we can define additional attributes and methods. Generalization classes are a special kind of derived classes, namely unions of classes.

4. The Method Lookup

Method lookup is the way to determine an implementation, given a method call to an object. Class-based approaches use classes to find implementations, while with object-based lookup each object knows what implementation to use. Of course, there cannot occur any conflicts with object-based lookup, but the size of the dispatch table is large and redundant for sets of similar objects — which are classes.

EXTREM is a class-based model, so each method has a name and a set of mappings from classes to implementations. The mappings are taken from the class graph: each explicit definition or redefinition forms a mapping from the class to the implementation given in it. This set of mappings defines a partial function from classes to implementations that needs to be extended by adding mappings for inherited methods. Of course, multiple inheritance introduces conflicts; we solve them by explicit redefinition in the subclass (Schlegelmilch, 1992). Now the function is still partial but maximal: given a class where m is defined or inherited, we can find its implementation of m by applying the function. It is therefore suitable for class-based method lookup.

4.1. Formalization

Let m be the method that is called on object o, with the mapping $\mathit{impl}_m : \mathit{Classes} \rightarrow \mathit{Impl}$, i.e. $\mathit{impl}_m(C)$ is the implementation of m in C; *impl* can be extended to sets of classes, delivering sets of implementations. In the rest of this paper we will omit m since it suffices to consider one method at a time.

Classes are ordered into a class hierarchy defined by inclusion of extents. Two relations *SPEC* and *GEN* describe the explicit subclass hierarchy: A free class C is a specialization of classes $C_1, \ldots, C_n$ iff $(C, C_i) \in$ *SPEC* $\forall i \in [1 : n]$, or it is a generalization of them iff $(C_i, C) \in$ *GEN* $\forall i \in [1 : n]$. Of course, no cycles are allowed in *SPEC* $\cup$ *GEN*. We write $C \prec D$ if $(C, D) \in (\mathit{SPEC} \cup \mathit{GEN})^+$.

Most derived classes cannot be placed into the class hierarchy with tuples in *SPEC* or *GEN*, and therefore have to be considered more specific than any other class; details can be found in (Schlegelmilch, 1992). Note that methods are type-dependent, so we cannot restrict our view to the class hierarchy. For base classes and a small set of derived classes the class and type hierarchy coincide; we call them *schema classes*. For other derived classes, the hierarchies diverge.

For object o, we define $c(o) := \{C \in \mathit{Classes} | o \in \mathit{extent}(C)\}$ the set of classes that o belongs to, and $c'(o) := \{C \in c(o) | \nexists D \in c(o) : D \prec C\}$ the set of most specific classes in $c(o)$.

4.2. *Choosing an Implementation*

With early binding, all we need is the class $c(v)$ that the variable v is declared for; the actual class of the object that is bound to v is ignored. For a call to the method m, we execute the implementation $\mathit{impl}(c(v))$. However, early binding does not fit well into an object-oriented environment. With late binding, we have to choose an implementation for method m from the set $\mathit{impl}(c'(o))$. If it contains more than one implementation, there is a method lookup conflict.

5. Conflict Resolution

The aim of conflict resolution is to have only one element in the set of conflicting implementations. This can be done by adding derived classes so that we can resolve conflicts by redefining m in the derived class. If there are n conflicting implementations, then the object is in at least n incomparable classes, and what we need is a specialization of these classes where we can redefine m, thus resolving the conflict. The important point here is that the object *automatically* has to belong to this new class — this is only possible with derived classes.

An *intersection class* is a class C derived from other classes C_i by $C := \bigcap_i C_i$. It is a proper specialization of the classes C_i, so we can make it a schema class by inserting tuples (C, C_i) into *SPEC*. If there already exists a specialization class C' of all the C_i, then we make C' a direct specialization of C by replacing all the tuples (C', C_i) by the one tuple (C', C) in *SPEC* (Fig. 1). This is allowed since the extent of C is the domain of C'. We need the intersection class even then because C' will resolve conflicts only for objects in its extent which is only a subset of its domain.

Because of the derivation, an intersection class C automatically holds all objects that are in (the intersection of) all its superclasses C_i. By redefining the method m in C, we resolve the conflict between the implementations in the C_i for all these objects. Since intersection classes are schema classes, any such class will replace its superclasses in $c'(o)$ and so reduce the number of conflicting implementations. The last problem is to determine the classes that are necessary to resolve all conflicts in a given schema.

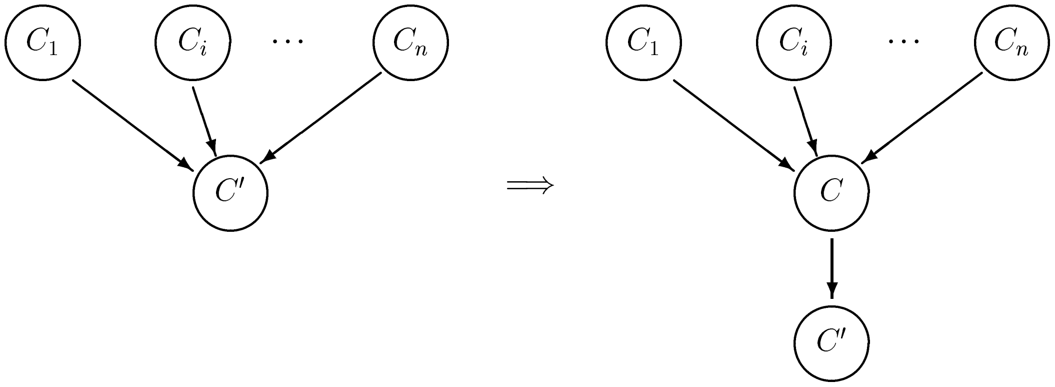

Figure 1: Insertion of a derived class C as intersection of classes C_i, $i \in [1:n]$

6. Generating the Intersection Classes

The basic idea of the algorithm is to generate all possible sets of classes, and if they could be $c'(o)$ for any object o, make sure there is only one implementation in the corresponding set of implementations for any method m. The means to ensure this are suitable derived classes, as described above. This naive approach is not optimal: it will generate far too many sets of classes, and it will add too many intersection classes because it ignores inheritance.

6.1. Observations

We start with some observations that allow us to optimize the generation of the sets of classes.

1. If we have already solved the conflict for a set of classes, doing so for any superset becomes easier: we can replace the set in the superset by the generated intersection class, thereby reducing the conflict set, because the object will automatically fall into the new class, and late binding will ignore its superclasses.

2. Since an intersection class is properly integrated into the class graph, all subclasses inherit its implementation. It is therefore necessary to start inserting intersection classes at the top of the class graph.

3. For any object o, there will be no generalization classes in the set $c'(o)$. An object appears in a generalization class if and only if it is in any of the classes that are generalized. So, if a generalization class is in $c(o)$, there will also be at least one subclass of it, and therefore the generalization will not be in $c'(o)$.

4. If a set S contains two classes with disjoint domain, there cannot be an object with $S \subseteq c(o)$, and consequently no method lookup conflict can occur. However, in EXTREM only abstract classes are pairwise disjoint, and specializations from different abstract classes. We therefore call two classes C and D *disjoint* iff $C \preceq E$ and $D \preceq F$ holds for two different abstract classes E and F.

5. In Sections 4 and 5, we considered only one method in the presentation. Since the method lookup handles each method independent from others, the conflict resolution for one method has no side effects on the conflict sets of other methods. Thus, we can resolve conflicts in sets of classes for all methods simultaneously.

6.2. The Algorithm

With these preliminaries, we can now present the resulting algorithm:

1. We generate the power-set $S_p := \mathcal{P}(\mathit{Classes})$ of the set of all classes, and then filter out superclasses from its elements: For each set $S \in S_p$, if it contains two classes C and D with $C \prec D$, we delete D from S. S_p will shrink due to duplicate elimination.

   The sets that we generate are potential sets $c(o)$ for some object o, and deleting superclasses corresponds to taking $c'(o)$ instead of $c(o)$. This is justified by late binding because this will ignore superclasses during the selection process.

2. In this step, we remove superfluous sets. These are sets that cannot be the set $c'(o)$ of most specific classes for any object o, or do not require conflict resolution for obvious reasons:

   - all sets S with $|S| \leq 1$ from S_p since for them $|\mathit{impl}_m(S)| \leq 1$ holds with any method m, so they cannot cause conflicts.
   - all sets containing a generalization class because of observation 3.
   - all sets containing disjoint classes because of observation 4.

3. The set S_p is then transformed into a list by applying a partial order $\prec_M$, defined as

   $$S \prec_M S' \iff S \subset S' \quad \vee \quad \exists C \in S' : S = \{D | (C, D) \in \mathit{SPEC}\}$$

   That means, we place a set S' after all its subsets and also after all sets S where S is the set of all direct superclasses of some $C \in S'$.[4]

   This takes the first and second observation into account, making sure that there are no side effects of intersection classes on sets found earlier in the list.

   Note that it suffices to consider only the relation *SPEC*, but not *GEN*: step 2 removes all sets containing generalization classes from the set S_p.

4. We iterate over the sets in the list in ascending order and make sure that $\mathit{impl}_m(S)$ holds only one implementation for each such set S and each method m:

   (a) First, we apply observation 1: if there already exists an intersection class for a subset of classes of S, we replace the subset by the corresponding intersection class. This has to be done in parallel for all possible replacements.

   (b) We then do the following test: if there is at least one method m in the set *Methods* of all methods where $1 < |\mathit{impl}_m(S)|$ holds, we generate a new intersection class for the set S, and ask the user for each such method m to redefine it in this class with a suitable implementation.

   ```
   if $\exists m \in \mathit{Methods} : 1 < |\mathit{impl}_m(S)|$ then
     create intersection class $C_S := \bigcap_{C \in S} C$
     $\forall m \in \mathit{Methods}$ :
       if $1 < |\mathit{impl}_m(S)|$ then redefine $m$ in $C_S$
   ```

[4]See below for a proof that $\prec_M$ is a partial order on the set S_p generated up to now.

Due to the partial order, this new class does not interfere with classes that were defined earlier in the process, or with base classes, since it is either a subclass of or incomparable to them.

Because of the fifth observation, we can resolve conflicts for all methods without side effects.

When implementing the algorithm, the steps 1, 2, and 3 can be combined into one step by generating elements of $\mathcal{P}(\mathit{Classes})$ on demand, skipping the trivial ones, and using the insertion sort algorithm to build up the list. However, we still have to build up this list, which is exponential in the number of classes $|\mathit{Classes}|$.

6.3. *The partial Order* $\prec_M$

Finally, we have to prove that $\prec_M$ is indeed a partial order on the set S_p after step 2. It is defined as the union of two relations, where the first is the well-known subset relation $\subset$ which is a partial order on any set of sets. The second relation $\prec_S$ is a partial order only on the set of sets generated by step 1 of the algorithm:

Let's assume $\prec_S$ is not a partial order, i.e. not antisymmetric:

$$\begin{array}{rc} & S \prec_S S' \quad \wedge \quad S' \prec_S S \\ \Longleftrightarrow & \exists C \in S' : S = \{D|(C,D) \in \mathit{SPEC}\} \\ & \wedge \quad \exists D \in S : S' = \{E|(D,E) \in \mathit{SPEC}\} \\ \Longleftrightarrow & \exists C \in S', D \in S, E \in S' : (C,D),(D,E) \in \mathit{SPEC} \\ \Longleftrightarrow & \exists C, E \in S' : (C,E) \in \mathit{SPEC}^* \equiv C \prec E \end{array}$$

The last assertion is false because of step 1 of the algorithm: all elements of S' are incomparable to each other under $\prec$. Consequently, the assumption was false, and $\prec_S$ is a partial order.

The second step is to show that $\prec_M := (\subset \cup \prec_S)$ is antisymmetric.

Again we assume it is not, and derive the contradiction:

$$\begin{array}{rl} & (S,S') \in \prec_M \quad \wedge \quad (S',S) \in \prec_M \\ \Longleftrightarrow & \quad (S,S') \in \subset \quad \wedge \quad S' \prec_S S \end{array}$$

Since both $\subset$ and $\prec_S$ are antisymmetric, this mixed case is the only possible one.

$$\begin{array}{rl} \Longleftrightarrow & S \subset S' \quad \wedge \quad \exists C \in S' : S = \{D|(C,D) \in \mathit{SPEC}\} \\ \Longleftrightarrow & \quad \exists C \in S', D \in S \subset S' : (C,D) \in \mathit{SPEC} \end{array}$$

As before, this is a false statement, because step 1 of the algorithm deletes D from S' if it finds such a pair (C,D). Therefore, the assumption must have been false, and $\prec_M$ is a partial order.

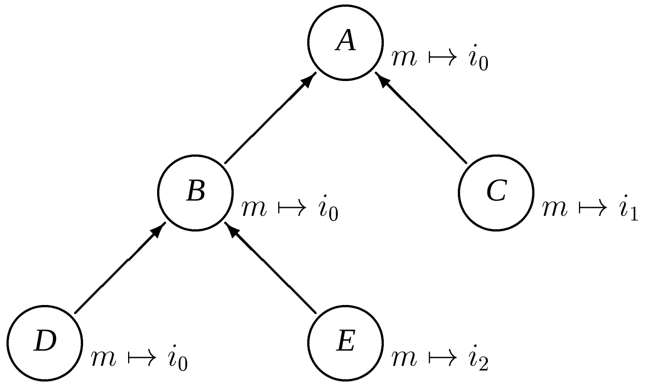

Figure 2: The class hierarchy of the sample schema

6.4. An Example

We use a simple schema with five classes $A, \dots, E$, with a class hierarchy (Fig. 2) defined by *GEN* $= \emptyset$ and *SPEC* $= \{(B,A),(C,A),(D,B),(E,B)\}$, and a method m with three implementations: *impl*$_m = \{A \mapsto i_0, C \mapsto i_1, E \mapsto i_2\}$. There are no conflicts due to multiple inheritance: B and D inherit the implementation i_0 from A. We can clearly see conflicts caused by multiple class membership, for example for objects that belong to both C and D.

The first step builds up S_p and filters out superclasses, and step 2 removes trivial sets; this gives us

$$S_p = \{\{B,C\},\{C,D\},\{C,E\},\{D,E\},\{C,D,E\}\}$$

from the original 32 sets. In the third step, we apply the partial order $\prec_M$ to this set, transforming it into the list $\langle BC, CD, CE, DE, CDE \rangle$.

We now go through the list and check *impl*(S) for each element S. The first element requires the intersection class B_C, defined as intersection of B and C. B_C is a specialization of both B and C, so we add tuples $(B\_C, B), (B\_C, C)$ to *SPEC*; an already existing specialization of B and C would become a subclass of B_C but there is none. Let us redefine our method m in B_C to use implementation i_1; it could have been any implementation, even a new i_3. The next three sets cause the generation of similar classes C_D, C_E, and D_E, for which we choose implementations i_1, i_2 and i_1 for m.

For the last set $\{C, D, E\}$, we can for the first time apply the first step of the loop: it contains the subsets $\{C, D\}$, $\{C, E\}$, and $\{D, E\}$, which are now replaced by their corresponding intersection classes C_D, C_E, and D_E, giving the set $S = \{C\_D, C\_E, D\_E\}$, with *impl*$(S) = \{i_1, i_2\}$. This requires the intersection class C_D_E, for which we choose implementation i_1.

Now we have resolved all conflicts: objects may now be in any subset of classes[5] and still have a well-defined reaction on a call to method m.

[5]with the requirement to be in all superclasses of a class it is in, of course.

6.5. An Incremental Algorithm

It does not make sense to give an incremental version of the algorithm because it would create intersection classes that could become obsolete later. Since the user can create intersection classes on his own, the classes created by the algorithm are not uniquely identifiable and can therefore only be removed by the user, thus causing unnecessary hassle.

Anyway, to integrate a new class into the class graph and resolve conflicts we simply start with a smaller set of classes, namely all sets of classes containing the new class. The algorithm then goes on as in the full version.

7. Related Work

We already mentioned in the introduction that typical programming languages require an object to have one most specific class and leave it up to the user to define all the classes he will need. In the narrow context of a single application, this is not too limiting, but for databases, the situation is different since they have to store objects for several applications. These applications generally do not agree on the exact class of an object, making role systems and object migration necessary. Therefore most of the following approaches are database models.

In the logic-based model DOL (Wieringa *et al.*, 1994), a class can have several partitions into subclasses; such a partition is called static if an object may not migrate between its classes, and dynamic otherwise. All possible combinations of classes from different partitions have to be instantiated, so an object is essentially in exactly one class, the most specific one. In addition, an object can have any number of roles which are dependent objects of special classes; roles can delegate method calls to their owner. (Wieringa *et al.*, 1994) does not discuss redefinition of methods in roles and therefore has no conflicts to resolve.

The object model of the object-oriented database system OpenODB (Ahad and Dedo, 1992) allows objects to be in more than one class, but disallows method redefinition in subclasses. Each method in OpenODB has exactly one implementation, so there cannot be any conflicts. However, method redefinition is considered one of the prime characteristics of object-oriented systems.

The semantic object model TROLL (Hartmann *et al.*, 1994) in its revised version allows objects to have several roles without requiring a most specific role. Roles may redefine methods, but in TROLL you can only specify additional actions for events instead of completely new implementations, to ensure semantic properties. This excludes optimizations in subclasses as well as omitting redundant actions, but avoids conflicts since the system simply executes all associated actions at the arrival of an event. On the other hand, order-dependencies in the set of actions are not handled.

The persistent programming language Fibonacci (Albano *et al.*, 1995) uses roles just like TROLL but allows for arbitrary redefinition. It offers two different kinds of method lookup to cope with the conflicts: the first one is equivalent to static binding since you have to name a role to see the role-specific behaviour, while the second one implements late binding but orders incomparable roles by the time the object acquired them. This results in strange semantics for objects with the same set and hierarchy of roles but different history when the late binding variant of method lookup is used.

In (Gottlob *et al.*, 1994), the authors propose an object model very similar to that of Fibonacci.

Objects consist of one class object and a set of role objects. Methods can be called for roles, showing role-specific behaviour, as well as for the object. With respect to method redefinition, this approach resembles TROLL: it allows any redefinition in roles but executes all the implementations in the most-specific roles, combining the results in one out of a fixed number of ways. Like TROLL, this avoids the combinatorial explosion only for semantic overriding but offers no help if method redefinitions for optimization are necessary; one has to make the role with the optimized implementation the one most-specific role — this is where our algorithm comes into play.

Other alternatives are rule-based implementation selection and ordering of implementations. In both approaches the method lookup has two stages: the first computes the set of applicable implementations just like conventional approaches, but the second stage uses a rule set or a total order to select one of them. In CLOS (Keene, 1989), classes have a list rather than a set of superclasses, and the list orders the classes and with them their implementations; Dylan (Bowen and Hall, 1993) uses a metric to order the superclasses and picks the implementation of the least one. Both programming languages use this approach only to resolve the conflicts caused by multiple inheritance — they still have the simpler object model.

The approach presented in this paper refines the technique introduced in (Schlegelmilch, 1992), where view classes were added without integrating them completely into the class lattice. The resulting algorithm was simpler, but required more classes because it could not use inheritance. In this paper, we are able to place the required classes into the class and type lattice, so that subclasses can profit from them.

An early version of Cecil (Chambers, 1993) supported predicate classes that are simple view classes: a predicate defines class membership, so that an object may belong to several incomparable classes. However, the resulting conflicts were not resolved but caused a runtime message.

8. Conclusion

Our aim was to allow arbitrary redefinitions in subclasses while still having late binding with the traditional method lookup strategy. We achieved this by adding derived classes with method redefinitions. The advantages of our approach are:

- The method lookup strategy is simple and straightforward, as opposed to the twofold solution in Fibonacci (Albano *et al.*, 1995). Once we know one most specific class of an object where the method in question is defined, any path upwards through the class graph will lead to the correct implementation.
- Each object shows uniform behaviour independent of the context it is used in. This is in contrast to static binding and its variants as used in Fibonacci or in (Gottlob *et al.*, 1994). Note that our algorithm can be used for role-specific behaviour, too; we simply start with $\{C|C \preceq R\}$ for a role R instead of the full set *Classes*.
- We do not restrict the kind of refinement as TROLL does, and do not have to care for order-dependencies. Also, we need not prescribe algorithms to combine sets of single results as in (Gottlob *et al.*, 1994).
- Unlike (Wieringa *et al.*, 1994), our approach avoids the combinatorial explosion of sets of subclasses: only necessary derived classes are added.

- Intersection classes are intuitive, and already supported by EXTREM; we do not need an additional concept in EXTREM just to cope with lookup conflict resolution.

However, there are still some open points:

- Since the placement of arbitrary derived classes in the class hierarchy is undecidable, we restricted our view to schema classes. Other derived classes can be ignored as long as they do not redefine methods; if they do, we revert to static binding (see (Schlegelmilch, 1992) for details).
- Once the algorithm has terminated, the generated intersection classes are not distinguishable from those intersection classes that belong to the application domain. However, all resolved conflicts have their counterpart in the application domain, so the algorithm can be seen as a consistency check: in a behaviourally consistent model it will not have to add classes.

The work presented in this article deals with single dispatch methods, i.e. implementation selection based on a single receiver object; we intend to include multi-dispatch methods in EXTREM and will extend the algorithm to handle this case as well.

References

[Ahad and Dedo, 1992] R. Ahad, D. Dedo. *OpenODB from Hewlett-Packard: a commercial object-oriented database management system*. *Journal of Object-Oriented Programming*, 5(1):31–35. 1992.

[Albano *et al.*, 1995] A. Albano, G. Ghelli, R. Orsini. *Fibonacci: A Programming Language for Object Databases*. *VLDB journal*, 4(3). 1995.

[Bowen and Hall, 1993] T. D. Bowen, K. M. Hall. *Towards a Better Understanding of Dylan*. Technical report, Laboratory for Applied Logics, University of Idaho. 1993.

[Chambers, 1993] C. Chambers. *Predicate Classes*. In *ECOOP '93 Conference Proceedings*, Universität Kaiserslautern, Institut für Informatik, Kaiserslautern, Germany. 1993.

[Goldberg and Robson, 1983] A. Goldberg, D. Robson. *Smalltalk 80: The language and its implementation*. Addison-Wesley. 1983.

[Gottlob *et al.*, 1994] G. Gottlob, M. Schrefl, B. Röck. *Extending Object-Oriented Systems with Roles*. *ACM Transactions on Information Systems*. 1994.

[Hartmann *et al.*, 1994] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, *et al. Revised Version of the Modelling Language* TROLL *(Version 2.0)*. Informatik-Bericht 94–03, Technische Universität Braunschweig. 1994.

[Heuer *et al.*, 1990] A. Heuer, J. Fuchs, U. Wiebking. *OSCAR: An object-oriented database system with a nested relational kernel*. In *Proc. of the 9th Int. Conf. on Entity-Relationship Approach, Lausanne*, pp. 95–110. Elsevier. 1990.

[Heuer and Sander, 1991] A. Heuer, P. Sander. *Classifying object-oriented query results in a class/type lattice*. In *Proceedings of the 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems, Rostock, MFDBS 91*, vol. 495 of *Lecture Notes in Computer Science*, pp. 14–28, Berlin. Springer-Verlag. 1991.

[Hörner and Heuer, 1991] C. Hörner, A. Heuer. *EXTREM — The structural part of an object-oriented database model*. Informatik-Bericht 91/5, TU Clausthal. 1991.

[Keene, 1989] S. Keene. *Object-Oriented Programming in Common LISP — A Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA. 1989.

[Meyer, 1993] B. Meyer. *Eiffel: The Language*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs. 1993.

[Reiser and Wirth, 1992] M. Reiser, N. Wirth. *Programming in OBERON — steps beyond Pascal and Modula*. acm press. Addison-Wesley, New York. 1992.

[Schlegelmilch, 1992] J. Schlegelmilch. *Inheriting Methods in OSCAR*. Master's thesis, Computer Science Dept., TU Clausthal, Erzstraße 1, D-38678 Clausthal–Zellerfeld, Germany. 1992. Available only in german.

[Ungar and Smith, 1991] D. Ungar, R. B. Smith. SELF*: The Power of Simplicity*. LISP *and Symbolic Computation*, 4(3). 1991.

[Wieringa *et al.*, 1994] R. Wieringa, W. de Jonge, P. Spruit. *Roles and dynamic subclasses: A modal logic approach*. In M. Tokoro, R. Pareschi (eds.), *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94), Bologna, Italy*, vol. 821 of *Lecture Notes in Computer Science*, pp. 31–59, Berlin. Springer-Verlag. 1994.