Methods and Views

Jan Van den Bussche
Limburgs Universitair Centrum
Department WNI
B-3590 Diepenbeek
Belgium
vdbuss@luc.ac.be

Emmanuel Waller
Université de Paris-Sud
LRI, bât. 490
F-91405 Orsay Cedex
France
waller@lri.fr

Abstract

Many papers have been written on the structural aspect of view mechanisms for object-oriented databases. A clean model that focuses specifically on the interplay between methods and views, however, is still lacking. This paper addresses this problem. First, an abstract model of *behavioral views* is introduced, from which it becomes clear that the main issue is to allow transfer of control of a computation between the root database and the view. Hence, good primitives for specifying such transfers are needed. A specification formalism for this purpose is presented in the paper and implemented on top of the formalism of method schemas. Correctness issues that arise in this context are explored, and a general result on the possibility of automated verification of behavioral views is proven.
Keywords: views, methods, object databases, formal views mechanisms, practical views specification languages, static correctness checking, inheritance, overloading, late binding

Résumé

De nombreux papiers ont été écrits sur l'aspect structurel des mécanismes de vues pour bases de données orientées-objets. Un modèle simple mettant spécifiquement l'accent sur les interférences entre les méthodes et la vue manque toutefois encore. Ce papier traite de ce problème. Tout d'abord, un modèle abstrait de *vues comportementales* est introduit, à partir duquel il s'avère clairement que le problème principal est d'autoriser le transfert de contrôle lors d'un calcul entre la base de données racine et la vue. Il y a donc besoin de bonnes primitives pour spécifier de tels transferts. Un formalisme de spécification à cette fin est présenté dans le papier et implanté au-dessus du formalisme des schémas de méthodes. Les problèmes de correction qui apparaissent dans ce contexte sont explorés, et un résultat général sur la possibilité de la vérification automatique des vues comportementales est prouvé.
Mots-clef: vues, méthodes, bases de donnés à objets, mécanismes formel de vues, langages pratiques de spécification de vues, vérification statique de correction, héritage, surcharge, liaison dynamique

1 Introduction

The importance of views in database applications is generally recognized. As Abiteboul and Bonner argued most eloquently [1], this applies no less to object-oriented database systems. The literature on OO views is considerable; for a survey, see [13].

The motivation for the present paper stems from the observation that most of the work on OO views has focused primarily on the *structural* aspects. The purpose of this paper is to contribute towards a formalization of the *behavioral* aspect. Put very roughly, we want to capture and formalize the mechanisms for deriving new methods from old methods that are useful in view specification. Designers and developers of OO view systems currently lack a guiding model in this important direction.

To make clear exactly what we have in mind with behavioral views, we first introduce an abstract model, in which a view is defined in general as a mapping. The possible inputs and possible outputs of a view mapping are carefully calibrated so as to model as faithfully as possible what happens in practice.

Towards a concrete realization of our abstract model, we propose two basic primitives: *import of values*, used to call a method in the root database and import the resulting object to the view; and *import of code*, used to import the code of a method defined in the root database and execute it in the view. Using these two primitives, combined with writing new methods in the view that can call imported methods, one can model arbitrarily complicated computations in which control can be transferred back and forth between the root database and the view. Our conviction that the primitives of value import and code import are "necessary and sufficient" is supported by extensive programming experience with one of the few industrial-strength OO view systems currently available, namely O_2Views [18].[1]

From the outset, our view specification mechanism is independent of any particular programming model. Yet, in order to be able to concretely assess the ramifications of our mechanism, we have implemented it on top of the formalism of *method schemas*. Method schemas ([6, 5]; see also [2, 11, 21]) provide a formalization of object-oriented database programming using methods, in much the same way that the classical formalism of program schemes [8] does for classical programming.

In this concrete context, we then explore the various correctness issues that arise in connection with behavioral views.

The import of code is a very powerful mechanism which can unavoidably lead to run-time errors during execution of view methods. However, we show that when the code of the methods defined in the root database is known, *consistency checking* of a view over a method schema (i.e., checking whether view code never leads to a run-time error) is no more difficult than consistency checking of an ordinary method schema. The latter problem was studied in depth by Abiteboul et al.; in general it is undecidable, but useful special cases (the *monadic* case, and the *recursion-free* case) have been identified where it becomes decidable. By our result, decidability

[1] Another such system is MultiView [15].

for these decidable cases (and others that would be discovered) carries over to the context of views.

As already indicated, our approach is characterized by an almost exclusive focus on the behavioral aspect of OO views. Consequently, our core model is independent of the particular query language used to define the population of the classes in the view. All our formalism assumes about populations is that some *class correspondence* is given, which specifies for each view class from which root classes it can be populated.

We finally mention here that our model is independent also of the particular way the class hierarchy in the view is determined. Various approaches to automated inference of the view class hierarchy have been described in the literature (e.g., [1, 16, 17]).

The further organization of this paper is as follows. Section 2 recalls the basic notions concerning method schemas. Section 3 presents an abstract model of behavioral views. Section 4 defines our extension to method schemas to realize this abstract model. Section 5 deals with consistency checking. Section 6 concludes by discussing possible extensions to our core formalism.

2 Preliminaries on method schemas

In this section we informally recall the vocabulary and basic notions concerning method schemas.[2] While this extended abstract is largely self-contained, we provide the formal definitions concerning method schemas in Appendix A.

Syntax. We distinguish between two kinds of methods: *base methods* and *coded methods*.

Base methods are extensionally stored functions which can be thought of as built-in, or as stored attributes with arguments. Base methods are declared with an output type, which is a union of class names. For example, $transport@Person, Destination : Car \cup Plane$ is the declaration of a base method *transport* which when applied to a Person and a Destination, yields either a Car or a Plane as result (which could also be an object belonging to a subclass of *Car* or *Plane*).

The most basic part of an OODB schema, namely, the class hierarchy (modeled here as a partial order on class names) and the base method declarations, is called a *pre-schema*. In a pre-schema, a base method name may appear in several different declarations. So, using OO terminology, we can have overriding of base methods. An example of a pre-schema is the following: (this example does not use overriding of base methods)

[2] Our presentation differs slightly from the original one [5]. We also make an inessential extension to the original formalism, namely union types in base method declarations, which will turn out to be useful in Section 5.

Class names:	Int, $Base\_part$, $Part$, and $Part\_list$.
Hierarchy:	$Base\_part < Part$
Base method declarations:	$sum@Int, Int : Int$;
	$head@Part\_list : Part$;
	$tail@Part\_list : Part \cup Part\_list$;
	$price@Base\_part : Int$;
	$subparts@Part : Part\_list$;
	$assembly\_cost@Part : Int$.

We next turn to coded methods. We use a simple functional programming paradigm, where all we can do in a method body is to call other methods. So a method body is nothing but a term built up from variables using method calls. A coded method definition thus consists of a declaration $m@c_1, \ldots, c_n$ with method name and input classes, and a body $\lambda x_1, \ldots, x_n.t$. A collection of coded method definitions over some pre-schema is naturally called a *behavior* for that pre-schema. An example of a behavior for our example pre-schema is the following:

$$\begin{array}{lcl} cost@Part & = & \lambda x.sum(assembly\_cost(x), cost(subparts(x))); \\ cost@Base\_part & = & \lambda x.price(x); \\ cost@Part\_list & = & \lambda x.sum(cost(head(x)), cost(tail(x))). \end{array}$$

Note the use of overriding and recursion. Of course in general a behavior will contain definitions for many different method names, not just one as in this example.

A *method schema* now consists of a pre-schema together with some behavior. We require that the set of names of base methods is disjoint from the set of names of coded methods.

Semantics. Intuitively, each class is a set of objects. Each object belongs to a unique class,[3] although of course all methods defined in the superclass and has no visible structure; all information about the object has to be obtained via its methods. As already mentioned, base methods are stored functions. If a class d is specified in the output type of a base method declaration, this means that the result of the method can be an object of class d or one of its subclasses. The mapping of class names to sets of objects, and of method names to functions is naturally called an *instance*.

The inheritance mechanism, which applies to base methods as well as to coded methods, is the standard one based on late binding. If there is no, or no unique, resolution, the method call *fails*; otherwise we say it is *well defined*.

The semantics of coded methods is defined in a simple operational manner, using rewriting of instantiated terms. For example, continuing our earlier example, if o

[3] In particular, this implies that the extension of a subclass is disjoint from, rather than a subset of, the extension of a superclass. This point of view corresponds to the situation in real OO programming, follows the formalisation of OO data modeling as given for the O_2 data model [12], and allows the cleanest treatment of inheritance and overriding.

is a Part that is not a Base_part, with assembly cost 42, and the first subpart of o is o', which is a Base_part, with price 15, then the computation sequence of $cost(o)$ starts with the following rewritings: (ℓ is the list of subparts of o)

$$\begin{aligned} cost(o) &\to sum(assembly\_cost(o), cost(subparts(o))) \to sum(42, cost(subparts(o))) \\ &\to sum(42, cost(\ell)) \to sum(42, sum(cost(head(\ell)), cost(tail(\ell)))) \\ &\to sum(42, sum(cost(o'), cost(tail(\ell)))) \to sum(42, sum(price(o'), cost(tail(\ell)))) \\ &\to sum(42, sum(15, cost(tail(\ell)))) \to \cdots \end{aligned}$$

Note that in every rewriting step we always replace the leftmost occurrence of what is called the *first redex*. A redex is an instantiated term of the simple form $m(o_1, \ldots, o_n)$, with $o_1, \ldots, o_n$ objects; the first redex is the leftmost occurring redex. Should at some point a method call fail, the whole instantiated term is replaced by the error symbol $\perp$ and the rewriting stops.

To conclude this quick review of the basic model of method schemas, we emphasize that this model is meant to serve as an abstraction of everyday database programming with inheritance, late binding, and overloading. The "functional appearance" of the way methods are treated in the model is merely a byproduct of this abstraction process, and does *not* imply that we are limiting ourselves to the functional programming paradigm. In particular, because of our independence of specific programming paradigms, we cannot (and do not) use a higher-order type system with function types such as the systems proposed in the functional programming language literature.

3 An abstract model of behavioral views

In general one might call a view *behavioral* if it contains not only classes and attributes (base methods), but also behavior (coded methods). However, just as ordinary views are only interesting if the view classes and attributes depend in some way on the classes and attributes from the root database over which the view is defined, behavioral views are only interesting if the behavior in the view depends in some way on the behavior in the root database.

Example 3.1 Consider a database of an insurance company with a class *Client*. Each Client has attributes *city*, *risk*, and *age* (and possibly others). There is a coded method *fee* which computes the insurance fee of a Client by some complicated formula depending on the age and risk of the Client.

Now suppose the company considers doubling the risk of every Client living in Paris. This can be easily simulated using a behavioral view. The view has a class *Parisien* containing all Clients living in Paris. We define a view attribute *old_risk* of Parisiens which equals simply the value of the attribute *risk* in the root database. We then define a new coded method in the view:

$$risk@Parisien = \lambda x.double(old\_risk(x)).$$

Finally we import the *code* of the method *fee* from the root database into the view.

This view has several advantages. It has all the advantages of ordinary views: for example, if a Client moves to Paris he will automatically show up in the view with the right value for *old_risk*. But it has also the additional advantage that the formula used to compute fees is exactly the same in the view as in the root database. If this formula is changed, the view will adapt automatically. Hence this view can truly be called "behavioral". ■

In the method schema model as defined in the previous section, an instance of a method schema actually depends only on the underlying *pre-schema* of that method schema. In other words, the contents of the classes and the interpretations of the base methods can change, but the behavior is fixed. However, we have just argued that fixed behavior is too restrictive for modeling behavioral views.

This motivates the following new definitions:

Definition 3.2 A *root schema* R is like a method schema, except that the bodies of coded methods have been omitted. So, from each coded method definition $m@c_1, \ldots, c_n = \lambda x_1, \ldots, x_n.t$ we simply retain the declaration $m@c_1, \ldots, c_n$.

Definition 3.3 Let R be a root schema. A *root instance of* R consists of an instance of the underlying pre-schema S, together with a behavior over S supplying a body to each coded method declaration in R. So, each coded method declaration $m@c_1, \ldots, c_n$ in R is completed into a definition of the form $m@c_1, \ldots, c_n = \lambda x_1, \ldots, x_n.t$.

Now that we have defined the input to a view (root schema and instance), what is the output of a view? We take the position that the output of a view should appear to the user of the view simply as an ordinary instance. Moreover, every method defined by the view output should appear to him as a base method, although internally its behavior may be arbitrarily complicated. The user can then use the view by adding his own coded methods on top of these "base methods". So, on the most abstract level, we define:

Definition 3.4 Let R be a root schema, and let V be a pre-schema, called the *view schema*. A *view from* R *to* V is a mapping from the root instances of R to the instances of V.

Observe that the view schema is entirely independent of the root schema, or, put differently, there are no restrictions on how the view schema can look like. This is actually quite normal; in principle any "virtual" view of the root database should be able to serve as a view. Yet, another approach, which is often encountered in the literature, is to try to force the view schema within the root schema. We believe such an approach is mistaken in that it mixes up syntax and semantics; why should

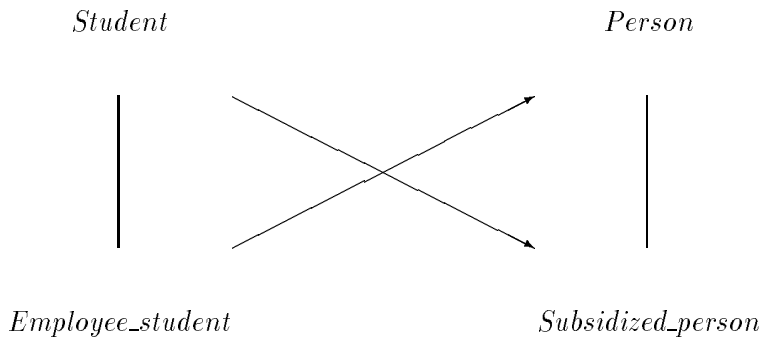

Figure 1: Illustration to Example 3.5.

the mere act of defining a view over a database have an effect on the schema of that database? Actually, as shown in the following example, there are natural situations where the view hierarchy is even the exact opposite of the root hierarchy,

Example 3.5 Consider a root database with a class *Student* and a subclass *Employee_student*. Employee_students are people who have an income and study in their spare time; proper Students are full-time students without income. In the view of the social security administration, there is a class *Subsidized_person* that is a subclass of *Person*. All Employee_students become proper Persons in the view, and all Students become Subsidized_persons in the view. The situation can be depicted in Figure 1. The arrows denote the view population. The figure clearly shows that the view hierarchy in fact turns the root hierarchy upside down. ■

Note that a view mapping can be split into two independent parts: one determining the interpretations of the methods in the view, called the *view behavior*, and one determining the contents of the classes in the view, called the *view population*.

A view population is nothing but a family of queries; for each class name c in V we specify a query q_c such that for each root instance I, $q_c(I)$ is the content of class c in the view applied to I. These queries may depend on the behavior part of I in a limited manner: for example, in a standard OO query language such as OQL, one can call methods in queries.

As already motivated by Example 3.1, the view behavior will need more dependence on the root behavior than simply calling methods and getting results back. How these dependencies can be specified is the subject of the next section.

4 Specifying view behavior

In this section we show how by adding two new primitives we can turn method schemas into a view behavior specification language. The two new primitives also make sense for programming formalisms different from method schemas.

Value import. Our first primitive is called *value import*. Recall Example 3.1, where we defined a property *old_risk* of Parisiens as being equal to the value of their property *risk* in the root database. This is an example of view method definition by value import. We will denote it syntactically as follows:

$$old\_risk@Parisien = \textbf{import value } risk.$$

In this example, *risk* is a base method in the root database, but value import can equally well be applied to coded methods in the root database. In that case the result of the coded method applied to the view objects is computed entirely in the root database, before it is imported into the view.

Code import. Our second primitive is called *code import*. Consider again Example 3.1, where we import the code of root method *fee* into the view, with the intention of executing it in the view. Since *risk* is redefined in the view, this execution will yield a different result than when we would have simply performed an **import value** on *fee*.

Syntactically, we can specify this example as follows:[4]

$$new\_fee@Parisien = \textbf{import code } fee \textbf{ with } (age : \textbf{import value}).$$

What is the purpose of the **with**-clause? It is what we call a *code import specification*. The names of the root methods called in an imported code might also be names of view methods (indeed, otherwise **import code** would make no sense). Hence, we must specify for these method calls how they are to be interpreted: simply in the view, or as an **import value**, or in turn as an **import code**. In this example, the root methods in question are *age* and *risk*: the code import specification specifies that the call to *age* in the code of *fee* is to be interpreted as an **import value**. The name *risk* is not mentioned in the specification, meaning that it is to be interpreted in the view (this is the default).

General syntax of view behaviors. Of course in general, we do not know exactly which root methods are called in an imported code. (Indeed, if we knew what was in the root code, we would not have to import it; we could just copy it verbatim into the view!) So in general we define a code import specification on all method names in the root schema. Moreover, if some code import specification specifies some root method call as **import code** as well, we need a specification for that code import in turn. However, since methods can be recursive, this specification process can go on infinitely. To solve this problem, we define one *global* collection of code import specifications, with cross-references between them, as follows:

[4] We could also have reused the name *fee* in the view instead of inventing a new name *new_fee*.

Definition 4.1 Let R be a root schema, and let M be the set of method names in R. A *global code import specification over* R is a mapping σ on an initial segment $\{1, \ldots, n\}$ of the natural numbers, such that for each $i \in \{1, \ldots, n\}$, σ_i is a partial function from M to $\{\mathbf{iv}\} \cup (\{\mathbf{ic}\} \times \{1, \ldots, n\})$.

Here, '**iv**' stands for **import value**, and '**ic**' stands for **import code**. Observe how each $i \in \{1, \ldots, n\}$ serves as an "identifier" for a concrete code import specification within the global specification, to which others can refer. It will be convenient to use the notation $i \in \sigma$ to denote that i is an identifier in the global code import specification σ.

We are now ready to define view behaviors formally:

Definition 4.2 Let R be a root schema, and let V be a view schema. A *view behavior for* V *over* R consists of a global code import specification σ over R, and a set β of *view method definitions* which can have one of the following three forms:

1. $m@c_1, \ldots, c_n =$ **import value** m', where m is a method name in V, $c_1, \ldots, c_n$ are class names in V, and m' is a method name in R;
2. $m@c_1, \ldots, c_n =$ **import code** m' **with** i, where $i \in \sigma$;
3. $m@c_1, \ldots, c_n = \lambda x_1, \ldots, x_n.t$, where t is a term built up from the variables $x_1, \ldots, x_n$ using method names in V (this is a coded method definition as in ordinary method schemes).

Corresponding to each view method declaration in V, there must be precisely one view method definition in the view behavior.

Let us fix a view behavior (σ, β) in the following definitions.

Formal semantics of view behaviors. The semantics of view behaviors is defined operationally using rewriting, as in ordinary method schemas. However now, the rewrite system is a bit more complicated due to the transfers of control between the root database and the view that can take place during the execution of a view method.

Assume given a root instance $I = (I_0, \beta_0)$ of the root schema R, where I_0 is the underlying instance and β_0 the underlying behavior.

Instantiated terms can now be built up from objects in I_0 using method names of R or the view schema V. Additionally, each internal node of an instantiated term (viewed as a tree in the obvious way) may—but does not have to—carry a *label*: this label can be either **iv** or of the form $(\mathbf{ic}, i)$ with $i \in \sigma$.

Assume further given a view population π from R to V. We now define the reductions (steps) of the rewrite system:

Definition 4.3 Let t be an instantiated term that is not an object. The *reduction of* t, denoted by $\rho(t)$, is defined as follows.[5] Let $r = m(o_1, \ldots, o_n)$ be the first redex of t. Let l be the label, if existing, of the leftmost occurrence of r in t.

If l is not there, this means reduction must be carried out in the view itself. Let c_i be the class to which o_i belongs in $\pi(I)$, for $i = 1, \ldots, n$.

- If for some i, c_i does not exist, because o_i is actually outside $\pi(I)$, then $\rho(t)$ is undefined. We denote this by $\rho(t) = \top$.
- If every c_i exists, but m is not well-defined at $c_1, \ldots, c_n$ in V, then $\rho(t)$ is also undefined. We denote this by $\rho(t) = \bot$.
- If m is well defined at $c_1, \ldots, c_n$ in V, with resolution $m@c'_1, \ldots, c'_n$, let $m@c'_1, \ldots, c'_n = s$ be the associated definition in β. We distinguish the following cases, corresponding to those in Definition 4.2:
  - s is of the form **import value** m'. Then $\rho(t)$ is obtained from t by replacing the leftmost occurrence of r in t by $m'(o_1, \ldots, o_n)$, and labeling it **iv**.
  - s is of the form **import code** m' **with** i. Then $\rho(t)$ is obtained from t by replacing the leftmost occurrence of r in t by $m'(o_1, \ldots, o_n)$, and labeling it $(\mathbf{ic}, i)$.
  - s is of the form $\lambda x_1, \ldots, x_n.t'$. Then $\rho(t)$ is obtained from t by replacing the leftmost occurrence of r in t by $t'(o_1, \ldots, o_n)$. No additional labeling is performed.

If l is **iv**, $\rho(t)$ is obtained from t by replacing the leftmost occurrence of r in t by its reduction in I (which could be an object, if m is a base method in I, or a new term, if m is a coded method), in which all internal nodes are again labeled **iv**; this yields a computation in the root database.

If l is $(\mathbf{ic}, i)$, $\rho(t)$ is also obtained from t by replacing the leftmost occurrence of r in t by its reduction in I, in which we now label each internal node as specified by σ_i.

In the last two cases, the reduction in I may not be defined; in this case $\rho(t)$ is also undefined which we denote again by $\rho(t) = \bot$.

If $\rho(t_1) = t_2$ we will denote this by $t_1 \to t_2$. The transitive closure of $\to$ is denoted by $\to^+$.

Example 4.4 In the view described in Example 3.1, suppose o is a Client living in Paris, aged 42 and having a risk factor of 11. Suppose the definition of *fee* in the root behavior is $fee@Client = \lambda x.sum(age(x), risk(x))$. In this simple example, the

[5] Actually, $\rho(t)$ depends on I, π, σ, and β, so formally we should write $\rho_{I,\pi,\sigma,\beta}(t)$.

global code import specification σ consists of only one identifier with $\sigma_1 = \{age \mapsto \mathbf{iv}\}$. (We naturally assume the class *Int* with all its methods such as *sum* and *double* to be part of the view.) Then in the view we have the following rewriting sequence:

$$new\_fee(o) \rightarrow fee^{(\mathbf{ic},1)}(o) \rightarrow sum(age^{\mathbf{iv}}(o), risk(o)) \rightarrow sum(42, risk(o))$$
$$\rightarrow sum(42, double(old\_risk(o))) \rightarrow sum(42, double(risk^{\mathbf{iv}}(o)))$$
$$\rightarrow sum(42, double(11)) \rightarrow sum(42, 22) \rightarrow 64.$$ ■

Some expressiveness considerations. The power of the code import mechanism is unleashed only in combination with the writing of new code in the view. Otherwise, code import degenerates to value import, as shown next:

Proposition 4.5 *A view behavior in which the global code import specification specifies everything either as* **iv** *or* **ic***, is equivalent to the same view behavior in which every code import has been replaced by a value import.*

The proof is based on two observations: *(i)* root coded methods eventually call root base methods; and *(ii)* code import of a root base method is equivalent to value import of that method. ■

Observation *(ii)* actually implies also that value import can be simulated using code import:

Proposition 4.6 *Every view behavior is equivalent to one that does not use value import.*

Indeed, we can add to σ the "constant" code import specification i_0 with $\sigma_{i_0}(m) = (\mathbf{ic}, i_0)$ for each root method m. We can then replace every **import value** by an **import code with** i_0, and every **iv** by $(\mathbf{ic}, i_0)$. ■

The above proposition indicates that we could have omitted value import altogether from our formalism. Of course, we have not done this because we feel that value import and code import must be highlighted as two distinct ways of depending on a root behavior. That the former can be simulated using the latter is then merely an added bonus.

To conclude this section, we emphasize once more that our notion of view behavior is the exact analogue (for methods) of the classical notion of view in relational databases; in both cases we have a certain base of information (methods in our case, relations in the classical case) on which we want to define an alternative view (in the form of new methods in our case, in the form of new relations in the classical case).

5 Verifying view specifications

A specification of a view from a root schema R to a view schema V consists of a view population and a view behavior. Various properties have to be satisfied in order for

this to correctly define a view in the abstract sense of Definition 3.4. In this section we discuss these properties and their possible automated verification.

Here is the **correctness criterion for behavioral views:** *For every view method m, for any view classes $c_1, \ldots, c_n$ at which m is well defined in V, for every root instance I, and for any objects $o_1, \ldots, o_n$ such that $o_i \in \pi(I)(c_i)$ for $i = 1, \ldots, n$, we want the existence of an object o such that $m(o_1, \ldots, o_n) \rightarrow^+ o$ and $o \in \pi(I)(d)$, for some view class d appearing in the output type of the resolution of $m@c_1, \ldots, c_n$ in V.*

This correctness criterion has many different aspects:

Closure: View methods must always return objects themselves in the view. More precisely, for any m and $o_1, \ldots, o_n$ as above, we never want $m(o_1, \ldots, o_n) \rightarrow^+ \top$ (cf. Definition 4.3), and neither $m(o_1, \ldots, o_n) \rightarrow^+ o$ for an object o not in $\pi(I)$.

Consistency: We never want $m(o_1, \ldots, o_n) \rightarrow^+ \bot$.

Termination: We never want the rewriting sequence starting from $m(o_1, \ldots, o_n)$ to be infinite.

Typing: Even if $m(o_1, \ldots, o_n)$ rewrites to an object in the view in a finite number of steps, that object should belong to one of the output classes given in the appropriate declaration of m.

Closure and typing That *closure* may be violated is a consequence of our approach in which view population and view behavior can be specified independently. Other approaches found in the literature couple these two, using some kind of default semantics which guarantees closure. Unlimited flexibility is what we get for the price (possible closure violation) we pay. In fact, this price is not so high. After all the builder of the view should know exactly what he wants to be seen in the view, so that ensuring closure should not pose a problem in practice. Example 5.1 below gives an illustration of the closure "problem".

Similar remarks apply to the *typing* aspect of view correctness.

Example 5.1 Consider a root schema with classes *Person* and *Car* and base methods *age@Person* : *Int*, *drives@Person* : *Car*, and *age@Car* : *Int*. Consider a view schema with classes *Young_person* and *Old_car*, which we populate with all Persons younger than 20 and all Cars older than 10, respectively. The view schema also has a method declaration

$$drives@Young\_person : Old\_car.$$

We define this method in the view simply as **import value** *drives*. Since a root instance may well contain Persons younger than 20 driving Cars younger than 10, closure may be violated.

We do not expect a responsible view designer to design such views; this would correspond to an attitude of "let's import some methods more or less by chance, and see whether it works". A moment's reflection reveals that there are in fact two different possible interpretations of this example, which both almost automatically satisfy closure:

1. We really want in the view only young persons driving old cars. So we correct the population of *Young_person* accordingly and we are done.

2. We want all young persons in the view, as well as the cars they drive. In this case we add a superclass *Car* of *Old_car* in the view, and declare the method *drives* more properly as $drives@Young\_person : Car$. We populate *Car* in the view with all Cars younger than 10. We can now safely do an **import value** of *drives* without violating closure. ■

Consistency and termination The aspects of *consistency* and *termination*, however, are of a different nature. Inconsistencies or non-terminations are essentially bugs which must be captured. They can show up in the root instance as well as in the view behavior. Although a bug-free view on a buggy root instance is in principle possible (if the bugs in the root are not reachable from the view), in general it is helpful and not unreasonable to assume the following:

Assumption 1 *We only consider root instances which, considered in isolation, are free of inconsistencies and non-terminations.*

Automated verification of ordinary method schemas was investigated in some depth by Abiteboul et al. [5]. The general problems are undecidable, but for specific cases, namely monadic schemas, or recursion-free schemas, useful techniques were developed which can be directly applied to help support the above assumption.

In what follows we will focus on the consistency problem, but techniques for consistency checking can typically be applied as well to termination analysis.

View consistency is not quite similar to ordinary method schema consistency. In ordinary method schemas one quantifies over all possible instances. If however we do the analogous for views, we *never* get consistency (except in trivial cases such as views having only one class or always-empty populations). Indeed, by Proposition 4.5, any non-trivial use of code import must involve a call to some view method m from within code imported from the root instance. We can then easily construct a root instance in which that code is concocted in such a way that the call to m will go wrong.

So, consistency in general is hopeless for behavioral views. As a consequence, *for the purpose of consistency checking only*, we must turn to the following situation:

Assumption 2 *The root behavior is fixed and known.*

Having set up the necessary assumptions, our goal is to substantiate the following:

Theorem 5.2 *Consistency checking of behavioral views can be reduced to consistency checking of ordinary method schemas.*

We first have to agree in what form the view specification is presented to the consistency checker. For the view behavior this is clear from Definition 4.2. For the view population, this is less clear: all we know about a view population is that it is a family of queries. The approach we take here is not to tie ourselves to one particular query language, but rather to depart from a more abstract description of the view population, which merely indicates how the view classes relate to the root classes. Thereto we define:

Definition 5.3 A *class correspondence* between root schema R and a view schema V is a binary relation from the set C_R of class names in R to the set C_V of class names in V (i.e., a subset of $C_R \times C_V$).

A population π *satisfies* a class correspondence γ if for each root instance I with underlying instance I_0, and for each $c \in C_V$, we have $\pi(I)(c) \subseteq \bigcup\{I_0(c') \mid (c', c) \in \gamma\}$.

In other words, a population satisfies γ if each object belonging to a view class c belongs to some root class c' such that $(c', c) \in \gamma$.

Note that, if the population queries are expressed in recursion-free Datalog, it is effectively decidable whether a given population satisfies a given class correspondence, using query containment tests [19, 3]. Alternatively, one may express population queries using a many-sorted logic (with the root class names as the sorts), in which the required containments can be syntactically enforced.

We can now present:

Algorithm *Consistency checking.*

Input: A class correspondence γ and a view behavior (σ, β).

Output: An ordinary method schema that is consistent if and only if the view behavior is consistent for all populations satisfying γ.

Description: We begin by "flattening" V and β. This means that we explicitly add the resolutions of all view method definitions at all classes where they are well defined. The class hierarchy in V is now no longer needed. We similarly flatten R and the root behavior.

We now define the desired method schema S. The set of class names equals γ. The intuition behind a class $(c', c) \in \gamma$ is that it stands for the objects in view class c coming from root class c'.

Every coded view method definition of β at some class[6] c, is incorporated in S at all classes of the form $(c', c) \in \gamma$.

For every method name m' in R, every class $(c', c) \in \gamma$ such that m' is well defined at c', and every $i \in \sigma$, we do the following:

1. Determine the set P' of possible output classes in the root behavior of a method call $m(x)$ with x in class c'. Let $P := \{d \in C_V \mid \exists c' \in P' : (c', d) \in \gamma\}$. Then add the base method declaration $m'_{\mathbf{iv}}@(c', c) : \bigcup_{d \in P} d$ to S.
2. (We do this only if m' is coded.) Let the definition of $m'@c'$ be $\lambda x.t'$, and let $t'^{(i)}$ be t' in which every occurring method name ℓ on which σ_i is defined, is replaced by $\ell_{\sigma_i(\ell)}$. Then we add the coded method definition $m'_{(\mathbf{ic},i)}@(c', c) = \lambda x.t'_{(i)}$ to S.

Now every view method definition in β of the form $m@c = \textbf{import value}\ m'$ is incorporated in S by adding the calls $m@(c', c) = \lambda x.m'_{\mathbf{iv}}(x)$ at each class $(c', c) \in \gamma$. So in effect we have changed a value import into a call to a virtual base method.

Similarly, every view method definition in β of the form $m@c = \textbf{import code}\ m'\ \textbf{with}\ i$ is incorporated in S by adding the calls $m@(c', c) = \lambda x.m'_{(\mathbf{ic},i)}(x)$ at each class $(c', c) \in \gamma$. ■

A detailed proof of correctness of this algorithm has been omitted from this extended abstract; however, we make the following remarks:

1. The only non-constructive step in the algorithm is the determination of the output type P' of a given method call in the root behavior. The root behavior forms, together with the root schema, an ordinary method schema. The crucial observation now is that the known techniques for method schema consistency checking [5] can be adapted for output type inference. (Actually, conversely, one can even prove in general that consistency checking and termination analysis of method schemas can be reduced to output type inference.)

2. A second crucial observation is that the algorithm does not change the "nature" of the codes used in the view or root behavior. More specifically, if the methods in the input to the algorithm are monadic, or recursion-free (the two concrete cases where consistency checking is decidable [5]), then so are the methods in the output.

We can thus conclude that we indeed have established a reduction from view consistency to ordinary consistency, at least for those cases where the latter problem is known to be decidable.[7] Hence, Theorem 5.2, under this interpretation, is proven.

[6] For the sake of notational simplicity, we consider only methods having only one argument.

[7] Technically speaking, the reduction is a Turing reduction rather than a many-one reduction.

6 Concluding remarks

We hope the formalization we have presented will be useful to designers and developers of OO view systems, who were lacking a guiding model of especially the behavioral aspects of such systems.

One can easily imagine useful variations of our two core primitives **import value** and **import code**. One such variation is what could be called *overloaded value import*: try first to resolve a given method call in the view, and if this fails, try second to import its value from the root.

One can also imagine extensions to the basic programming model provided by method schemas that are useful in view programming. One such extension is a **case** statement, by which we can discriminate the actions to be performed on some view object on the basis of the root class it comes from. Note that extensions like this one do not necessarily increase the expressive power of the formalism; for example, such **case** statements can alternatively be implemented using auxiliary methods that are added to the root database before construction of the actual view.

Our core model can be also extended to capture OO view features suggested by Abiteboul and Bonner [1], which space limitations prevented us from discussing in this extended abstract. For example:

Hiding: It is straightforward to allow for certain parts of the view schema to be hidden to the user of the view. This is desirable, e.g., for auxiliary methods used in the construction of the view. (A good example is the method *old_risk* in Example 3.1.)

Object creation: This is well understood from the theory of OO query languages (e.g., [4, 20]); one creates a new object as a function of a tuple of existing objects. For example, consider *Family* objects created as a function of (father, mother) pairs. We can easily model such an example by augmenting the root database, before actual view construction, with an intermediate layer holding the class *Family* and base method declarations *father*@*Family* : *Person* and *mother*@*Family* : *Person*. Now view construction proceeds as usual. The system of course has to keep track of what is in the real root database and what is in the intermediate layer of newly created objects (tabulation techniques for this purpose were described by Abiteboul and Bonner, who referred to newly created objects as "imaginary").

Acknowledgments

We are indebted to Cassio Souza dos Santos for his contributions to the early stages of this research. We also thank Claude Delobel for a number of inspiring discussions.

References

[1] S. Abiteboul and A. Bonner. Objects and views. In J. Clifford and R. King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, volume 20:2 of *SIGMOD Record*, pages 238–247. ACM Press, 1991.

[2] S. Abiteboul and G. Hillebrand. Space usage in functional query languages. In G. Gottlob and M.Y. Vardi, editors, *Database Theory—ICDT'95*, volume 893 of *Lecture Notes in Computer Science*, pages 439–454. Springer-Verlag, 1995.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[4] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, volume 18:2 of *SIGMOD Record*, pages 159–173. ACM Press, 1989.

[5] S. Abiteboul, P.C. Kanellakis, S. Ramaswamy, and E. Waller. Method schemas. *Journal of Computer and System Sciences*, 51(3):433–455, December 1995.

[6] S. Abiteboul, P.C. Kanellakis, and E. Waller. Method schemas. In *Proceedings 9th ACM Symposium on Principles of Database Systems*, pages 16–27. ACM Press, 1990.

[7] G. Castagna. *Object-Oriented Programming: A Unified Foundation.* Birkäuser, 1997.

[8] B. Courcelle. Recursive applicative program schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 9. Elsevier, 1990.

[9] C. Delobel, M. Kifer, and Y. Masunaga, editors. *Deductive and Object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[10] G. Guerrini, E. Bertino *et al.* A formal model of views for object oriented database systems. *Theory and Practice of Object Systems*, 3(3), 1997.

[11] G.G. Hillebrand, P.C. Kanellakis, and S. Ramaswamy. Functional programming formalisms for OODBMS methods. In A. Dogac et al., editors, *Advances in Object-Oriented Database Systems*, volume 130 of *NATO ASI Series F: Computing and Systems Sciences*, pages 73–99. Springer, 1994.

[12] P. Kanellakis, C. Lécluse, and P. Richard. The O_2 data model. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an object-oriented database system: The story of O_2*, chapter 3. Morgan Kaufmann, 1992.

[13] W. Kim and W. Kelley. On view support in object-oriented database systems. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pages 108–129. ACM Press, 1995.

[14] R. Motschnig-Pitrik Requirements and comparison of view mechanisms for object-oriented databases. *Information Systems*, 21(3):229-252, 1996.

[15] H.A. Kuno and E.A. Rundensteiner. The MultiView OODB view system: Design and implementation. *Theory and Practice of Object Systems*, 2(3):202–225, 1996.

[16] E.A. Rundensteiner. A classification algorithm for supporting object-oriented views. In *Proceedings 3rd International Conference on Information and Knowledge Management*, pages 18–25. ACM Press, 1994.

[17] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In Delobel et al. [9], pages 189–207.

[18] C. Souza dos Santos. Design and implementation of object-oriented views. In N. Revell and A. Min Tjoa, editors, *Database and Expert Systems Applications*, Lecture Notes in Computer Science, pages 91–102. Springer, 1995.

[19] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, 1989.

[20] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44(2):272–319, 1997.

[21] E. Waller. Schema updates and consistency. In Delobel et al. [9], pages 167–188.

A Formal definitions concerning method schemas

Syntax. We use the following kinds of syntactical symbols: *class names*; *method names*; and *variables*. Each method name has an associated *arity*, a natural number.

A *declaration* is an expression of the form

$$m@c_1, \ldots, c_n$$

where m is a method name of arity n, and $c_1, \ldots, c_n$ are class names.

A *base method declaration* is an expression of the form

$$m@c_1, \ldots, c_n : d_1 \cup \cdots \cup d_\ell$$

where $m@c_1, \ldots, c_n$ is a method declaration and $d_1, \ldots, d_\ell$ are class names.

A *pre-schema* is a triple $(C, \leq, \Sigma_0)$, where C is a set of class names, $\leq$ is a partial order on C, and Σ_0 is a set of base method declarations with class names from C, such that there are no two different base method declarations for the same method declaration.

Terms are inductively defined as follows:

1. Each variable is a term;
2. If $t_1, \ldots, t_n$ are terms, and m is a method name of arity n, then $m(t_1, \ldots, t_n)$ is a term.

A *coded method definition* is an expression of the form

$$m@c_1, \ldots, c_n = \lambda x_1, \ldots, x_n.t$$

where $m@c_1, \ldots, c_n$ is a method declaration, $x_1, \ldots, x_n$ are distinct variables, and t is a term in which only these variables occur.

Let $S_0 = (C, \leq, \Sigma_0)$ be a pre-schema. A *behavior for* S_0 is a set Σ_1 of coded method definitions with classes from C such that

1. the set M_0 of method names occurring in Σ_0 is disjoint from the set

$$M_1 = \{m \mid \text{there is a method declaration } m@\ldots \text{ occurring in } \Sigma_1\}.$$

2. all method names occurring in Σ_1 are in $M_0 \cup M_1$.
3. there are no two different coded method definitions for the same method declaration.

The elements of the set M_0 are called *base method names*, and those of M_1 *coded method names*.

A *method schema* S consists of some pre-schema S_0 together with some behavior Σ_1 for S_0. Let us fix a method schema S in what follows.

Let $c_1, \ldots, c_n$ be class names in S, and let m be a method name. Then m is said to be *well defined at* $c_1, \ldots, c_n$ *in* S if there exists a *unique* method declaration $m@c'_1, \ldots, c'_n$ occurring in S such that $c_i \leq c'_i$ for $i = 1, \ldots, n$. If this is the case, $m@c'_1, \ldots, c'_n$ is called the *resolution of* m *at* $c_1, \ldots, c_n$ *in* S.[8]

Note that, instead of using only the class of the first argument (the "receiver") for determining method resolution, we use all arguments simultaneously (this mechanism is known as "multi-methods" [7]).[9]

[8] The resolution of m at $c_1, \ldots, c_n$ is part of either a base method declaration or a coded method definition. It will be convenient to refer to this base declaration or coded definition also as the resolution of m at $c_1, \ldots, c_n$.

[9] The use of multi-methods instead of the more common receiver-based methods is more a design choice than a crucial aspect of method schemas.

Semantics. Formally, we assume given a universe $\mathcal{O}$ of *objects*.

An *instance* I of S is a mapping on the class and base method names of S, such that:

1. For each class name c, $I(c)$ is a finite subset of $\mathcal{O}$, such that $c \neq c'$ implies $I(c) \cap I(c') = \emptyset$. The union $\bigcup_{c' \leq c} I(c')$ is denoted by $I^*(c)$. If $o \in I(c)$ then we say that *o belongs to class c*.

2. For each base method name m of arity n, $I(m) : \mathcal{O}^n \to \mathcal{O}$ is a partial function. Let $o_1, \ldots, o_n \in \mathcal{O}$ and let c_i be the class name to which o_i belongs, for $i = 1, \ldots, n$. Then $I(m)(o_1, \ldots, o_n)$ is defined if and only if m is well defined at $c_1, \ldots, c_n$. In this case, if the resolution of m at $c_1, \ldots, c_n$ is
$$m@c'_1, \ldots, c'_n : d_1 \cup \cdots \cup d_\ell$$
then $I(m)(o_1, \ldots, o_n)$ must be an element of $I^*(d_1) \cup \cdots \cup I^*(d_\ell)$.

In what follows we fix an instance I.

Instantiated terms over I are defined inductively just like terms, except that we start from objects in I instead of from variables.

A *redex* is an instantiated term of the form $m(o_1, \ldots, o_n)$, where m is a method name and o_1, ..., o_n are objects.

Let t be an instantiated term that is not an object. The *first redex* of t is inductively defined as follows:

1. If t is a redex, it is its own first redex.

2. If t is not a redex, and of the form $m(t_1, \ldots, t_n)$, then the first redex of t is the first redex of t_i, where $i \in \{1, \ldots, n\}$ is the smallest such that t_i is not an object.

Let t be an instantiated term over I that is not an object. The *reduction of* t, denoted by $\rho(t)$, is defined as follows. Let $r = m(o_1, \ldots, o_n)$ be the first redex of t, and let c_i be the class name to which o_i belongs, for $i = 1, \ldots, n$.

- If m is not well-defined at $c_1, \ldots, c_n$, then $\rho(t)$ is undefined. We denote this by $\rho(t) = \perp$.

- If m is a base method well-defined at $c_1, \ldots, c_n$, then $\rho(t)$ is obtained from t by replacing the leftmost occurrence of r in t by the object $I(m)(o_1, \ldots, o_n)$.

- If m is a coded method well-defined at $c_1, \ldots, c_n$, let
$$m@c'_1, \ldots, c'_n = \lambda x_1, \ldots, x_n.t'$$
be the corresponding resolution. Let $t'(o_1, \ldots, o_n)$ be the ground term obtained from t' by replacing each occurrence of x_i by o_i, for $i = 1, \ldots, n$. Then $\rho(t)$ is obtained from t by replacing the leftmost occurrence of r in t by $t'(o_1, \ldots, o_n)$.

(Note that $\rho(t)$ depends on the method schema S and the instance I, so formally we should have written $\rho_{S,I}(t)$.)