University of Alberta

On Separation between Interface, Implementation, and Representation in Object DBMSs

by

Yuri Leontiev, M. Tamer Özsu, Duane Szafron

Technical Report TR 98–02
March 1998

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

On Separation between Interface, Implementation, and Representation in Object DBMSs

Yuri Leontiev, M. Tamer Özsu, Duane Szafron
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E1
{yuri,ozsu,duane}@cs.ualberta.ca

Abstract

In this paper we present a model that supports a clean separation between the concepts of interface, implementation, and representation. We present several problems that are difficult to solve in the absence of such separation and describe how the proposed model can be used to provide a solution. We also describe the principles that can be used to implement the proposed model in an existing object-oriented database management system.

Keywords: object-oriented, interface, implementation, representation, type, class, database programming languages

1 Introduction

In the past two decades, object-oriented technology has been extensively used in design and development of database management systems. This technology has been introduced to meet the challenging requirements posed by current applications.

One of the major advantages of object-oriented software technology is its strong support for code reuse. While code reuse is important in application programs, it is even more important in database programming. Data stored in many databases far outlives the application programs originally written to process it. Therefore, incremental application development is the only strategy that helps to avoid huge reinvestments in the development of new applications.

Object-oriented technology supports code reuse via the concept of *inheritance*, where a subclass (subtype) automatically reuses the code written for its superclasses (supertypes). Major object-oriented design strategies associate classes (types) with real-world concept descriptions, thus providing support for concept-oriented modeling.

Surprisingly enough, this design principle limits code reuse since in current object-oriented database programming languages and systems, a class (type) describes not only the concept it denotes, but also the interface, implementation (code), and representation (data format) of its instances. Therefore, it is impossible to use the same implementation or representation for two unrelated concepts without repeating the code related to it.

The problem described above stems from the fact that the concepts of *subtyping* and *subclassing* are unified in most of the current database systems (as discussed in Section 2). A clean separation of these concepts allows for much better code reuse, as will be shown in Section 5.

Some of the important problems that can be addressed with the help of the mechanism proposed in this paper include extensibility, incremental data evolution, and foreign data integration.

1.1 Extensibility

Modern database systems often provide support for schema extension and manipulation. The ability of a programmer or database administrator to introduce new classes (types) into an existing schema becomes increasingly more important as new application areas arise. For example, CAD/CAM applications generally require the addition of a new class (type) into the schema once a new component is introduced into the design process. Medical applications also require schema changes as new diagnostic equipment and new, non-standard methods of treatment become available.

While schema evolution capabilities are present in most current database systems, both traditional and object-oriented, there is one "forbidden" section of the schema that severely restricts possible extensions. This section consists of the so-called *primitive* classes (types). It usually includes boolean, numeric, character, and string types. Dates and certain multimedia data types are sometimes included in this section as well. Purely object-oriented systems (such as GemStone [BMO+89]) usually allow the programmer to define new methods for the primitive types, but disallow the addition of new primitive types. Hybrid systems (such as ObjectStore [LLOW91]) support the addition of new primitive types (sometimes termed as *value types*), but severely limit the addition of any user-defined methods that work on them, and disallow user-defined placement of the newly defined primitive types in the type (class) hierarchy. Traditional database systems completely seal the primitive types from any kind of extension or modification.

These restrictions have an obvious justification: representation of primitive types as well as their operations are internally optimized, and schema changes that might affect the correctness of such optimizations are disallowed. However, the ability of a database management system designer to foresee all types that

need heavy optimization is questionable. For example, just a decade ago addition of an audio or an image type into the primitive type system would not even be considered, while nowadays such additions seem to be almost inevitable. There are two major reasons. First, operations on audio and video data need to be heavily optimized. Second, the audio and video formats (representations) are specified by standards which are not under the control of the database designer. The specifics of a particular application might also require an optimization of some non-primitive types. For example, a scientific application might need to optimize matrix or complex number types, so a special representation is necessary.

It is therefore desirable for a database system to allow the addition of new primitive, heavily optimized types and to support the optimization of types that already exist in the system. Such a task can be achieved by the introduction of low-level primitives that will be discussed in Section 5.

1.2 Incremental data evolution

One of the major effects of schema evolution on the database is the necessity to perform *data evolution*. For example, changing the attribute "Name" of a Person class to three attributes ("FirstName", "SecondName", and "MiddleName") requires changing all instances of this class in the database. The straightforward approach to this problem is to find all instances of the changed class and change them right after the schema change is made. However, this approach suffers from two major drawbacks: first, it is very inefficient and makes the database virtually inaccessible while the change is being made. Second, the programs that were written for the old schema will not work for the new one. On the other hand, *incremental data evolution* (or *lazy data evolution*), only changes objects when they are requested by an application. This approach eliminates the first of these drawbacks, but still suffers from the second one. The reason for this is the fact that an old application might create new objects in the old form [Höl93]. Thus, a mechanism that allows coexistence of old and new data is required to overcome this drawback. Such a mechanism will be described in Section 5.

1.3 Foreign data integration

One of the hot topics in today's database systems research is that of *interoperability* [Weg96], [Kon93]. Since enormous amounts of data are currently stored in traditional databases and plain files, the ability of new database systems (including object-oriented ones) to use such "foreign" data is extremely important. However, support for such interaction is quite limited in current database systems. An indirect confirmation of this fact can be seen in the number of frameworks proposed to achieve this task (e.g. [LP97], [BY95], [BPK95], and many others). These frameworks are not a part of any database system but rather they are built on top of such systems. Most of the proposed frameworks are object-oriented or object-based, which supports the opinion that an object-oriented database system can have built-in facilities for interoperability.

While the interoperability problem is not the topic of this paper, the mechanisms described here can be used to support foreign data integration in object-oriented databases. We believe this to be an important step towards the development of truly interoperable and extensible object-oriented database management systems.

1.4 Organization of the paper

The paper is organized as follows: Section 2 reviews current database systems and programming languages from the point of view of the separation between interface, implementation, and representation. In Section 3 a model that provides such separation is proposed and an example of its use is presented. Section 4 provides

a description of dispatch principles in the presented model according to two different dispatch paradigms: single and multiple dispatch. Then Section 5 shows how the model proposed in the paper can solve the problems discussed in the current section. Section 6 describes a method that can be used to shield an ordinary user from most of the conceptual complexities of the proposed model and outlines a way that the model can be implemented in a sufficiently powerful database management system such as ObjectStore. Finally, Section 7 concludes the paper and outlines the directions for future research.

2 Related work

Before we introduce the main concepts of the proposed model, we will discuss the issues related to the separation between the notions of interface, implementation, and representation in current database systems and object-oriented programming languages. We start with the controversy surrounding the meaning of the notions of *type* and *class*, which are considered to be central to object-oriented systems. Both *type* and *class* are used by different researchers to refer to one or more of the following notions:

1. A real-world concept (CONCEPT)
2. Programmatic interface (INTERFACE)
3. Implementation of the programmatic interface (IMPLEMENTATION)
4. Internal machine representation (REPRESENTATION)
5. A factory for creation of instances (FACTORY)
6. The maintainer of the extent (the set of all instances) (EXTENT)

For example, in GemStone [BMO+89], *class* refers to all of the above notions. In ObjectStore [LLOW91], *type* refers to the notions (INTERFACE) and (REPRESENTATION), while *class* refers to the notions (CONCEPT), (IMPLEMENTATION) and (FACTORY). Alternately, the *type* of an object of a particular *class* is implicitly defined by that *class*, so it may also be argued that *class* in ObjectStore actually refers to all notions except (EXTENT). In O_2 [LRV92], *type* refers to the notions (INTERFACE) and (REPRESENTATION), while *class* is a special kind of *type* that refers to the notions (CONCEPT) through (EXTENT). In Iris [Fis89], *type* refers to the notions (CONCEPT), (REPRESENTATION), (FACTORY), and (EXTENT); the (INTERFACE) and (IMPLEMENTATION) are defined with respect to the *type* but are not a part of the *type*. In VODAK [VOD95] (which is built on top of ObjectStore but has a different data model), *type* refers to the notions (INTERFACE), (IMPLEMENTATION), and (REPRESENTATION), while *class* refers to the notions (CONCEPT) and (FACTORY). *Type* in VODAK has two components: the *interface*, referring to the notion (INTERFACE), and the *implementation*, referring to the notion (IMPLEMENTATION). Both parts of the VODAK *type* can define the representation. In relational database systems, *type* refers to the notions (INTERFACE) and (REPRESENTATION).

The above analysis shows that none of these systems (except for VODAK) distinguishes between the notions (INTERFACE), (IMPLEMENTATION), and (REPRESENTATION), i.e., between the programmatic interface, its implementation, and the internal representation. The separation in VODAK is only partial, since both components of the VODAK type do not exist independently of the type they belong to. The database model AQUA [LMS+93] completely separates the notions (INTERFACE) and (REPRESENTATION); however, the issues related to the notion of (IMPLEMENTATION) and to dispatch are not fully developed.

Surprisingly, there seems to be much more consensus on the meaning of the derived terms *subtyping* and *subclassing* ([LP91], [Tai96]). *Subtyping* usually refers to the relationship between the interfaces (INTERFACE), while *subclassing* refers to the relationship between representations (REPRESENTATION) and

implementations of the interface (IMPLEMENTATION). While these two relationships are often unified, there are languages that distinguish between them (e.g. [CL95], [CL97], [RTL+91], [SOM93], [BBB+93], [ACO85], [DGLM95], and [Fra97]).

Some object-oriented programming languages do provide much cleaner separation between interface and implementation. Languages Cecil [CL95] and its ancestor BeCecil [CL97] provide a clean separation between these two notions, as well as between the notions of subtyping and subclassing. However, no separation between representation and implementation is provided. Emerald [RTL+91] also distinguishes between types (interfaces) and classes (implementations), but does not support implementation inheritance (subclassing). Sather [SOM93] distinguishes between interface and implementation inheritance, but does not completely separate interface (type) from implementation (class). Lagoona [Fra97] completely separates object interface (called *a category*) from representation (type). However, implementation (a set of methods) is tied to the representation (type). Inheritance in Lagoona works differently for all of these three concepts: interfaces (categories) support multiple inheritance, representations (types) support single inheritance, and implementations (methods) can not be inherited. Theta [DGLM95] is another language that cleanly separates interface (type) from implementation (class), but ties together implementation and representation. Other languages that provide partial separation between interface and implementation include TM [BBB+93] and Galileo [ACO85].

In the next section, we will describe a model that completely separates interface, implementation, and representation by providing *implementation types* that are used to describe the internal representation of data and *implementation functions* that operate on them.

3 The model

In the model we propose, the separation between interface, implementation, and representation is total. We will use the term *type* to refer to the notions (CONCEPT) and (INTERFACE), the term *implementation type* to refer to the notion (REPRESENTATION), and the term *class* to refer to the notions (FACTORY) and (EXTENT). The notions of (EXTENT) and issues related to extent maintenance are marginal in traditional programming languages; however, they are crucial for databases and persistent programming. The notion (IMPLEMENTATION) is supported by *behavior – function – implementation function* bindings to be discussed later. Thus, a *type* (denoted by the prefix `T_`) defines a programmatic interface, while an *implementation type* (denoted by a prefix `IT_`) defines an internal representation. When a type meets an implementation type, a class (denoted by the prefix **C_**) capable of producing new instances is created. Type and implementation type hierarchies are totally independent. Usually, there is one-to-one correspondence between types `T_X`, implementation types `IT_X`, and classes **C_X**. However, it is possible to use the same implementation type for classes of unrelated types. This occurs when two unrelated types use the same or related internal representations. It is also possible to implement a type using more than one implementation type. Therefore, objects of the same type can have different internal representations.

Example 3.1 Let us assume that we are dealing with the banking system of a bank called MegaBank that has two types of accounts: chequing and savings. It is possible to draw a cheque on a chequing account, while drawing a cheque on a savings account will result in a substantial service charge. An account stores the balance (and possibly some other information, such as owner and account number). It is also possible to have a term deposit in MegaBank. For a term deposit, the system stores the same information as for an account with the addition of the term length. We will also assume that MegaBank has a partner bank MiniBank and can handle its accounts by interaction with the banking system of MiniBank. In order to model this

situation, we will define types `T_Account`, its subtypes `T_ChequingAccount` and `T_SavingsAccount`, and an unrelated type `T_TermDeposit`. We will also define implementation types `IT_Account` and its implementation subtype `IT_TermDeposit` to handle MegaBank's accounts, and an implementation type `IT_PartnerAccount` to handle MiniBank's accounts. Note that while `IT_TermDeposit` defines only one field (attribute), data of this implementation type will have at least two fields (attributes) since at least one field is inherited from `IT_Account`. Type definitions for this example are shown in Figure 1, implementation type definitions in Figure 2, and class definitions in Figure 3. Note that the type and implementation type hierarchies here are independent of each other; any type can combine with any implementation type to produce a class.

```
TYPE T_Account
     BEHAVIOR B_balance() : T_Natural :: FUNCTION F_balance END END
     BEHAVIOR B_setBalance(T_Natural amount) :: FUNCTION F_setBalance END END
     BEHAVIOR B_deposit(T_Natural amount) ::
          FUNCTION SELF.B_setBalance(SELF.B_balance + amount); END
     END
     BEHAVIOR B_withdraw(T_Natural amount) ::
          FUNCTION
               IF SELF.B_balance >= amount THEN SELF.B_setBalance(SELF.B_balance − amount);
               ELSE RAISE "Not enough money";
          END
     END
     BEHAVIOR B_drawCheque(T_Cheque cheque) END
END
TYPE T_ChequingAccount
     SUPERTYPES T_Account;
     BEHAVIOR B_drawCheque(T_Cheque cheque) ::
          FUNCTION
               SELF.B_withdraw(cheque.B_amount);
               cheque.B_account.B_deposit(cheque.B_amount);
          END
     END
END
TYPE T_SavingsAccount
     SUPERTYPES T_Account;
     BEHAVIOR B_drawCheque(T_Cheque cheque) ::
          FUNCTION
               SELF.B_withdraw(cheque.B_amount);
               SELF.B_withdraw(ServiceCharge);
               Bank.B_deposit(ServiceCharge);
               cheque.B_account.B_deposit(cheque.B_amount);
          END
     END
END
TYPE T_TermDeposit
     BEHAVIOR B_balance() : T_Natural :: FUNCTION F_balance END END
     BEHAVIOR B_setBalance(T_Natural amount) :: FUNCTION F_setBalance END END
     BEHAVIOR B_term() : T_TimeSpan :: FUNCTION F_term END END
     BEHAVIOR B_setTerm(T_TimeSpan span) :: FUNCTION F_setTerm END END
     BEHAVIOR B_renew ::
          ...
     END
     ...
END
```

Figure 1: Type definitions for the banking system example

```
IMPLEMENTATION TYPE IT_Account
    FIELD IT_Natural balance;
    FUNCTION F_balance() : IT_Natural :: ACCESS balance END
    FUNCTION F_setBalance(IT_Natural) :: SET balance END
    ...
END
IMPLEMENTATION TYPE IT_TermDeposit
    SUPERTYPES IT_Account;
    FIELD IT_TimeSpan term;
    FUNCTION F_term() : IT_TimeSpan :: ACCESS term END
    FUNCTION F_setTerm(IT_TimeSpan) :: SET term END
    ...
END
IMPLEMENTATION TYPE IT_PartnerAccount
    STRUCTURE { MiniBankAccountNumber remoteNumber; } END
    FUNCTION F_balance() : IT_Natural ::
        IMPLEMENTATION FUNCTION
            { return miniBankSystemInterface.getBalance(remoteNumber); }
        END
    FUNCTION F_setBalance(IT_Natural amount) ::
        IMPLEMENTATION FUNCTION
            { miniBankSystemInterface.setBalance(remoteNumber,amount); }
        END
    END
    ...
END
```

Figure 2: Implementation type definitions for the banking system example

```
CLASS C_SavingsAccount
    TYPE T_SavingsAccount;
    IMPLEMENTATION TYPE IT_Account;
END
CLASS C_ChequingAccount
    TYPE T_ChequingAccount;
    IMPLEMENTATION TYPE IT_Account;
END
CLASS C_PartnerSavingsAccount
    TYPE T_SavingsAccount;
    IMPLEMENTATION TYPE IT_PartnerAccount;
END
CLASS C_PartnerChequingAccount
    TYPE T_ChequingAccount;
    IMPLEMENTATION TYPE IT_PartnerAccount;
END
CLASS C_TermDeposit
    TYPE T_TermDeposit;
    IMPLEMENTATION TYPE IT_TermDeposit;
END
```

Figure 3: Class definitions for the banking system example

The actions in our model are performed by *behaviors* (messages; denoted by the prefix *B_*) applied (sent) to objects. When a behavior is applied, it is *dispatched* to an appropriate *function* (denoted by the prefix *F_*). The dispatch is done according to the *behavior-to-function bindings*[1]. For instance, in the above example the behavior *B_balance* is bound to the function *F_balance* on the type `T_Account`. An ordinary (high-level) function consists of behavior applications (e.g. an anonymous function bound to the behavior *B_withdraw* on the type `T_Account`). High-level functions are independent of implementation types of their arguments as they are concerned with types (interfaces) only. From the software engineering perspective for code maintenance purposes it is best to make as few functions as possible dependent on the actual implementation (i.e. implementation type). However, when the actual data representation must be accessed by a function, then that function must be bound to an *implementation function* defined on the implementation type describing the data representation to be accessed. For example, the function *F_balance* is bound to an anonymous field-access implementation function on the implementation type `IT_Account`. The same function is bound to a low-level implementation function on the implementation type `IT_PartnerAccount`. The implementation functions are written in a lower-level language that is capable of accessing data, thus providing support for low-level optimization and interoperability.

In our model, types define behaviors. Functions are bound to behaviors on types, and subtypes inherit behaviors and bindings from their supertypes. In the above example, the behavior *B_balance* is defined on the type `T_Account` and it is also bound to the function *F_balance* on that type. This behavior along with its binding is inherited by the types `T_ChequingAccount` and `T_SavingsAccount` and is used in the code of the functions bound to the behavior *B_drawCheque* on these types.

Types together with behaviors and high-level functions provide a traditional object-oriented framework that supports inheritance, overriding, and dynamic binding. For example, the behavior *B_drawCheque* originally defined on the type `T_Account` is implemented differently for chequing and savings accounts. Thus, our model is an extension of the traditional object-oriented models rather than a substitute for them. As will be shown in Section 6, this allows us to provide an ordinary user with a familiar framework while giving the database administrator all the power of the presented model.

Implementation types define functions and bind implementation functions to them. Functions and implementation function bindings are inherited by implementation subtypes of a particular implementation type. The structure of the internal data representation is also inherited between implementation types, so that an implementation type always has at least the fields that are defined by its implementation supertypes. For example, the implementation type `IT_TermDeposit` inherits the function *F_balance* and the field it accesses from the implementation type `IT_Account`. The data of the implementation type `IT_TermDeposit` have the same fields as the data of the implementation type `IT_Account` plus the field that stores the term length. The latter is accessed by an anonymous field access implementation function bound to the function *F_term*.

The implementation type `IT_PartnerAccount` illustrates additional capabilities of implementation types. The implementation of the function *F_balance* by this implementation type is designed to access the remote MiniBank system. The code is written in a lower-level language (in this case, a dialect of C++). It uses the field `remoteNumber` that contains "foreign" data. This field can not be directly accessed by the high-level code. Note that while the implementation and representation of the implementation types `IT_Account` and `IT_PartnerAccount` are fundamentally different, they are both used to implement the interface defined by the types `T_ChequingAccount` and `T_SavingsAccount`. The same high-level code transparently deals with both ordinary and remote accounts.

The banking system example illustrates the need for the total separation of type and implementation type hierarchies. The interface defined by the type `T_ChequingAccount` has two unrelated implementations:

[1] The particulars of the dispatch mechanism will be discussed in Section 4.

the one defined by the implementation type `IT_Account` and the one defined by the implementation type `IT_PartnerAccount`, yet the high-level code is transparently reused for both of them. At the same time, unrelated interfaces defined by the types `T_ChequingAccount` and `T_TermDeposit` are implemented by related implementation types `IT_Account` and `IT_TermDeposit` that allows the latter to transparently reuse the field definitions of the implementation type `IT_Account`. It would not be possible to achieve the degree of code reuse illustrated by this example if the type and implementation type hierarchies were not totally independent of each other.

A class can use any type `T_X` and any implementation type `IT_X`. The only restriction here is the requirement that if the type `T_X` defines or inherits a behavior *B_alpha*, then there exists one and only one most specific binding of *B_alpha* to a function *F_alpha*, and there exists one and only one most specific binding of *F_alpha* to an implementation function *IF_alpha* on the implementation type `IT_X` (see Section 4 for the definition of the most specific binding). In other words, when a class is created, the system makes sure that the dispatch of every behavior defined on, or inherited by its type, is unambiguous and yields an implementation function.

To sum up, the model we are proposing consists of:

1. The set of types T with a partial order $\leq_t$ (subtyping). A type `T_X` can define a set of behaviors ($\{B\_alpha_i\} = explicitelyDefined(\texttt{T\_X})$). It can also bind behaviors to functions ($F\_alpha = explicitBinding(B\_alpha, \texttt{T\_X})$)[2].
2. The set of implementation types IT with a partial order $\leq_{it}$ (implementation subtyping). An implementation type `IT_X` can define a set of functions ($\{F\_alpha_i\} = explicitelyDefined(\texttt{IT\_X})$). It can also bind functions to implementation functions ($IF\_alpha = explicitBinding(F\_alpha, \texttt{IT\_X})$). An implementation type can also define additional data structures[3].
3. The set of classes C. Each class **C_X** has an associated type $associatedType(\mathbf{C\_X}) = \texttt{T\_X}$ and an associated implementation type $associatedImplementationType(\mathbf{C\_X}) = \texttt{IT\_X}$.
4. The set of behaviors B.
5. The set of functions F. A function may or may not have high-level code.
6. The set of implementation functions IF. An implementation function either has low-level code or it is a field access/storage implementation function.
7. The set of *objects* O. Each object *o* belongs to a single class ($\mathbf{C\_X} = classOf(o)$).

The consistency condition that must be satisfied is:

> For each class $\mathbf{C\_X} \in$ C, for each behavior *B_alpha* defined on the type $associatedType(\mathbf{C\_X})$ or one of its supertypes, the cardinality of the set $mostSpecificBindings(B\_alpha, associatedType(\mathbf{C\_X}))$ (defined in Section 4) is 1, and the only element of this set, function *F_alpha*, satisfies the following:
>
> > Either the set $mostSpecificBindings(F\_alpha, associatedImplementationType(\mathbf{C\_X}))$ (defined in Section 4) has cardinality 1, or it has cardinality 0 and the function *F_alpha* has high-level code.

This consistency condition ensures that any behavior applicable to an object can be unambiguously dispatched on the class of that object.

In this section, we have presented a model in which a clean separation between interface, implementation, and representation is achieved. In the next section, we will discuss the principles behind behavior (message) dispatch in the proposed model.

---

[2] An example of a behavior that is *defined* on a type but is not *bound* on it is the behavior *B_drawCheque* defined on the type `T_Account`.

[3] For example, the field `remoteNumber` in the implementation type `IT_PartnerAccount`.

4 Dispatch

There are currently two major dispatch techniques: the traditional, receiver-only dispatch (*single dispatch*) and a newer, more complicated but also more powerful, *multiple dispatch* that chooses the function (method) according to the types of all message arguments. We will consider dispatch in the proposed model using both of these techniques.

4.1 Single dispatch

In the case of single dispatch, the code chosen for execution when a behavior is applied is determined by the class of the receiver.

Theoretically, in the proposed model, the function is chosen according to the behavior being applied and the receiver type, and the implementation function is then chosen according to the function and the receiver's implementation type (two-phase dispatch). However, since both type and implementation type are known by a class, it is possible to implement this theoretical model by a single-phase dispatch. We will discuss this issue further in Section 6.

Since it is possible that there is more than one binding of a particular behavior inherited or defined by a particular type, it is important to know which binding prevails. The same situation occurs for function-to-implementation function bindings. The process used to pick a binding defines the semantics of interface and implementation inheritance respectively.

That is, the behavior-to-function bindings are defined and inherited along the type hierarchy, while function-to-implementation function bindings are defined and inherited along the implementation type hierarchy. The mechanisms used in these two cases are identical, therefore we will only describe the type inheritance mechanism.

```
mostSpecificBindings(behavior,type)
      IF (explicitBinding(behavior,type) ≠ NONE) THEN
            RETURN explicitBinding(behavior,type);
      ELSE
            result = ∅;
            FOREACH supertype IN immediateSupertypesOf(type) DO
                  result := result ∪ mostSpecificBindings(behavior,supertype);
            RETURN result;
```

Figure 4: Algorithm for finding most specific bindings

Let us consider a particular type `T_X` and a particular behavior *B_alpha*. The set of *the most specific bindings* is then determined by the algorithm depicted in the Figure 4. This is the set of bindings that are not overridden in the inheritance hierarchy. If it is empty, the behavior *B_alpha* is inapplicable to the receiver of type `T_X` (message not understood). If the set has a cardinality of 1, the only function in the set is chosen. If the set has a cardinality which is greater than 1, the behavior application is ambiguous.

An example of behavior-to-function binding inheritance is depicted in Figure 5. Each type is represented by a box divided into three parts: the upper part shows the name of the type, the middle part shows explicit bindings for that type, and the lower part depicts the set of the most specific bindings derived by the algorithm. All bindings are given for a single behavior *B_alpha*. Note that the type `T_7` has two most specific bindings and thus cannot be used to define any class since otherwise the consistency constraint would be

violated. However, its subtype `T_8` can be used to define a class as it has a single most specific binding for the behavior *B_alpha*.

Thus, the consistency condition described in the previous section ensures that a legal behavior application can always be dispatched unambiguously. The same algorithm is used when a function is dispatched on the implementation type of its receiver to yield an implementation function.

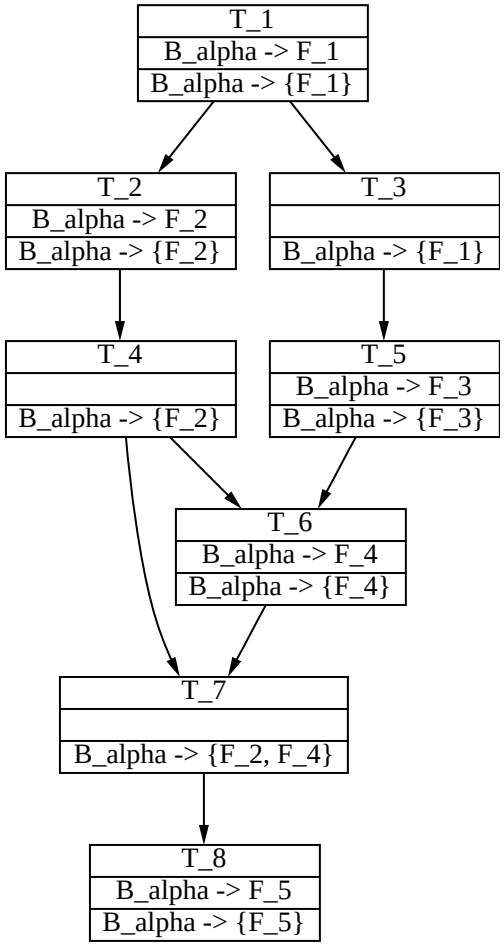

Figure 5: Inheritance example

If a function *F_alpha* is written in a high-level language (e.g. the function attached to the behavior *B_drawCheque* on the type `T_ChequingAccount` in Figure 1), such a function is applicable to objects irrespective of their implementation types. This is treated as an association between the function *F_alpha* and an anonymous implementation function *IF_anonymous* on the implementation type `IT_*`, which is the supertype of all implementation types in the system. Such a treatment allows us to use the dispatch model already presented with no modifications. It also corresponds to the intuition that a high-level function does not care about the implementation types of its receiver and other arguments.

4.2 Multiple dispatch

Multiple dispatch differs from single dispatch in that classes of all arguments of a behavior, rather than just the receiver class, participate in dispatch. Multiple dispatch can solve many typing problems in object-oriented languages and systems, but this increased power comes with a steep price: a significantly more complicated dispatch mechanism and decreased efficiency ([BCC+96], [BG95], [Cas95], [Cha92]).

The choice between single and multiple dispatch is a design decision. In this section, we will show how our model can be adopted for multiple dispatch without making any arguments in favor of either single or multiple dispatch.

In the presence of multiple dispatch, there can be more than one binding of a behavior to a function on a type (a function to an implementation function on an implementation type) for different types (implementation types) of arguments. Such bindings can be understood as bindings on a set of *product types*, where a product type is a tuple of all argument types, including the receiver type. Subtyping between product types is a component-wise subtyping of their components ([BG96], [CTK94]). If all bindings are understood in this manner, the mechanism of the previous subsection can be used to choose an appropriate (most specific) binding.

Thus the only changes to the model required by the introduction of multiple dispatch are the following ones:

1. It should be possible to provide more than one binding for a behavior on a type provided the types of arguments differ. The same is true of functions and implementation types.
2. The dispatch mechanism should be able to handle product types.

In fact, the space of design decisions here is much wider than single-versus-multiple dispatch. Since type and implementation type hierarchies are completely independent, it is possible, for example, to provide multiple dispatch on types while providing only single dispatch on implementation types. The same is true of other design decisions related to the construction of the type hierarchy and inheritance, such as single-versus-multiple subtyping or the methods for conflict resolution (disambiguating ambiguous behavior (function) bindings).

5 Discussion

In this section, we will describe how the model introduced in this paper can help to solve the problems described in Section 1.

We will start with the problem of code reuse. The model proposed in this article cleanly separates interface from implementation and representation. Since only types are used to specify variables in high-level (function) code, such code is reusable without modifications for all implementation types. In other words, the high-level code is totally independent of underlying machine representation. For instance, the code written for the type `T_Account` and its subtypes `T_ChequingAccount` and `T_SavingsAccount` in Example 3.1 works for both ordinary and remote accounts.

The ability to keep high-level code intact when the underlying implementation type hierarchy changes also helps to provide a very high degree of extensibility. For example, it is possible to provide a highly optimized version of complex numbers by introducing a new implementation type and a class that provides optimized implementation functions for all operations on the complex numbers. The application design can thus proceed in two stages: first, a prototyping stage, when all the necessary types are defined and implemented by the default mechanism (which will provide capabilities for fast prototyping, like the one used in Smalltalk [GR89]). After the prototype is tested and its performance is measured, new implementation types for the

types that need optimization are developed and integrated into the system. It is important to note that at this stage the (high-level) code that was developed in the first stage is kept intact, significantly reducing the possibility of introducing new bugs during optimization. This approach can also be used to introduce new "primitive" types into the system, such as multimedia types that might not have been provided as "primitive" by the system kernel. These new primitive types will have exactly the same status as the kernel ones.

The transparent introduction of new implementation types for the types that already exist in the system can also be used to support incremental data evolution. As has been argued in Section 1, coexistence of old and new data has to be supported if we want to continue using old code during and after incremental data evolution. This can be achieved by introducing the "new" implementation type describing the new data structure, while keeping the "old" implementation type in the system. This way, both old and new high-level code will be able to use old as well as new objects since the dispatch mechanism will choose the correct implementation function (old or new) every time a behavior is applied to a particular object. For example, if the banks MegaBank and MiniBank from Example 3.1 merge, the objects representing remote accounts (classes **C_PartnerChequingAccount** and **C_PartnerSavingsAccount**) will be gradually transformed to the objects representing ordinary accounts (classes **C_ChequingAccount** and **C_SavingsAccount**, respectively). Both representations can coexist as long as the implementation of the remote (MiniBank) system access is changed to access the local system.

A similar method can be employed to provide transparent foreign data integration. An example of such integration can be seen in the definitions of remote account implementation types and classes in the MegaBank system example (Section 3). High-level code written for account types does not depend on the fact that some accounts are remote ones and the data related to them have to be obtained via interaction with a remote system. In fact, it is possible that the implementation type `IT_PartnerAccount` as well as the classes **C_PartnerSavingsAccount** and **C_PartnerChequingAccount** are introduced at the time when MegaBank establishes partnership with MiniBank rather than at the time the database is created. The introduction of the above entities does not affect the code written for account types; it is still valid and works the same way as before.

In this section, we have outlined the ways in which the proposed model can help in solving the problems described in Section 1. In the next section we will consider the method of shielding an ordinary user from the internal complexity of the proposed model. We will also show how the proposed model can be implemented in an existing object-oriented database management system.

6 Implementation notes

The implementation type mechanism is designed primarily for low-level tasks. Thus, it is possible to shield an ordinary user from the complexities related to the existence of two independent type systems. The user will only be able to deal with types, classes, behaviors, and functions written in high-level code. A special keyword has to be provided to allow the user to specify a function as "stored". Then, the default system mechanism will create appropriate implementation types as slot lists, where the number of slots is equal to the number of user-defined stored functions for the given class. From the user perspective, the behavior of the system will resemble that of the Smalltalk run-time environment.

On the other hand, a database administrator will be able to use all the capabilities of the user-defined implementation type mechanism. The changes made by the system administrator to the implementation of types will be transparent to the ordinary users, since high-level code (the only code the users are allowed to write) is independent of the underlying implementation type system. The same is true of the type system

the user sees and deals with.

While the ability to cleanly separate interface from implementation and representation at the language level has many advantages, the model presented in this paper can also be implemented in any sufficiently powerful language or database system. The features that are required are:

1. Support for multiple inheritance
2. The ability to support low-level optimized user code

The C++ language [Str91] and the ObjectStore DBMS [LLOW91] satisfy these requirements. The following is a description of design principles that can be used to gain many of the advantages of the proposed model in ObjectStore.

We map types, implementation types, and classes to C++ classes, behaviors to C++ method declarations, implementation functions to C++ method definitions, and functions to protected methods. More precisely, types map to abstract C++ classes with no attributes. Behavior-to-high-level function bindings on types correspond to non-abstract method definitions on these classes. Implementation types correspond to abstract C++ classes that can define attributes and protected methods that correspond to function-to-implementation function bindings. C++ subclassing between C++ classes corresponding to types (implementation types) represents subtyping (implementation subtyping). The C++ classes that represent classes in our model have no attributes and no methods of their own, they just inherit from both the C++ class representing their type and the C++ class representing their implementation type. The code in C++ classes corresponding to types is independent of the implementation type hierarchy with many of the advantages described in this paper.

Having described principles that can be used to implement the proposed model in an existing database system, we now turn to the question of the price we have to pay for the additional functionality provided by our model. First, a price is paid due to the conceptual complexity of the model. However, this complexity can be shielded from an ordinary user, as described above. Second, the proposed model may have a negative impact on the performance of the dispatch mechanism. However, this impact will only be felt in systems that utilize a run-time lookup dispatch strategy. Systems that do not allow run-time schema modifications and use table-based dispatch, such as ObjectStore, BeCecil, and many others, will not suffer from any negative impact[4]. On the other hand, if run-time schema modifications are allowed and run-time lookup is used, the straightforward approach to the implementation of dispatch in the proposed model can decrease performance by a factor of 2 (instead of dispatching on a type, we now have to dispatch on a type and then on the implementation type). The more the original dispatch process is optimized, the less it will suffer from the negative impact of the new model. Therefore we feel that the increased functionality of the model outweighs the potential impact on performance.

7 Conclusions

In this paper we have described several problems related to various aspects of data representation in current object-oriented database systems. Following the analysis of existing systems and languages, we have presented a model that supports a clean separation between interface, implementation, and representation. We have also described the principles of behavior (message) dispatch in the proposed model under different dispatch paradigms (namely, single and multiple dispatch).

[4] The proposed implementation in ObjectStore described above has the same dispatch performance as any other application written in ObjectStore, i.e., the dispatch performance is not affected.

We have also shown how such a model can be used to solve the problems described in Section 1. Finally, we have presented a mechanism that can be used to shield an ordinary user from the internal complexity of the model along with a method to implement the proposed model on top of a sufficiently powerful database management system such as ObjectStore and a discussion of the performance impact of the proposed model.

We believe that the additional functionality provided by the proposed model can have a significant impact on the development of interoperable, extensible, and reusable object-oriented DBMSs. The improved support for code reuse, extensibility, and foreign data integration provided by the model will help in the design and implementation of such systems. At the same time, the part of the model visible to an ordinary user will provide him with a familiar object-oriented framework.

The future research directions include static consistency checking of behavior and function definitions, addition of parametric types (implementation types) to the model, and extension of the machine representation definitions.

References

[ACO85] Antonio Albano, Luca Cardelli, and Renzo Orsini, *Galileo: A strongly-typed, interactive conceptual language*, ACM Transactions on Database Systems **10** (1985), no. 2, 230–260.

[BBB+93] René Bal, Herman Balsters, Rolf A. De By, Alexander Bosschaart, Jan Folkstra, Maurice Van Keulen, Jacek Skowronek, and Bart Termorshuizen, *The TM manual*, December 1993, Version 2.0 revision C.

[BCC+96] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce, *On binary methods*, Theory and Practice of Object Systems **1** (1996), no. 3, 221–242.

[BG95] Elisa Bertino and Giovanna Guerrini, *Objects with multiple most specific classes*, ECOOP'95, 1995.

[BG96] M. F. Barrett and M. E. Giguere, *A note on covariance and contravariance unification*, ACM SIGPLAN Notices **31** (1996), no. 1, 32–35.

[BMO+89] Robert Bretl, David Maier, Allen Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams, *The GemStone data management system*, Object-Oriented Concepts, Databases and Applications (Won Kim and Frederick H. Lochovsky, eds.), Addison-Wesley, 1989.

[BPK95] A. Bouguettaya, M. Papazoglou, and R. King, *On building a hyperdistributed database*, Information Systems **20** (1995), no. 7, 557–577.

[BY95] W. C. Burkett and Y. W. Yang, *The STEP integration information architecture*, Engineering with Computers **11** (1995), no. 3, 136–144.

[Cas95] Giuseppe Castagna, *Covariance and contravariance: Conflict without a cause*, ACM Transactions on Programming Languages and Systems **17** (1995), no. 3, 431–447.

[Cha92] Craig Chambers, *Object-oriented multi-methods in Cecil*, ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands (New York, N.Y.) (Ole Lehrmann Madsen, ed.), Lecture Notes in Computer Science, vol. 615, Springer-Verlag, New York, N.Y., 1992, pp. 33–56.

[CL95] Craig Chambers and Gary T. Leavens, *Typechecking and modules for multimethods*, ACM Transactions on Programming Languages and Systems **17** (1995), no. 6, 805–843.

[CL97] Craig Chambers and Gary T. Leavens, *BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing*, FOOL 4, The Fourth International Workshop on Foundations of Object-Oriented Languages, Paris, France, January 1997.

[CTK94] Weimin Chen, Volker Turau, and Wolfgang Klas, *Efficient dynamic look-up strategy for multi-methods*, ECOOP '94, European Conference on Object-Oriented Programming, Bologna, Italy (New York, N.Y.) (Mario Tokoro and Remo Pareschi, eds.), Lecture Notes in Computer Science, vol. 821, Springer-Verlag, July 1994, pp. 408–431.

[DGLM95] M. Day, R. Gruber, B. Liskov, and A. C. Myers, *Subtypes vs where clauses: Constraining parametric polymorphism*, SIGPLAN Notices **30** (1995), no. 10, 156–168.

[Fis89] D. Fishman, *Overview of the Iris DBMS*, Object-Oriented Concepts, Databases and Applications (Won Kim and Frederick H. Lochovsky, eds.), Addison-Wesley, 1989, pp. 219–250.

[Fra97] Michael Franz, *The programming language Lagoona — A fresh look at object-orientation*, Software — Concepts and Tools **18** (1997), 14–26.

[GR89] A. Goldberg and D. Robson, *ST-80, the language*, Addison-Wesley, 1989.

[Höl93] Urs Hölzle, *Integrating independently-developed components in object-oriented languages*, Proceedings of ECOOP'93, 1993.

[Kon93] Dimitri Konstantas, *Object oriented interoperability*, Proceedings of the Seventh European Conference on Object Oriented Programming (ECOOP'93), July 1993.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, *The ObjectStore database management system*, Communications of the ACM **34** (1991), no. 10, 50–63.

[LMS+93] Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Bennet Vance, Scott L. Vanderberg, and Stanley B. Zdonik, *The AQUA data model and algebra*, Tech. Report CS-93-09, Brown University, 1993.

[LP91] W. R. LaLonde and J. Pugh, *Subclassing $\neq$ subtyping $\neq$ is-a*, Journal of Object-Oriented Programming **3** (1991), no. 5, 57–62.

[LP97] L. Liu and C. Pu, *An adaptive object oriented approach to integration and access of heterogeneous information sources*, Distributed and Parallel Databases **5** (1997), no. 2, 167–205.

[LRV92] C. Lécluse, P. Richard, and F. Vélez, *O_2, an object-oriented data model*, Building an Object-Oriented Database System: The Story of O_2 (François Banchilon, Claude Delobel, and Paris Kanellakis, eds.), 1992.

[RTL+91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul, *Emerald: A general-purpose programming language*, Software Practice and Experience **21** (1991), no. 1, 91–118.

[SOM93] Clemens Szypersky, Stephen Omohundro, and Stephan Murer, *Engineerig a programming language: The type and class system of Sather*, Tech. Report TR-93-064, The International Computer Science Institute, November 1993.

[Str91] B. Stroustrup, *The C++ programming language*, Addison-Wesley, 1991.

[Tai96] Antero Taivalsaari, *On the notion of inheritance*, ACM Computing Surveys **28** (1996), no. 3, 439–479.

[VOD95] *VODAK V4.0 user manual*, April 1995, GMD Technical Report No. 910.

[Weg96] P. Wegner, *Interoperability*, ACM Computing Surveys **28** (1996), no. 1, 285–287.