An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach

Mehmet Akşit, Lodewijk Bergmans & Sinan Vural

University of Twente, Dept. of Computer Science
Enschede, The Netherlands

Abstract. This paper introduces a new model, based on so-called *object-composition filters*, that uniformly integrates database-like features into an object-oriented language. The focus is on providing persistent dynamic data structures, data sharing, transactions, multiple views and associative access, integrated with the object-oriented paradigm. The main contribution is that the database-like features are part of this new object-oriented model, and therefore, are uniformly integrated with object-oriented features such as data abstraction, encapsulation, message passing and inheritance. This approach eliminates the problems associated with existing systems such as lack of reusability and extensibility for database operations, the violation of encapsulation, the need to define specific types such as sets, and the incapability to support multiple views. The model is illustrated through the object-oriented language Sina.

1. Introduction

Traditionally, data-intensive applications have been developed as application programs executing on top of a database management system, and using database services through embedded data manipulation statements. This approach suffers from the need to manage two different languages, and to interface them with extra programming effort. There have been numerous attempts at integrating these two systems within the framework of the object-oriented paradigm [Kim 90]. It is claimed that the object-oriented model provides a more suitable basis both for application programming and data management operations, when it is selected as a common computation model. In addition, since objects can represent complex data structures, object-oriented databases are presumably more capable in dealing with emerging applications such as computer-aided engineering.

A considerable number of object-oriented database management systems have been developed or are currently under development (e.g. [Maier 86], [Kim 89] and [Ontologic 91]). These systems support the basic elements of the object-oriented model, and provide efficient data management, transaction support, and querying facilities. The full integration of language and database systems, however, cannot be considered to be solved completely. The problem is three-fold.

Firstly, since these systems extend an object-oriented computation model with conventional database mechanisms like (non-object-oriented) query languages, the advantages of the object-oriented model do not fully extend to database features. For example, encapsulation and inheritance cannot be used together conveniently with the database-like features in uniform way. Consequently, it is more difficult to obtain modular, reusable and extensible software for the data management part of applications. In addition, the programmer still has to deal with two different systems.

Secondly, introducing database-like features into the object-oriented language model generally introduces weakened encapsulation, and these features are generally provided only for a restricted number of language structures such as sets or classes.

Thirdly, neither languages nor object-oriented database systems address the problem of providing different interfaces on the same object in a general way [Hailpern 90]. This is the so called *multiple views problem* and manifests itself in many software designs. Views have traditionally been supported in database systems, and it should be possible to define them for all language objects within the system.

The model presented in this paper extends the conventional object-oriented model through *object composition filters* which are an integral part of our object model. The database-like features are defined in terms of these filters. As a result, data abstraction, polymorphic message passing and inheritance are fully integrated with them. On the other hand, no compromises are made for object-oriented principles such as encapsulation, and all language objects potentially support database-like behavior. Transactions and multiple views are supported as well.

This paper is organized as follows: The next section gives an overview of the state-of-the art systems. Section 3 summarizes the major problems, which will be taken into account explicitly throughout the paper. The proposed language model is introduced in section 4. Section 4.1 explains the basic object model. Section 4.2 describes how multiple views can be constructed in this model. Section 4.3 extends the model to incorporate inheritance, delegation and associative-access mechanisms. Object management features are explained in section 4.4. Section 4.5 introduces transaction mechanisms and persistency. Finally, section 5 evaluates the computation model and gives conclusions.

2. Background and Related Work

In this section we describe several systems that attempt to integrate database features with an object-oriented language.

2.1 Smalltalk & Smalltalk-based Systems

The *Smalltalk* system [Goldberg 83] offers a limited set of database-like features within its programming environment. Smalltalk provides persistence for all objects, using the *save image* facility which saves a snapshot of the Smalltalk environment as a whole. The *Orwell* system [Thomas 88], which is based on Smalltalk, introduces individual storage for objects, but is mainly intended for version and configuration management.

In Smalltalk, associative access is provided through the method *select:* defined on collections, such as *Set*, *Dictionary* and *Bag*:

```
aCollection select: [:element | ... ]
```

Here, *aCollection* is an instance of a collection class, *select:* is the name of a method defined for collection objects, and the brackets "[...]" indicate a constant argument object of the class *Block*. The class *Block* represents Smalltalk programs. Within this block object, *element* is called the *block argument*. A block serving as an argument to a *select:* message must have a single block argument and a body returning a boolean value. The block body is evaluated for each element of *aCollection*. The result of the method execution is another instance of the collection class, containing elements from *aCollection* for which the argument block has evaluated to *true*.

GemStone ([Maier 86], [Bretl 89]) is an object-oriented database system based on Smalltalk. Its language *OPAL* extends Smalltalk in a number of ways. The "{...}" constructor is introduced as a substitute for "[...]" in order to signal the use of indices for selections on nonsequencable collections. A second extension is the usage of path expressions to represent joins in the relational sense. The path expressions are also used to define indexes. A path expression is a sequence of instance variable names separated by periods, e.g. *student.dept.location*. Sequences of messages, e.g. *student dept location*, could be used for the same purpose as well, but path expressions bypass the execution layer, and allow query optimizations at the database level. User sessions are considered to be transactions. A shadow paging mechanism is employed to ensure database consistency.

2.2 ORION

ORION ([Kim 88], [Kim 89]) is an object-oriented database system based on an object-oriented version of Common Lisp. Persistent storage is provided for all objects, and a transaction subsystem is in charge of database consistency. *ORION*'s Common Lisp defines a method *select* on classes, instead of on collections:

```
(select aClass QueryExpression)
```

Here, *aClass* denotes the class which is the receiver of the message, and *QueryExpression* is a boolean expression expressed in Lisp which is the argument of the message *select*. The result is returned as a set object containing the qualified instances of the class. Paths of instance variables (called complex attributes) may be used in query expressions, e.g. *Dept Location*. Transaction control is supported by functions *commit* and *abort*.

2.3 Ontos

Ontos ([Ontologic 90], [Ontologic 91]) extends C++ with a class library that includes a persistent root class *Object*. Objects of a class are persistent if the class is a direct or indirect subclass of *Object*. Objects must be saved by explicit *put* messages even though they are persistent through their class. There are several additional requirements for a persistent object, which force the programmer to write a considerable amount of code only to make a C++ object persistent.

For associative access, an SQL-like query facility is introduced. Queries may be directed both to classes (indicating a table of all the instance of the class) and to aggregates like sets, lists, dictionaries and arrays (the *from* clause). As with GemStone and ORION, instance variables may be cascaded to form path expressions that simulate relational joins. In order to execute a query, an instance of the *QueryIterator* class has to be created, supplying the text for the query as an argument. The rows that qualify according to the *select* clause may be returned by successive *yieldRow* messages to the *QueryIterator* instance. Transactions are supported by global functions to start, commit and abort a transaction.

3. Our View of the Problem

We may sub-divide the language-database integration problem into *duality in conception*, *restriction in associativity*, *violation of encapsulation*, *fixed views*, and *lack of object-oriented support* in database features. These problems will be explained in turn below:

3.1 Duality in Conception

There is a clear difference between "integrating" and "interfacing" programming languages and database systems. From the above accounts of object-oriented language-database systems, it is evident that language and database models are still kept separate, but the programmer is offered possibilities within the language to access database facilities that are in fact not part of the language model. This results in a set of constructs separated from the language, rather than embedded within it. Moreover, the programmer is frequently confronted with the fact that he/she is actually dealing with two systems instead of one. For instance, the usage of a separate block constructor in GemStone's OPAL for queries to be optimized by database indexes conflicts even with the essential data independence claim of database systems. Similarly, the necessity of explicit *object lookups* and *puts*, *object-type links*, and the SQL interface in Ontos, force the programmer to deal with two distinct systems.

3.2 Restriction in Associativity

For almost all systems, associative access is restricted to a fixed number of classes, and thus objects to be accessed associatively have to be inserted into one of such structures explicitly. For example, the selection capability in Smalltalk and Gemstone is restricted to instances of collection classes. The problems with Orion's approach are that associative access is defined on classes and produces sets, and the resulting sets cannot be further

restricted. In Ontos, queries can only be directed to classes and aggregates. Similar to Orion, return values are restricted to a few types. A query may return rows that are not objects.

3.3 Violation of Encapsulation

In Gemstone, Orion and Ontos, attempts to formulate object queries have resulted in path expressions which make object structures visible and thus are against the encapsulation principle of the object-oriented model: encapsulated data should be accessed via message sends only. Since Smalltalk does not introduce path expressions and is a pure object-oriented language, its query mechanism using the *select:* method does not violate encapsulation.

3.4 Fixed Views

Relational databases invariably support views on base tables, which allow users to work only on the parts of the database that are relevant to them. It is also possible to create virtual tables through the view mechanism by joining several tables under a view. The multiple views problem in object-oriented designs has been addressed by several authors (e.g. [Pernici 90], [Hailpern 90]). Not all methods of an object are of interest to (all) other objects that use its services. Therefore, it is desirable to define views on an object, differentiating between clients, for better information hiding and improved structuring of object relationships.

In languages such as C++ [Ellis 90], Trellis/Owl [Schaffert 86] and PAL [Björnerstedt 88], multiple views can be defined by the programmer with respect to the different clients of an object. These mechanisms in general only distinguish between the following categories of clients; the object itself, the descendants of an object, and other client objects. However, they do not allow any distinction between different kinds of external client objects. In the Smalltalk programming environment, the concept of *private methods* is introduced, but it is not enforced by the language. Gemstone and ORION do not provide multiple views at all. Multiple views in Ontos are based only on C++, thus its view mechanism is very limited.

3.5 Lack of Support of Object-Oriented Features

Since data management features of most systems can be considered as add-on extras, object-oriented properties can not be used optimally for all system components. For instance, all discussed systems except for Smalltalk support transactions. However, they introduce transactions separately from object-oriented features like data abstraction, message passing and inheritance. Therefore it is in general not possible to construct extensible software with transaction characteristics. Moreover, this applies for all database-like features. For example, it should be possible to combine associative access with any object-oriented feature such as inheritance. This would result in *associative inheritance*, which is useful in case of complex inheritance hierarchies. Associative inheritance will be discussed in more detail in section 4.3.

4. The Language Model

We believe that an object model that provides abstract operations for its users and encapsulates its implementation details is a good starting point for building complex systems. It is commonly accepted that polymorphic message passing between objects, and sharing mechanisms such as inheritance or delegation are important techniques in building reusable and extensible systems [Wegner 90]. However we feel that committing to a single abstract class inheritance model is far too restricted. In particular, this object model is found to be too simple to deal with the problems related to language-database integration.

We are strongly convinced that the starting point for language-database integration lies in casting database principles onto the data abstraction model of the language, and making them inherent throughout. Otherwise, we end up with language counterparts of database structures and facilities, i.e. dedicated classes and methods, requiring extra overhead for the programmer, and not mingling properly with other elements of the model, such as inheritance. This was identified in Section 3. If database-like features are to be integrated into an object-oriented programming language, then they should be available for all objects without any restrictions or implications. Therefore, we have enhanced the basic object model to incorporate associativity and multiple views. The vehicle for providing these mechanisms is provided by the so-called *composition filters*, which are explained in this section.

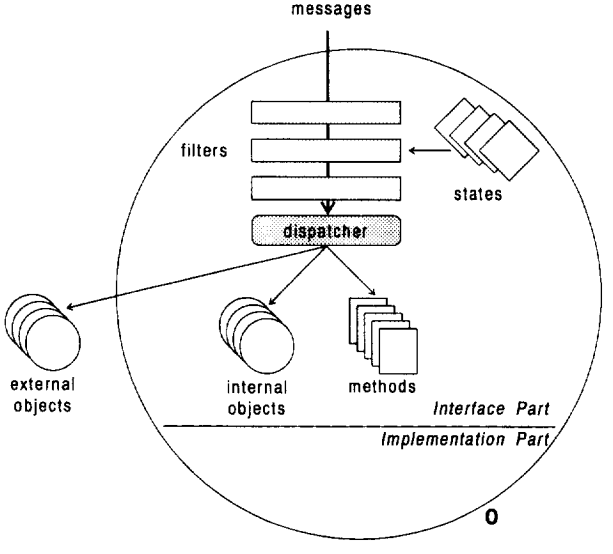

Fig. 1. Extension of the object-model with composition-filters.

As shown in Figure 1, in its input part, object O defines its set of own methods, interface objects, and states[1]. Interface objects are sub-divided as internal and external objects. In addition to that, a set of composition filters are defined and organized in a certain way. Message invocations for this object are first evaluated by these filters and then dispatched to an appropriate method. States are used to control filters. The selected method can be one of the elements of the method set, or a method of one of the internal or external objects.

This mechanism provides a higher degree of flexibility than the conventional fixed set of methods at the interface of an object. The crucial property of this model is that it can support basic object-oriented constructs such as inheritance and delegation, as well as database-like features such as dynamic data structures, transactions, multiple views and associative access exclusively via filters. The only additional operations needed are some basic object methods, for instance *copy*, inherited from the root class in the hierarchy, named *Object*. In the following sections, we will describe this new model adopted by the Sina language starting from simple objects to more sophisticated structures[2].

4.1. The Basic Object Model

In Sina, every object o is an instance of a class $c \in C$. An object $o \in O$ is modeled as a quadruple, $\langle I, M, S, F \rangle$, where

C is the set of all Sina classes.
O is the set of all Sina objects.
I is the set of interface objects of o; these are objects that are within the scope of the object, although not necessarily encapsulated by the object.
M is the set of methods defined within class c.
S is the set of states defined within class c.
F is the (ordered) set of filters defined within class c. (1)

As shown in Figure 2, a class definition is divided into separate parts: the *input part* and the *implementation part*. The input part contains the declaration of the *interface objects I*, divided into two components. The first component consists of encapsulated interface objects called *internals*. The second component consists of interface objects that are outside, but within the scope, of the object. These are called *externals*. The input part also declares the class-specific methods M, states S, and the filters F. Method declarations in

---

1) The term input part implies the existence of both input and output parts. Indeed, an output part can be defined to control the messages that are sent outside of the object. However, in this paper, we are only concerned with the input part of an object. Therefore, for simplicity, instead of using the term input methods and input interface objects, we will refer to them as methods and interface objects. The output part is concerned with implementing the so-called *abstract communication types* (ACTs). ACTs can abstract patterns of communication and large scale synchronization among objects [Aksit 89]. We are currently experimenting with these mechanisms.

2) The early version of the Sina language was published in [Aksit 88] and [Aksit 91]. These publications only illustrated the basic data abstraction model, and did not cover the database-like features that are presented here.

the input part only give names, argument types and return types of methods that are available to users of the object.

The implementation part contains the declarations of the implementation objects, or *instance variables*, and the implementation of the class's *methods* and *states*. It also includes an *initialization* method which is executed immediately after the creation of an instance of a class. If we do not consider filters and object states, this model is somewhat similar to the C++ object model with public and private methods and objects [Ellis 90].

```
class c input
   externals
    //  external objects that are referred to are declared
        here.
   internals
    //  the internal, encapsulated, objects are declared here
   methods
    //  locally defined methods are declared here
   states
    //  local states are declared here
   filters
    //  filters are declared here
end;

class c implementation
   insvars
    //  declaration of instance variables
   states
    //  states are implemented here
   initial
    //  initialization method is defined here
   methods
    //  implementation of methods is defined here
end;
```

Fig. 2. Class template in Sina.

The interface objects are declared as follows:

```
doc: Document;
```

Here *doc* is an interface object, which is declared as an instance of class *Document*.

A state *s* is a certain condition that describes the object at a given time.

$$(s \in S) = \langle \text{proposition, id} \rangle \qquad (2)$$

A *state* may be viewed as a side-effect free boolean function, *proposition*, which can be referred to in filters via an identifier *id*, and which maps the state of the object at a certain moment to *true* or *false*. For example, in the following state implementation, the state *user_view* becomes *true* if the sender of the current message to this object is a subtype of class *User*[3]:

```
user_view  return sender.subtypeOf(User);
```

This condition is expressed as < *sender.subtypeOf(User)* , *user_view* >.

3) In Sina, subtype relations are deduced based on the signatures of objects.

State implementations can be specified in two ways. If the implementation is fixed, it can be defined in the *states* clause of the implementation part. In this case, the state description cannot be changed. If the state function may vary during the lifetime of the object, another instance of class *State* can be assigned to it. This can be done during object initialization, or within a method.

States are declared in the input part since we intend to make them available to users of the object, but their implementation is encapsulated in the object's implementation part. An important property of the state implementation is that it is *side effect-free*. The utilization of states will be illustrated in connection with filters.

The set of methods M_o of object o contains all the methods that are *defined* for the object. But an object may provide other methods on its interface, through the filter mechanism. The largest possible set U_o of methods that are available, is the union of all the methods provided by the interface objects. This rule applies recursively for the interface objects, resulting in the following rule:

$$U_o = M_o \cup (\bigcup_{i \in I} U_i)$$

Which methods eventually become available for the clients of the object is determined by the filters, as will be explained later.

A filter $f \in F$ defines the compositional object behavior and may be defined as a pair:

$$A = \{ \langle s,m \rangle \mid s \in (S \cup (\bigcup_{i \in I} S_i)) \wedge m \in U) \}$$
$$A(f) = [\![\langle s,m \rangle \mid \langle s,m \rangle \in A]\!]$$
$$f = \langle \text{handler}(f), A(f) \rangle \qquad (3)$$

So a filter *f* consists of two components: the first, *handler(f)* is a so-called *filter-handler*, which is an instance of a filter-handler class. A filter handler determines what is to be done with messages after they have passed the a filter (respectively failed to do so). The second component, *A(f)*, is defined as an ordered subset of *A*, which is denoted by the brackets "[" and "]", and is called an *accept set function*. An accept set function defines the conditions (expressed by states) which determine the acceptance of messages. *A* is the set of all possible state-method combinations <*s,m*> within the object. The ordering of the state-method pairs in *A(f)* corresponds with the definition-order. S_i denotes the states that are defined by interface object *i*.

Filters define the guidelines for the object's behavior in terms of methods and states defined by the object and/or those available through its interface objects. A sample filter *f1* is shown below:

```
f1 : Error = { self.user_view=>self.attach, ... }
```

This filter has a filter handler which is an instance of class *Error*. The dot notation is used to bind the state and method names to objects. *s=>m* is the syntactic counterpart of <*s,m*>. It indicates that method *m* is accepted only when state *s* is *true*. In the above filter description, the state *user_view* and the method *attach* that are bound to the object owning this filter (*self*) are used. The pseudovariable *self* might have been omitted here because

whenever a qualifying object name is absent, *self* is substituted[4]. Examples where states and methods of objects other than *self* are be combined in filters will be given in section 4.3.

A filter controls the interface of an object, by filtering incoming messages. The character "," that is used above is called a *selector* and is one of the filter operators. Elements of the filter that are separated by selectors, are processed in left-to-right order.

The class *Error* defines handlers that reject a message whenever it fails to pass through the filter. Similarly, a handler class *Buffer* blocks the message until the object's state allows it to proceed[5]. New handler classes may be defined for any general purpose handling procedure. The admittance of an incoming message is determined according to the state-message pairs. In the above example, an *attach* message is admitted by the filter only if the *user_view* state evaluates to *true*.

Message invocation is a triple $\langle o, m, P(m) \rangle$, where o is the object to which the message is sent, m is the name of the method that is invoked and $P(m)$ is a possibly empty set of arguments (parameters) required by m. Invoking the interface methods of an object is the only means by which another object can communicate with, and change/access the internal state of that object. Invocations are based on messages using the *request-reply* model of communication. An invoked method can return the result (any object) to the sender using the *return* statement. The *nil* object is returned when a method does not explicitly return an object.

An object can communicate with another object by using that object's name which is subject to scope rules. An object can access itself by using the pseudo-variable *self*. An example for a message invocation is the expression

```
mailer.attach(aLetter);
```

This results in sending a request message to the object *mailer*, which is the receiver object, *attach* is the method to be invoked, and *aLetter* is the message argument.

The Sina compiler incorporates a preprocessor to allow programmers to use a more familiar short-hand notation such as the assignment, arithmetic and logical operations. For example, assigning object *a* to *b* may be denoted by *b.assign(a)*, but also by *b:=a*. In the latter case, the preprocessor converts the expression to the standard form *b.assign(a)*.

---

4) Other pseudo variables are *inner*, *sender* and *server*. *inner* is used to designate the locally defined part of an object, which only supports the methods that are implemented by the object itself, whereas *self* refers to the entire object, thereby also supporting the inherited and delegated methods. *sender* is defined in the next section under the topic multiple views. *server* is defined in section 4.3 for constructing *delegation-based* hierarchies.

5) We use the handler class *Buffer* to implement (extensible) concurrent structures; this topic is presented in another paper [Bergmans 92].

The algorithm in figure 3 shows that each received message must be checked by all filters in filter set *F* (line 2), and for every filter again by all filter elements. A filter element is shown as the pair $<s_j,m_j>$ in line 4. A filter *f* only accepts a message *m*, when the message selector matches, i.e. $m=m_j$, and the corresponding state s_j evaluates to true (line 5). When this is the case, no further filter elements of the current filter need to be checked, which is realized by the *break* in line 9. The destination, or *target* of the message is deduced in line 8 from the filter element. In line 12-14, the filter-handler determines what to do with an accepted or rejected message. After the last filter has been passed, the message is dispatched to the desired method, matching the message *m* and destination *dest* (note that *self* or *inner* are also possible destination objects). When a message is rejected, the filter handler may terminate the algorithm, in which case the message is not dispatched (for instance filter handlers which are instances of class *Error*).

Notice that the message is accepted by a filter when it matches any filter element. Thus the selector operator "," can be seen as a logical OR between different filter elements. Only when a message is accepted by all filters it will be dispatched. Hence the subsequent passing through the filters is similar to a logical AND.

```
1)      algorithm pass_filters(m, F)
2)         forall f in F = [f1, ..., fn] do
3)            accept := false;
4)            forall <sj,mj> in A(f) = [<s1,m1>, ..., <sk,mk>] do
5)               if (m = mj) ∧ sj then
6)               begin
7)                  accept := true;
8)                  dest := target(mj);
9)                  break;
10)           end;
11)        endfor;
12)        if accept
13)        then handler(f).acceptMessage(m)
14)        else handler(f).rejectMessage(m);
15)     endfor;
16)     dispatch(m, dest);
17)  end pass_filters;
```

Fig. 3. The algorithm that evaluates received messages with respect to filters.

Because instance variables are not allowed to be targets in the filters, their methods never become available on the interface of the object. In fact, this could also be realized by programmer's discipline only, without the need to declare implementation objects and methods separately. The rationale for this is improved readability of class definitions and separation of the input and the implementation parts of an object.

An important property of the model is that the states and the filters can be treated as first-class objects and are within the set of interface objects *I*. For clarity, we have distinguished them from other interface objects. The basic set operations are defined on the set of interface objects for all objects. The first-class properties are useful for defining object management operations as is shown in section 4.4.

In the sections that follow, a number of applications of the data abstraction model are shown.

4.2. Multiple Views

In this section, we will illustrate how filters can be used to implement multiple views upon objects.

A view is a triple $\langle o_c, o_s, V\rangle$, where o_c is a client object that invokes a message $m \in V$ on a server object o_s. V is the set of messages that provided by o_s for o_c. Having a multiple view mechanism means that the server object supports multiple views depending on its state or on characteristics of its client such as class or identity. For example, it may make some methods visible to instances of one class, and others to instances of another class, or it may define methods that may be executed only by clients that are instances of subclasses of its class. The following filter f implements the view $\langle o_c, o_s, V\rangle$ where V is a subset of all available methods on o_s. As in (3), U denotes all the methods defined for the current object as well as all the methods available from all interface objects:

$$
\begin{aligned}
&o_s = \langle I, M, S, F\rangle \\
&V \subseteq U \\
&(f \in F) = \langle \text{handler}(f), A(f)\rangle \\
&A(f) = [\ \langle s,m\rangle \mid m \in V \wedge s = \langle \text{view\_prop}, \text{view\_id}\rangle\ ]
\end{aligned}
\qquad (4)
$$

As before, *handler(f)* denotes the filter handler object. Now suppose that the proposition *view_prop* is defined as "sender=o_c". Then *A(f)* is the set of state-method pairs that allow only sender o_c to execute methods in V on o_s. The pseudo variable *sender* indicates the object that sent the current message. Apart from the identity of the *sender* object, the implementation of a view may use any general proposition related to the sender object, or the state of the receiver object. In the latter case, an object may provide changing views to its clients.

A sample class definition implementing multiple views is provided in Figure 4.

The class *Text_mail* defines four methods; *attach*, *send*, *deliver* and *route*. The method *attach* takes one parameter of class *Letter* which includes the contents of the mail. The method *send* requires the address of the receiver object as a parameter, and transfers the text to the mail system for delivery. The method *deliver* is used by the mail system to physically deliver the mail. It returns a boolean indicating whether the mail was delivered successfully. The method *route* is used by the mail system to transfer the mail to another mail system, when the destination is not directly accessible to it.

The filter handler class is *Error*. In the filter definition, the curly brackets indicate a shorthand notation for expressing "s=>m_1, s=>m_2, ..., s=>m_n" as "s=>{m_1, m_2, ..., m_n}". The wildcard character "*" can be used in filters to indicate any matching method. Note that the name *self* might have been omitted from the filter definition since it is the default, or *inner* might have been used instead

In this example, two views on the class *Text_mail* are defined. Objects of class *User* are only allowed to invoke messages *attach* and *send* while objects of class *Mail_system* or its subclasses are only allowed to send messages *deliver* and *route*. The pseudo variable *sender* is used to check the class of the client object in the implementation of the states *user_view* and *system_view*. Note that only an object that is a subtype of class *User* or class *Mail_system* is allowed to invoke a message!

```
class Text_mail input
   methods
      attach(Letter) returns Nil;
      send(Address) returns Nil;
      deliver(NodeId) returns Boolean;
      route(NodeId) returns Nil;

   states
      user_view;
      system_view;

   filters
      f1 : Error = { user_view=>{self.attach, self.send},
                     system_view=>{self.deliver, self.route}  };
end;

class Text_mail implementation

   states
      user_view
         return sender.subtypeOf(User);
      system_view
         return sender.subtypeOf(Mail_system);
   ...
end;
```

Fig. 4. Interface and part of implementation of class *Text_mail*.

4.3. Inheritance, Delegation and Associativity

As already identified in the problem statement, we find it too restrictive to adopt a single class inheritance mechanism; rather we want to provide mechanisms like multiple inheritance and delegation as well. In addition, we want associativity to be orthogonal to object-oriented features such as inheritance, so that they can be combined. We will first describe how filters can be used to implement different forms of code sharing mechanisms such as inheritance and delegation. Then we will introduce associative behavior, and explain how it can be defined.

The computation model as introduced by formulas (1-3) and algorithm *pass_filters* of Figure 3 allows interface objects to be made available to the users of the encapsulating object by naming them in filters. We will now show how the methods of an encapsulated object can be made available on the interface of the object:

```
class O input
   internal
      q : ClassQ;
   filters
      f1 : Error = { True=>q.* };
end;
```

The definition according to the formal object model is as follows. Object o is defined, which has a single interface object q:

$o = (I, M, S, \Gamma$

$I = \{q\}$

$F = \{f1\}$

$V = M \cup V_q \Rightarrow V = \emptyset \cup V_q \Rightarrow V = V_q$

Interface object q provides the methods m_1 to m_n, and is defined as:

$q = (I_q, M_q, S_q, \Gamma_q)$

$M_q = \{m_1, m_2, ..., m_n\}$

$V_q = M_q$

The filterset F of o contains only filter *f1*, with accept set function *A(f1)*:

$f1 = (handler(f1), A(f1))$

$A(f1) = [\![<s,m> \mid (s \equiv True) \wedge (m \in V_q)]\!]$ (5)

Here *True* is a state that is always valid; this is provided as the default when no state is indicated. Now, suppose some client sends the message "o.m_i" where $m_i \in M_q$. This message will be accepted according to the accept set function *A(f1)*, since the message is in the set $V_q=M_q$ and the corresponding state is also valid. Then the message will be delegated to, and executed by the interface object q. Note that the client object is not aware of the fact that it is actually executing the method of an interface object. Also note that when m_i is dispatched to q, it has to pass through the filters defined by q before it can be executed.

This mechanism is actually a simulation of *inheritance*, since the object o now provides all messages of *ClassQ* on its interface, using the implementation of q, which is an instance of *ClassQ*. This mechanism is also called *delegation-based inheritance*. If we replace the interface object q with an external object g then the filters implement a form of -pure- *delegation*. In this example, object o includes only one interface object and does not introduce its own methods, thus providing methods of q only. If o had defined its own methods or other interface objects, then the first state-method pair matching the incoming message would have been dispatched. *Multiple inheritance* can be implemented by using several interface objects. The left-to-right evaluation order of filter elements together with the values of states would resolve name conflicts, if any.

In order to access their own methods or methods of their interface objects, within an object messages can be sent to pseudo variables *inner*, *self* and *server*. The pseudo variables *inner* and *self* in a message expression always refer to the implementation respectively interface of the instance of the class where they are used. Because this is defined statically, the semantics of a method implementation can be guaranteed not to be changed due to overriding. Performing an invocation on *server*, however, causes the search for the invoked method to start with the original recipient of the message. Since the objects in Sina can

be nested or the messages can be delegated to the external objects, the recipient of the message and the object in which the invocation appears can be different. We call the receiver of the message *server*, because this object can be thought of as performing a service for the object that originally sent the message (the *sender*). *server* is similar to Smalltalk *self*, in the sense that it supports dynamic binding. But *server* in Sina can handle delegated messages, whereas Smalltalk *self* cannot do this [Lieberman 86].

Typically in most object-oriented languages every class inherits -either directly or indirectly- some default behavior from a root class called *Object*[6]. Sina does not introduce inheritance as a language feature, but using a filter construct, inheritance can be implemented. The Sina system contains a primitive class called *Object* which abstracts the default operations of all the classes. Typical example operations used in this paper are *assign*, *equal*, and *copy*. The Sina compiler provides an option to insert an instance of class *Object* called *default* automatically as the first filter element of every filter in a class. This option makes it unnecessary for programmers to define the default operations explicitly for every new class. Since *default* is the first element of a filter, it prevales over other interface objects. Of course, programmers can explicitly turn off this option and create an instance of class *Object* at the interface of a new class. Then, for example, they can eliminate the assign operation of *Object* so that a constant behavior of the class can be assured.

In Figure 5, we give a sample class definition which uses filters to implement inheritance:

```
class Text_mail input
   internals
      doc : Document;
   methods
      attach(Letter) returns Nil;
      send(Address) returns Nil;
      deliver(NodeId) returns Boolean;
      route(NodeId) returns Nil;
   states
      user_view;
      system_view;
   filters
      f1 : Error = { user_view=>{attach, send, doc.*},
                     system_view=>{deliver, route}  };
end;
```

Fig. 5. Interface definition of *Text_mail*, which inherits from *Document*.

Class *Text_mail* in Figure 5 is similar to class *Text_mail* in Figure 4. An interface object named *doc* of class *Document* is now introduced. The class *Document* includes methods such as *update* and *print*, and by including "doc.*" in the filter, instances of the *Text_mail* class in Figure 5 will also support these methods. The rationale for this is that the class *Text_mail* can now be used to edit the mail text directly, instead of using a separate copy of the mail text.

---

6) Some languages such as C++ do not enforce programmers to inherit from a single class. However, even for C++ programmers it is common practice to introduce a base class such as *Object*.

The filter associates all methods of class *Document* with the state *user_view* defined in class *Text_mail*, which means that only objects that are subtypes of class *User* may send these message to objects of class *Text_mail*. Note that the pseudo-variable *self* is here eliminated from the filter specification, since it is provided as the default.

Having introduced inheritance and delegation through composition filters, we now proceed to define the associative access mechanism and its relation with inheritance.

We have seen that in most systems, a collection of objects is accessed by a condition that applies to all contained objects through a predefined *selection* operation. Since we do not want this mechanism to be available only to a restricted set of objects, it naturally follows that associativity is attributed to every single object. For our model, it means that the collection to be accessed is the set of interface objects. Since the object can use or inherit the methods of its interface objects as shown in (5), the ability to restrict the set interface object set leads to the notion of *associative inheritance* or *associative delegation*. The client may affect the inheritance (or delegation) web to some extent, and specify associatively the objects from which it would like the server object to inherit. In short, a dedicated container class which supports associative access through a special method is replaced by the set of interface objects which every object may possess.

Associativity for interface objects is realized as follows: a received message will be dispatched only to interface objects *i* for which the associated state evaluates to true. This state is defined by $< p(i), id_p >$, where the proposition is expressed by a message expression in which *i* is a receiver (since *p(i)* tests the properties of *i*). *p(i)* is evaluated only for proper interface objects *i* that support all the messages that are required for evaluating *p(i)*. These messages are defined by M_p:

$$M_p = \{ m \mid p(a) \textit{ involves } 'a.m' \} \quad (6)$$

Only those interface objects *i* are selected for which the proposition applies (i.e. which implement M_p), for which *p(i)* evaluates to true, and which implement the received message *m*. This is defined in the accept set function *A(f)* as follows:

$$A(f) = [\![<s, i.m> \mid i \in I \wedge s = \langle p(i), id \rangle \wedge m \in U_i \wedge M_p \subseteq U_i]\!] \quad (7)$$

The filter *f* will then include all interface objects which implement the methods that are required by the proposition *p*, and which satisfy *p*. Since availability of interface objects is determined by their responses to certain conditions but not by their names, such a filter implements associative inheritance. The syntactic equivalent of filter *f* of (7) in Sina is as follows, where *p* is the state which implements the proposition, and which is parameterized subsequently by all suitable interface objects. When *p* evaluates to *true* for object *i* and *i* supports the received message *m*, the message will be accepted, and eventually dispatched to *i*. Proposition *p* can be defined by the object itself, but the object may also allow the client to provide this proposition.

```
{ p(#)=>{#.*} }
```

We illustrate this in figure 6. with an example class, *Multimedia_mail*, which provides a different behavior, depending on the type of media that is desired. The latter can be determined by the client by sending the message *select_mail*, providing the proposition

(query condition) as a 'block' argument. Note that in class *Text_mail* in figure 5, the criterion for associative inheritance is solely determined by the *server* object.

```
class Multimedia_mail input
   internals
      text: Text_mail;
      binary: Binary_mail;
      voice: Voice_mail;
   methods
      select_mail(Block) returns Nil;
   states
      mail_state;
   filters
      f1 : Error = { inner.*, mail_state(#)=>#.* };
end;

class Multimedia_mail implementation
   methods
      select_mail(new_prop:Block)
         begin
               mail_state.proposition(new_prop);
         end;
end;
```

Fig. 6. Definition of class *Multimedia_mail* which associatively inherits from various types of mail objects.

The input filter of class *Multimedia_mail* specifies associative inheritance controlled by state *mail_state*. Since this state can be redefined using the method *select_mail*, the class *Multimedia_mail* can associatively inherit from various mail types as required by the user.

The class *Multimedia_mail* declares three interface objects; *text*, *binary* and *voice* of classes *Text_mail*, *Binary_mail*, and *Voice_mail*, respectively. The definition of class *Text_mail* was given in figures 4 and 5. All these classes implement a specific electronic mail object for the type of mail-data they contain. They also provide dedicated methods for their respective data types.

The method *select_mail* is defined on class *Multimedia_mail* to let the user specify the required mail type. A client of the object may provide a new proposition for the state *mail_state*, as the argument of the *select_mail* method. The method *proposition* takes the argument, which must be of class *Block*, and stores it as the (new) proposition of *mail_state*. An example of invoking *select_mail*, using an instance of *Multimedia_mail* called *aMultimedia_mail*, is:

```
aMultimedia_mail.select_mail( [#.subtypeOf(Voice_mail) ] );
```

The proposition is specified as a constant object of class *Block*, which is denoted with the brackets "[...]". The number symbol "#" stands for the argument of the proposition (interface objects will be substituted here). This proposition will evaluate to *true* only when the argument is a subtype of class *Voice_Mail*.

4.4. Associative Object Management

Associative inheritance provides flexibility in configuring the behavior of an object in a well-defined way. However, if client objects need to define and preserve their own views, but still share data, the associative inheritance mechanism will not be adequate since all client objects observe the same server, with the same view. We therefore need to give different object identities to different views of the same object. Besides, in addition to *selection*, the object model should also support data management operations such as *union*, *intersection* and *exclusion*. In this section we will show how this can be realized within the object model.

Our aim is to provide a different view of an object *o*, and retain this view over a number of method invocations. This cannot be realized by a filter construct only, since filters dynamically reconfigure for every received message. So some changes to the interface of an object need to be preserved over a number of message invocations. Since such changes may not be relevant to all client objects, a copy of *o* must be made, say *o'*, of which the interface will be changed to reflect a different view of *o*.

Since the state of the object *o* must be shared between all clients, *o'* must share its state with *o*. This is realized by making a *shallow-copy* instead of a complete copy. Shallow-copy means that a new object *o'* is created, with a different object identity, but which shares all objects nested within *o'* with the corresponding nested objects in *o*.

We first show the result, *o'*, of a *selection* of object *o* with condition *p*. This creates a view of the object with only those interface objects available that are selected according to condition *p* (making use of (6)):

$o = \langle I, M, S, F$

$o' = \langle I', M, S, F' \rangle$

$I' = \{i \in I \mid p(i) \wedge M_p \subseteq U_i\}$ (8)

Because now only a subset I' of the interface objects is available, the filters must be adapted to take only the accessible interface objects into account, which can be expressed as follows:

$(f' \in F') = <\ \text{handler}(f'),\ A'(f')\ >$

$A'(f') = [\![\ \langle s,m \rangle \in A(f') \mid s \in (S \cup (\bigcup_{i \in I'} S_i)) \wedge m \in (M \cup (\bigcup_{i \in I'} U_i))\ ]\!]$ (9)

These lines state that the filters of the new filter set F' are reduced so they only contain filter elements that refer to the states and methods of the selected interface objects.

Since the set operations *intersection* and *exclusion* are a specific kind of selection, they can be expressed in the same way. In that case only an appropriate selection proposition *p* is to be provided. To define intersection between the interface objects of *o* and the interface objects *o''*:

$o'' = \langle I'', M, S, F'' \rangle$

$p(x) = (\ x \in I''\ )$

Excluding all interface objects I'' of *o''* from *o* requires the following proposition:

$p(x) = (\ x \notin I''\ )$

In order to define a *union* of the interface objects from o with those from o'', we use the same approach (resulting in a new object o'):

$$o = \langle I, M, S, F \rangle$$
$$o' = \langle I', M, S, F \rangle$$
$$o'' = \langle I'', M'', S'', F'' \rangle$$
$$I' = I \cup I'' \qquad (10)$$

Notice that o' offers an alternative view of object o, and therefore only the methods and states that are defined for o are available for o', but not those the states and methods from o''. This is also the case for the filters: the constraints that are imposed by the filter of o, must still be valid for o'. To enforce this, object o' has the same filter set that o has.

As we mentioned before, the set of interface objects I is a first-class set object. Basic set operations like *union*, *intersect*, *exclude* and *select* are provided by set objects. By manipulating the set of interface objects using these operations, views that are combinations or restrictions of interface objects can be programmed. We show this in the following example:

In the example class *Multimedia_mail* of Figure 6 a method *select_mail* is provided that changes the type of mail-data handled by the mail system. One invocation of this method will cause the change to affect all client objects of the mail system. In order to provide a different view of the mail system, which does not affect all the clients, the method *select_mail* can be defined as follows:

```
select_mail(new_prop:Block)
   begin
      return (self.get_input_objects).select(new_prop);
         // get & select the set of (input-) interface objects
   end;
```

Fig. 7. Implementation of method *select_mail* which returns a new view of the receiver object.

For the implementation of method *select_mail* the method *get_input_objects* is used, which returns the set of interface objects. The method *get_input_objects* is inherited from class *Object*. Then a *select* is performed upon this set. The method *select* returns a shallow-copy which contains references to a selection of interface objects. This selection includes only those interface objects that satisfy the condition *new_prop*, which is provided by the client object as an argument of the method *select_mail*.

A possible effect of the method *select* is depicted by Figure 8. Here, the method *select* is invoked with the condition *subtypeOf(Voice_mail)* which results in a shallow-copy of the multi-media mail object o. The view object has a different object identity and shares the contents of the voice mail, possibly with other views. Note that this sharing mechanism is encapsulated and thus not visible to the users of view objects.

Apart from the method *select*, also the methods *union*, *intersect* and *exclude* can be invoked on the set of interface objects, as returned by *get_input_objects*. In formulas 9-10 it was shown how set operations on interface objects affect the behavior of objects. Thus

the programmer has the possibility to implement data management operations upon interface objects.

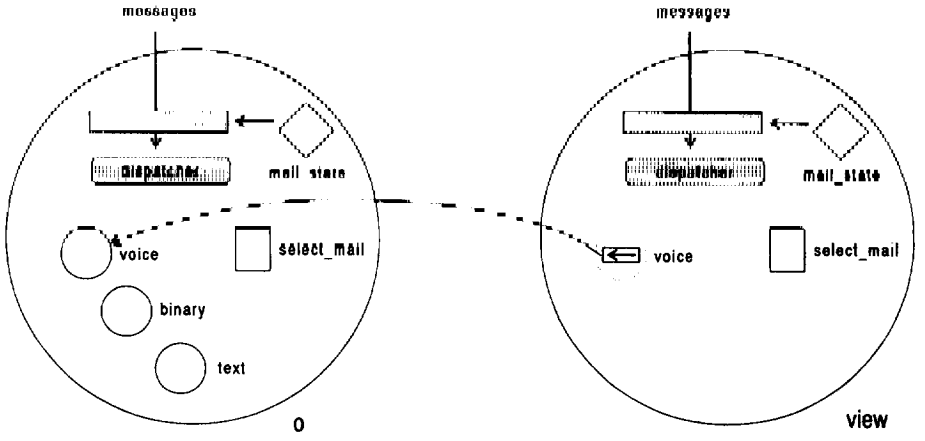

Fig. 8. A possible result of the method *select*.

4.5. Atomic Transactions & Persistence

Most databases support transactions. According to [Haerder 83], a transaction mechanism must provide these four properties: *atomicity*, *consistency*, *isolation* and *durability*. These properties ensure that a transaction always yields a consistent and stable state, even in the presence of system and program failure and concurrent access to shared data. Atomicity, consistency and isolation are provided by the mechanism of *atomic delegation* [Akşit 91]. Durability is separated from transactions, and provided as *object persistence*.

Transactions provided by databases are typically defined in some query language, for a sequence of database operations. Only a few languages, such as *Argus* [Liskov 87] and *Avance* [Björnerstedt 88] support transactions, which are called atomic actions, as a general mechanism in the language for preserving consistency of concurrently accessed resources.

Most object-oriented systems provide transactions for a program block by delimiting it with 'begin-transaction' and 'end-transaction' like constructs, or by making the complete method body atomic. This mechanism does not provide integration with object-oriented constructs such as inheritance. This is because combining inherited methods within an atomic construct requires -in the extreme case- the separate declaration of all atomic method combinations, which is not feasible.

Atomic delegation combines the concepts of delegation and atomic action in a uniform model which supports open-endedness of atomic actions. Atomic delegation allows an object to delegate a sequence of messages to one or more designated objects as a single atomic action; such atomic actions are indivisible and recoverable. This mechanism allows the programmer to define classes of atomic actions rather than defining each atomic action separately. Construction of open-ended systems is supported because new atomic actions may be added or existing ones may be modified by changing the delegation relationships between objects without requiring any redefinition of atomic actions, or recompilation of the objects performing the atomic actions.

We will now show an example of atomic delegation. In this example we add accounting facilities to the execution of every method of our *Multimedia_mail* class. Since, for instance, we do not want to charge when a call fails, and a caller with an exceeded budget limit is not allowed to use the mail facilities, we want to make this an atomic transaction.

```
class Multimedia_mail input
   externals
      acc : Accounting
   ...
   filters
      transact : Error = { True => <acc.*, inner.*> }
   ...
```

The filter *transact* defines an atomic action "*<acc.*, inner.*>*", which is indivisible and recoverable; either both messages are executed successfully and commit, or an abort and subsequent roll-back take place. The brackets "<" and ">" enclose a sequence of messages that form one transaction. The asterisk indicates that all methods that are provided by the target are supported. Note that extensions to object *acc* will automatically be available for clients of the *Multimedia_mail* objects, due to the use of the asterisk. The state *True* indicates that no additional constraints are imposed by this filter in order to execute the atomic action. It may be clear that the number of possible method combinations can be quite large, and it would be infeasible to declare all possible transactions separately, as conventional mechanisms would require.

Persistence of an object is the responsibility of the object itself, and must be transparent to its clients. We feel that conceptually, persistence is simply a property of an object, which has the effect that the object will survive user sessions. We consider the *efficient* implementation of a large amount of persistent objects as a complex, but separate research topic. For our object model, we are not concerned with these implementation issues[7].

The property of persistence of an object can be easily modeled with an attribute 'persistent', which can be affected by message invocations. However, a declarative way of stating the persistence property of an object is preferable, since it is more explicit, and allows for compile-time optimizations. This is realized in Sina by declaring an object as

---

7) In our current prototype, we use the object-oriented database system Ontos [Ontologic 91] for implementing persistent objects.

an instance of class *Persistent*, parameterized with the desired class of the object, as follows:

```
objects doc : Persistent(Document);
```

Here *doc* is an interface object and is declared as an instance of class *Persistent*, parameterized with class *Document*. This declaration results in *doc* being an object with an interface just like all other instances of the *Document* class[8], but the object will also be saved on stable storage. The class *Temporary* is defined analogously, and keeps the internal state only during execution time; *Temporary* is the default for plainly declared objects. Note that this can only be done for internal objects, since these are defined locally, but external objects are defined elsewhere, and are only referred to by this object.

5. Evaluation and Conclusions

Our starting point is an object model that provides abstract operations for its users and encapsulates its implementation details. This model is extended with the composition filters. This paper illustrates the following useful features of this model:

- Multiple views on objects, in section 4.2.
- Basic object-oriented mechanisms such as single and multiple inheritance/delegation, in section 4.3.
- Associative inheritance/delegation, in section 4.3.
- Database features such as sharing, and selection, union, intersection and exclusion, in section 4.4.
- Persistent objects and transactions, in section 4.5.

We will now evaluate our object-oriented model with respect to the problems that were identified in section 3:

- *Duality in conception*: In our model, all the database-like features are provided exclusively via composition filters and no separate query language is introduced. The basic object-oriented mechanisms such as inheritance and delegation are also provided via filters. As a consequence, there is no conceptual difference between the language and database-like features.
- *Restriction in associativity*: In our approach, associative access is available for all objects. Filters can be configured using an expression of the form *{ s(#)=>{#.*} }*. In addition, interface objects are stored in a first-class set object, supporting basic set operations like *union*, *intersect*, *exclude*, and *select*. By manipulating the interface objects with these operations, views that are combinations or restrictions of interface objects can be programmed. Thus our data management functionality is not restricted to dedicated types. However, this does not imply that there should never be dedicated container classes in a system. When an application explicitly deals with objects containing collections of objects, a container class may be created. Such a container

8) Class *Persistent* is implemented as a class that inherits from the class that is supplied as an argument to class *Persistent*; it is possible to express this with the Sina data abstraction model.

class may be similar to container classes in other systems. Our point is that we do not restrict data management operations to this kind of dedicated classes.

- *Violation of encapsulation*: The database-like features as presented in this paper do not violate encapsulation. Nested objects cannot be directly addressed from outside the object. They can solely be accessed by message invocation, but only when this is explicitly allowed by the filters.
- *Views*: Views are provided by the filters, and the view conditions are not restricted.
- *Support of object-oriented features*: We have integrated the database-like properties within our object-oriented model, but they are orthogonal, and can be freely mixed with the data abstraction features, resulting in, for example, associative inheritance or associative atomic delegations.

Various versions of the Sina language have been implemented. The early version of the Sina language was implemented using the Smalltalk language [Goldberg 83] on a Sun workstation. This implementation included only single filters without states. We are currently implementing the new version of the language, translating to C++ [Ellis 90].

References

[Akşit 88] M. Akşit & A. Tripathi, *Data Abstraction Mechanisms in Sina/ST*, OOPSLA '88, pp. 265-275

[Akşit 89] M. Akşit, *Abstract Communication Types*, On the Design of the Object-Oriented Language Sina, Ph.D. Dissertation, Chapter 4, Department of Computer Science, University of Twente, The Netherlands, 1989

[Akşit 91] M. Akşit, J.W. Dijkstra & A. Tripathi, *Atomic delegation: Object-Oriented Transactions*, IEEE Software, Vol. 8, No. 2, March 1991

[Bergmans 92] L. Bergmans & M. Akşit, *An Object-Oriented Model for Extensible Concurrency*, Working paper.

[Björnerstedt 88] A. Björnerstedt & S. Britts, *AVANCE: An Object Management System*, OOPSLA '88, pp. 206-221

[Bretl 89] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams & M. Williams, *The GemStone Data Management System*, Object-Oriented Concepts, Databases, and Applications, Ch. 11, eds. W. Kim and F. H. Lochovsky, pp. 283-309, Addison-Wesley, 1989

[Ellis 90] M.A. Ellis & B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

[Goldberg 83] A. Goldberg & D. Robson, *Smalltalk-80: The Language and its Implemenation*, Addison-Wesley, 1983

[Haerder 83] T. Haerder & A. Reuter, *Principles of Transaction-Oriented Database Recovery*, ACM Computing Surveys, Vol. 15, No. 4, December 1983, pp. 287-317

[Hailpern 90] B. Hailpern & H. Ossher, *Extending Objects to Support Multiple Interfaces and Access Control*, IEEE Transactions on Software Engineering, Vol. 16, No. 11, pp. 1247-1257, November 1990.

[Kim 88] W. Kim, N. Ballou, H.T. Chou, J.F. Garza, D. Woelk & J.Banerjee, *Integrating an Object-Oriented Programming System with a Database System*, OOPSLA '88, pp. 142-152

[Kim 89] W. Kim, N. Ballou, H.T. Chou, J.F. Garza & D. Woelk, *Features of the ORION Object-Oriented Database System*, Object-Oriented Concepts, Databases, and Applications, Ch. 11, eds. W. Kim and F. H. Lochovsky, pp. 251-282, Addison-Wesley, 1989

[Kim 90] W. Kim, *Object-Oriented Databases: Definition and Research Directions*, IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 3, pp. 327-341, September 1990

[Lieberman 86] H. Lieberman, *Using Prototypical Objects to Implement Shared Behavior*, OOPSLA '86, pp. 214-223

[Liskov 87] B. Liskov et. al., *Argus Reference Manual*, MIT Lab. for Computer Science, No. MIT-TR-400, November 1987

[Maier 86] D. Maier, J. Stein, A. Otis & A. Purdy, *Development of an Object-Oriented DBMS*, OOPSLA '86, pp. 472-482.

[Ontologic 90] *Ontos Object Database version 2.0 SQL User's Guide*, Ontologic Inc., Burlington (Mass.), December 1990.

[Ontologic 91] *Ontos Object Database version 2.0 Developer's Guide*, Ontologic Inc., Burlington (Mass.), February 1991.

[Pernici 90] B. Pernici, *Objects with Roles*, Proc. of the Conference on Office Information Systems, pp. 205-215, Cambridge (Mass.), April 1990.

[Schaffert 86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian & C. Wilpolt, *An Introduction to Trellis/Owl*, OOPSLA '86, pp. 9-16

[Thomas 88] D. Thomas & K. Johnson, *Orwell-A Configuration Management System for Team Programming*, OOPSLA '88, pp. 135-141

[Wegner 90] P. Wegner, *Concepts and Paradigms of Object-Oriented Programming*, OOPS Messenger, No. 1, Vol. 1, August 1990, pp. 7-87