Data Abstraction Mechanisms in Sina/st

Mehmet Aksit
University of Twente
Department of Computer Science
Enschede, the Netherlands

Anand Tripathi
University of Minnesota
Department of Computer Science
Minneapolis, MN55455

Abstract

This paper describes a new data abstraction mechanism in an object-oriented model of computing. The data abstraction mechanism described here has been devised in the context of the design of Sina/st language. In Sina/st no language constructs have been adopted for specifying inheritance or delegation, but rather, we introduce simpler mechanisms that can support a wide range of code sharing strategies without selecting one among them as a language feature. Sina/st also provides a stronger data encapsulation than most of the existing object-oriented languages. This language has been implemented on the SUN 3 workstation using Smalltalk.

1. Introduction

The notion of data abstraction is fundamental to object-oriented languages. Data abstraction provides encapsulation so that the internal structure and the implementation of an object are hidden from its users. The internal state of an object is only accessible through its abstract operations. Encapsulation is useful because it hides the unnecessary detail, and since the external interface of an object is independent of its actual realization, the reimplementation of an object will have no effect on the other objects in the system. This paper presents a new data abstraction technique in an object-oriented model of computing. The data abstraction mechanism described here has been developed in the context of the design of the Sina/st language (*).

Most of the existing object-oriented languages aim at a high reusability by introducing *class* and *class inheritance* concepts as a means of code sharing. A class is a template from which instance objects, that all implement the same kind of component, may be created by "new" operations. Inheritance is a structural organization of classes, whereby a class may inherit operations from ancestor classes, or may have its operations inherited by descendant classes. This structural relation provides organization of components so that they can be systematically reused. A useful extension to a single inheritance mechanism is to allow a class to inherit from two or more classes. The introduction of multiple inheritance obviously improves the reusability, because the programmer does not have to rewrite the operations that have been already implemented by the parent classes. Another important motivation for multiple inheritance is that if the code of one of the parents is modified, then all the code of descendant classes will be updated automatically.

Although encapsulation and data abstraction principles have been widely accepted as a basis for object-oriented programming, there has been a great deal of controversy on the principles of inheritance. Some have proposed a code sharing technique called delegation as an alternative to inheritance [Lieberman86]. Delegation is a mechanism that allows objects to delegate the requests of its users to one or more designated objects. A designated object is called "prototypical", if it is both an instance and a template. Delegation is orthogonal to the class concept, and therefore is adopted by classless languages [Agha86]. The designers of these languages claim that delegation is more powerful than inheritance because it can support dynamic evolution of systems, whereas a class definition restricts the behavior of its instance objects at the class definition time. In delegation, the designated object is a part of an extended identity of the delegating object, and this property can not be simulated by inheritance constructs. On the other hand, Stein [Stein87] argues that delegation is a special case of inheritance, because objects in delegation can be modelled as classes in inheritance. Contrary, Wegner [Wegner87] defines delegation as a generalization of inheritance. Moreover, many forms of inheritance have been proposed [Wegner87] [Nguyen86] [Carnese84].

Inheritance and delegation are not the only data abstraction techniques adopted in object-based languages. *Relations*, for example, are proposed to capture and enforce integrity constraints between pair of objects [Kim87] [Rumbaugh87]. Maes [Maes87] has introduced a technique to abstract the reflective part of a system into a module. Francez and others [Francez86] have defined a language construct called *scripts* to abstract pattern of messages.

The general idea behind the data abstraction model of Sina/st is that, starting from a simple object-based model, one can simulate various forms of abstractions without committing to a fixed number of alternative abstraction techniques such as delegation, relations or inheritance. Generality is consciously selected as a design criterion for data abstractions. Therefore, in Sina/st, no language constructs have been adopted for specifying inheritance or delegation, but rather, we introduce simpler mechanisms that can support a wide range of code sharing strategies without selecting one among them as a

(*) Named after I. Sina (980-1037), whose contribution to medicine has been used up to the present century. st stands for the Smalltalk-based implementation.

language feature. Besides, Sina/st provides a stronger data encapsulation than most of the existing object-oriented languages such as Smalltalk [Goldberg83] (**) and Flavors [Moon86], which weaken encapsulation through the introduction of inheritance [Snyder86a].

The focus of this paper is primarily on the data abstraction mechanisms used in Sina/st rather than on the complete definition of the language. This language has been implemented on the SUN 3 workstation using Smalltalk. All examples presented in this paper have been implemented and tested as parts of complete Sina/st programs.

This paper is organized as follows. The following section introduces the basic concepts of the computation model of Sina/st. In section 3 we show that the data abstraction mechanism presented here can be used to construct various abstraction techniques such as inheritance, delegation and aggregations. Data abstraction and encapsulation properties of Sina/st are studied in section 4. Section 5 gives an assessment study on Sina/st.

2. The Computation Model of Sina/st

The classical object model that provides abstract operations and encapsulates its internal state is selected as a basis for the computation model of Sina/st. Like in many other object-oriented languages, objects implementing the same type of component are classified under a type concept. Encapsulation is assumed to be a fundamental issue to object-oriented programming, and therefore it is strictly enforced. A type definition has two parts: interface definition and local definition. The interface part defines all the interface methods and objects and an interface predicate. The local definition consists of three parts: definition of all local objects and methods, an initialization process, and process descriptions of all local and interface methods. Figure 1 shows type *point* as an example of the structure of the type definition modules. *point* has the following abstract operations:

location returns the coordinates values of a movable display point of type *point*. *move* takes coordinate values as parameters and replace the current values with the new ones. *display* takes a string as a parameter and prints the string at the current location.

The invocation of a method is based on the remote procedure call model of synchronous communication. The messages to an object are queued at its interface. No assumption is made about the amount of buffer space available for queuing messages. An object interacts with another object by invoking its interface methods. This requires that in the invocation statement the method name be qualified with the name of the object. In this case, the request message is sent to the interface queue of that object. For example, a user of the object *display_point* of type *point* can change the coordinates of the point by executing the operation *display_point.move(coordinate)*, where *display_point* is the *receiver* object, *move* is the message selector (the method to be invoked), and *coordinate* is the message argument. This results in sending a request message to the object *display_point*. At some later time *display_point* would accept the message and evaluate it with respect to its interface *predicate*. The introduction of the interface predicate construct is the main difference between Sina/st and other object-oriented languages. An interface predicate defines the rules of its object. If the selector and the argument of the received message are valid with respect to the predicate, then this message is further sent to the method, i.e. *move* in this case. *move*, as a final receiver of the message, executes the requested operation and returns the result of the invocation.

```
type point interface is
 begin
  method location() returns integer [2];
  method move(integer as place[2]) returns nil;
  method display(string as datastream) returns nil;
  messages { self.location(), self.move(*),
             self.display(*)
           };
 end;
type point local is
 begin
  objects     /* local objects and methods  used
                 for implementing point. */
  initial     /* description of the process to
                 initialize the local objects */
  methods     /* description of interface/local
                 methods */
   location:  /* process description of the method
                 to obtain the coordinates of the
                 point */
   move:      /* process description of the method
                 move to  change the coordinates of
                 the point */
   display:   /* process description of the method
                 display to display a string at the
                 current location */
 end;
```

Figure 1.Structure of type definition module

A predicate is an ordered set of a message expression or a combination of message expressions with *connecting operators* such as *selection* "," , *exclusion* "\", or *aggregation* "<,,>". The components of a predicate are called propositions of predicate. In the present definition of Sina/st, predicate constructs are rather simple. Quantifiers (variables) and constants are not allowed as message arguments, and *if then else* like constructs are not defined for predicates. These extensions, however, are currently under consideration. Using connectives we can string lots of message expressions together to form a complex predicate. The interpretation of predicates is defined in the next paragraph. The following chapter illustrates how various forms of predicate constructs can simulate some well-known abstraction techniques such as single and multiple inheritance, delegations and aggregations.

Predicates have the following interpretation:

(R1) The receiver object (hereafter called target) in a proposition must satisfy the following requirements:

(R1a) the target is in the scope of the object which *owns* the predicate. An object is the owner of a predicate if its type declaration module defines this predicate;

(R1b) the target is declared at the interface of the owner object.

For example, in figure 1, the interface predicate of *point* is specified as a set of messages:

{self.location(), self.move(*), self.display(*)}.

The connecting operator used here is the *selection*, the propositions are *self.location()*, *self.move(*)* and *self.display(*)*, and the target object is also the owner (self) of the predicate. In Sina/st the pseudo-variable *self* in a message expression always refers to the instance of the type in which this message expression is defined.

(**) Smalltalk-80 is a trademark of Xerox Corporation.

(R2) If the target is the owner, then a proposition can be reduced to a method selector. For example, the predicate of type *point* can be reduced to the following declaration:

{ location(), move(*), display(*) }.

(R3) The character "*" in a proposition has the following interpretation:

(R3a) If the character "*" is used as a message argument, then the validity of the argument for a given message is determined by the corresponding method declaration. The propositions *move(*)* and *display(*)*, for instance, do not impose any restriction on the message arguments.

(R3b) If the character "*" is used within a message selector, then this selector is replaced by all the matching selectors of the target objects in an alphabetic order. Such a specification may generate a set of propositions. These generated propositions are combined together with a selection operator. Here, a star character indicates that any character can occupy that position and all the remaining positions.

For example, if we replace the predicate of type *point* with *{ *(*) }*, then this predicate will be equivalent to the following declaration:

{self.display(*), self.location(), self.move(*)}

Notice that the order of propositions above are different than the previous declaration.

(R4) A message *m* is valid for a proposition *r*, if the message selector and the arguments of *m* and r match with each other. We generalize this statement as *selector(m) = selector(r)*. The truth value of a predicate p depends on the truth values of its propositions, and the connecting operators used. All the connecting operators are left associative.

(R5) The connecting operator *selection* "," has the following meaning:

If predicate $p = \{r_1, r_2, .., r_n\}$, then p is true for message m, if

$$\bigvee_{i=1}^{n} (\text{selector}(m) = \text{selector}(r_i)).$$

∇ is a "conditional-or" operator

where

$C1 \nabla C2 \equiv$ if C1 then true
else C2.

(R6) The connecting operator *exclusion* "\" is defined as follows:

If predicate $p = \{r_1 \backslash r_2\}$, then p is true for message m if

$(selector(m) = selector(r_1)) \wedge (selector(m) \neq selector(r_2))$

Such a predicate is only meaningful if r_1 and r_2 share the same target object. The Sina/st compiler, therefore, checks the validity of exception operators.

(R7) The connecting operator *aggregation* "<..>" defines an indivisible message.

If predicate $p = \{ <r_1, r_2, .., r_n> \}$, then p is true for message m, if message m is an atomic message with n message selectors, and

$$\bigtriangleup_{i=1}^{n} (\text{selector}_i(m) = \text{selector}(r_i)), \text{ where } n \geq 2.$$

$\bigtriangleup$ is a "conditional-and" operator

where

$C1 \bigtriangleup C2 \equiv$ if C1 then C2.

(R8) Predicates can be nested. For example, the predicate

{ bh_point.(*)\ {move(*), display(*)} }*

excludes the *move* and *display* operations from the interface of *bh_point*.

3. Examples for Data Abstraction Techniques in Sina/st

3.1. Single Inheritance

Type definitions of Sina/st allow the programmer to declare variables at the interface of an object. These variables are potentially in the scope of the users of the object. Type *history_point* is such an implementation. As shown by figure 2, *history_point* is a *point* with an additional method called *history*. The *history* operation returns a list containing the sequence of locations the *point* has had.

```
type history_point interface is
 begin
  objects point as currentPoint;
  method move(integer as place[2])
         returns nil;
  method history() returns integer [100];
  messages {currentPoint.*(*)\{move(*)},
            move(*), history()};
 end;
type history_point local is
 begin
  objects integer as count, record[100];
  initial begin
           record[0]:= 0; record[1]:= 0;
           count:=2;
          end;
  methods
   move: begin
          currentPoint.move(place);
          record[count]:= place[1];
          record[count+1]:= place[2];
          count:= (count + 2) mod 100;
         end;
    history: begin return record end;
 end;
```

Figure 2. Type history_point.

The message predicate of type history_point is defined as

{ currentPoint.(*)\{move(*)}, move(*), history(*) } (3.1.1)*

Here, *currentPoint* is an instance of type *point* and *move* and *history* are the methods defined by type *history_point*. When a message is received by an object which enforces the predicate above, the received message is evaluated first with respect to the precedence rules of the connecting operator "," , which is from left to right. Since the left most element of the predicate is an instance of some other type, there is a hierarchy of interface predicates. The interface of *currentPoint* is specified in its type module as

{ location(), move(*), display(*) } (3.1.2)

Using rules (R3b) and (R5), from predicates (3.1.2) and (3.1.1) we obtain

{ display(*), location(), move(*)}\{move(*)}, (3.1.3)
move(*),history(*) }

The received message is evaluated first with respect to the interface predicate of *currentPoint*. This evaluation continues until the predicate of *currentPoint* is satisfied or all the other elements of the predicate of type *history_point* are evaluated. If all the predicate elements fail, then an exception is raised. Since for every *move* operation the coordinates must be stored in the history record, the *move* operation of *currentPoint* is overwritten by declaring it as a new method in *history_point*. The method *move* updates the value of *currentPoint* by performing the *move* operation on *currentPoint* directly. Performing an operation on the interface objects is like performing an operation on *self*, except that the search for the operation to invoke starts directly in the predicate of the interface object instead of the predicate of *self*. Hence, this is equivalent to a call on pseudo-variable *super* of Smalltalk. Obviously, type *history_point* inherits the abstract operations of type *point*.

3.2. Multiple Inheritance

Multiple inheritance can be easily introduced by declaring multiple variables at the interface and making them public by means of an appropriate predicate. In order to illustrate the simulation of multiple inheritance in Sina/st, two additional types *bounded_point* and *bh_point* are introduced.

Type *bounded_point* is a *point* with restricted display coordinates. This type introduces four new operations and a modification to the operations of type *point*. The *min* operation returns the current lower bound for the *point*. The *setmin* operation changes the lower bound of the *point* if the new value is smaller than the current location value. The *setmax* operation changes the upper bound of the *point* if the new value is greater than the current location value. The *move* operation will not move a point to a location outside boundaries. Figure 3 illustrates the interface declaration of *bounded_point*.

The message predicate of *bounded_point* includes the instance *currentPoint* of type *point* at its interface, excludes the *move* operation of *currentPoint*, and defines five new methods.

Using rules (R2), (R3) and (R5), if we replace the character "*" with the corresponding message selectors, then we obtain

{ display(*), location(), move(*)}\{move(*) }, (3.2.1)
max(), min(), move(*), setmax(*), setmin(*) }

This predicate, again, simulates a single inheritance.

```
type bounded_point interface is
 begin
  objects point as currentPoint;
  method move(integer as place[2])
         returns boolean;
  method min() returns integer [2];
  method max() returns integer [2];
  method setmin(integer as coordinate[2])
         returns boolean;
  method setmax(integer as coordinate[2])
         returns boolean;
  messages
    { currentPoint.*(*)\{move(*)}, *(*) };
 end;
type bounded_point local is
 begin
  ....
 end;
```

Figure 3. bounded_point.

Type *bh_point* combines the features of types *history_point* and *bounded_point*. Type *bh_point* is a *bounded_point* with a *history* operation. *bh_point* also defines a new method *bounds_history* which returns a list whose elements have the form (min, max, number, history_length). This list shows the length of the historylist when min/max was changed to number. The *bounds_history* can be used together with the *history* operation to reconstruct the sequence of state changes the *bh_point* has had.

The interface declaration of type *bh_point* is given by figure 4. *bh_point* is a synthesis of types *bounded_point* and *history_point*, and therefore, instances of these types are defined at the interface of *bh_point*.

Applying rules (R3) and (R5), the interface predicate of bh_point yields

{ bounds_history(), move(*), setmax(*), setmin(*), (3.2.2)
limited_point.(*), set_points.*(*) }*

The abstract operations of objects *limited_point* and *set_points* are also available to the users of type *bh_point*. Since the operators are left associative, the methods *move*, *setmin* and *setmax* of *bounded_point* and the method *move* of *history_point* are overwritten by *bh_point*. This example can be seen as a simulation of multiple inheritance.

```
type bh_point interface is
 begin
  objects bounded_point as limited_point;
  history_point as set_points;
  method move(integer as place[2])
         returns boolean;
  method
   setmin (integer array as coordinate[2])
   returns boolean;
  method
   setmax(integer array as coordinate[2])
   returns boolean;
  method bounds_history()
        returns integer [N];
  messages { *(*), limited_point.*(*),
             set_points.*(*)
           };
 end;
type bh_point local is
 begin
  ....
 end;
```

Figure 4. bh_point.

3.3. Delegation

In this section we introduce two new types *solid_line* and *dashed_line*. *solid_line* inherits from *bh_point* and defines methods *forward_line*, *forward_move* and *back*, and the object *degree* at its interface. *forward_line* takes the number of steps as a parameter and draws a line whose direction is defined by *degree*. *forward_move*, without drawing any line, moves the current location with respect to the number of steps required, and the given *degree* of move. *back* negates the number of steps and calls *forward_line*.

```
type solid_line interface is
 begin
  objects point as bh_point;
          angle as degree;
  method forwardline (integer as step)
         returns nil;
  method forwardmove(integer as step)
         returns nil;
  method back(integer as step)
         returns nil;
  messages
   { bh_point.*(*), degree.*(*), *(*) };
 end;
type solid_line local is
 begin
  ....
  back: begin
         step:= -step;
         server.forwardline(step);
        end;
 end;
```

Figure 5. solid_line.

Lieberman, in his paper, shows that inheritance can not implement delegation because of the so called "self problem" [Lieberman86]. In delegation, the designated object is a part of the extended identity of the delegating object, and this property can not be simulated by inheritance constructs. In case of inheritance, the pseudo-variable self is automatically bound to the recipient of a message during the execution of a method. In general, it is not possible for the user to designate another object to reply in place of the object which originally received the message. In order to overcome the *"self problem"*, in figure 5, method *back* performs *forwardline* on *server* instead of *self*. Performing an operation on *server* causes the search for the invoked operation start with the *recipient* of the message. Since the objects in Sina/st can be nested within each other, the recipient of the message and the object in which the invocation appears can be different. We call the receiver of the message server, since this object can be thought of as performing service for the object where the message originates. The difference between *server* and *self* is demonstrated by the following example.

Type *dashed_line* has the same interface but it draws dashed instead of solid lines. Method *forwardline* in *dashed_line* break up the line in pieces and calls *forwardline* and *forwardmove* in *external_line* to draw a series of shorter lines. Method *back* in *dashed_line* is delegated to method *back* in *external_line*.

As illustrated by figure 6, *external_line* is an instance of type *solid_line*, and it has been created as a global variable. Any message except *forwardline* is first delegated to *external_line*. If an instance of *dashed_line* delegates a message to *external_line*, then invocation on *server* in *external_line* by method *back* will refer to the instance of *dashed_line*. However, if *self* was used instead of *server*, then method *back* in *external_line* would invoke *forwardline* on *external_line*, and therefore, it would be impossible to draw dashed lines. According to the language definition of Sina/st, binding of objects are realized in compile-time, whereas for delegated (*external*) objects, binding is delayed to run-time. The programmer, therefore, has the possibility to implement incrementally and dynamically modifying components of systems by using delegation constructs.

```
type dashed_line interface is
 begin
  objects
   solid_line as external_line external;
   integer as angle;
  method forwardline(integer as step)
         returns nil;
  messages
   { external_line.*(*)\ {forwardline(*)},
     forwardline(*)
   };
 end;
type dashed_line local is
 begin
 ....
 end;
```

Figure 6. dashed_line.

3.4. Aggregations

Many applications require to capture and enforce the *is-a-part-of* integrity constraint between two or more objects [Kim87]. An aggregation enables an object to be a part of another object, and it is one of the fundamental data modeling concepts [Smit77]. To demonstrate how Sina/st is capable to simulate aggregations, we introduce two new types, *login_receive* and *protected_line*. Type *login_receive* is defined with the following methods:

login accepts *name* and *password* as parameters and returns true if these parameters form an authorized pair. Similarly, *owner* returns true if the given pair of parameters is defined as the owner's name and password. If the owner is not yet defined, then these parameters are stored as the owner's identity. In all other cases, the message processing is interrupted by raising an exception condition.

```
type login_receive interface is
 begin
  method login(string as name, pwd)
         returns boolean;
  method owner(string as name, pwd)
         returns boolean;
  messages { *(*) };
 end;
type login_receive local is
 begin
 ....
 end;
```

Figure 7: login_receive

The operations defined by type *login_receive* can be required by many objects to refuse the request of unauthorized users. This requirement can be easily fulfilled by creating an instance of type *login_receive* at the interface, and connecting it to another object using the aggregate construct. Figure 8 illustrates such an example. Type *protected_line* is a *dashed_line* whose methods are protected by an instance of type *login_receive*.

```
  objects login_receive as verify;
          dashed_line as line;
  messages ( <verify.*(*), line.*(*)> );
 end;
type protected_line local is
 begin
 ....
 end;
```

Figure 8: protected_line

The predicate of type *protected_line* is the aggregation of the predicates of *login_receive* and *dashed_line*. Operations on *line* can only be invoked, if object *verify* terminates its processing successfully.

Aggregate construct "<..>" implies the receipt of messages atomically. Since messages are processed one at a time, elements of such a message are accepted and returned in sequence. Before starting with processing, the validity of an atomic message consisting more than one message expression is checked by examining the elements of the input buffer. Until the last message component is processed and returned, an atomic message binds the client and server objects to each other. In order to specify atomic messages, Sina/st introduces the following syntactic construct:

receiver_object.< m1, m2, .. mn>,
where m1 .. mn are the list of messages to be sent atomically.

If, for instance, *pr_line* is an instance of type *protected_line*, then *"pr_line.<login(name, pwd), forward_line(20)>"* is syntactically a valid message.

4. Data Abstraction and Encapsulation in Sina/st

Most object-oriented languages such as Smalltalk and Flavors weaken encapsulation through the introduction of inheritance [Snyder86a]. In the following sections we compare the data encapsulation property of Sina/st with other object-oriented programming languages.

4.1. Inheriting Instance Variables

The degree of encapsulation property of an object can be measured by examining how freely an implementation of this object can be changed without affecting its interface. In most object-oriented languages, operations can directly process their own instance variables and also the instance variables of their ancestor classes. The implementor can not change the implementation (rename, etc.) of an instance variable freely, without considering its effects for its descendants. Instance variables must be protected from direct descendant classes as they are protected from direct users of an object. Snyder claims that if the designer of a class needs access to an inherited instance variable and the appropriate operations are not defined, the correct thing to do is to negotiate with the designer(s) of the ancestor class(es) to provide these operations [Snyder86a].

The data abstraction model of Sina/st does not allow objects to process the instance variables of ancestor classes directly. For instance, as illustrated by figure 5, the methods of type *solid_line* can not invoke operations on the instance variables of *bh_line*. If methods of *solid_line* require to access the instance variables of *bh_line*, then *bh_line* must provide the necessary methods.

Inheritance is generally accepted as an implementation issue, and therefore, it should be invisible for the users of an object. The data abstraction model of Sina/st allows programmers to change the inheritance hierarchy without affecting the message interface of objects. Suppose that type *history_point* is reimplemented and it no longer inherits from type *point* but it still supports the same behavior. The reimplementation of type *history_point* is shown by figure 8. The functional interface of type *history_point* is remained unchanged, because the users of *history_point* can invoke the same operations on the instances of *history_point* without modifying any code.

```
type history_point interface is
 begin
  method location() returns integer [2];
  method move(integer as place[2])
         returns nil;
  method display(string as datastream)
         returns nil;
  method history() returns integer [100];
  messages ( *(*) );
 end;
type history_point local is
 begin
  objects ...
  initial ...
  methods
   location: ...
   move: ...
   display: ...
   history: ...
 end;
```

Figure 8. Modified history_point.

4.3. Excluding Operations

Most object-oriented languages can not exclude the inherited operations although this feature is generally desired. In languages like Smalltalk, a class can redefine the methods of its ancestor classes, but, however, it can not exclude them. CommonObjects [Snyder86b] is a language that provides a selective inheritance. In Sina/st, programmers can also exclude the operations by using the exclusion "\" operator at the interface predicate.

4.4. Multiple Inheritance

The use of multiple inheritance may create name conflicts if the parents have operations and/or instance variables having the same name. The conflict resolution techniques available today, either weaken the encapsulation (Smalltalk and Trellis/Owl [Schaffert86]), or may initiate undesired operations (Flavors and CommonLoops [Kempf87]), or may create redundant objects (CommonObjects). Sina/st introduces a concept of hierarchically nested encapsulations, which eliminates the above mentioned problems. According to the message semantics of Sina/st, the message interface of an object is specified as a predicate with a certain precedence order in evaluation. This technique resolves the name conflict problem because always one operation is selected. In order to avoid programming errors, the programming environment of Sina/st warns the user about the possibility of inheriting operations from more than one type.

There are mainly three strategies for dealing with multiple inheritance:

(1) modeling the inheritance graph directly;

(2) modifying the graph into a linear chain;

(3) converting the inheritance hierarchy into a tree by duplicating nodes.

4.4.1. Graph-Oriented Solutions

Trellis/Owl and Smalltalk have this strategy. Operations are inherited along the inheritance graph until redefined in a class. One way to resolve the name conflict problem is to redefine the operations in the child class so that this definition can invoke the operations defined by the parent class. If the graph is not a tree, i.e. when a single class is reachable from the root class by multiple paths, Trellis/Owl and Smalltalk (***) signal an error. However, if a class inherits operations with the same name from two or more parents, but the operations are not different, both of these languages do not signal error. This conflict resolution strategy is not desirable, because it makes inheritance to be part of the external interface.

In the languages that adopt graph-oriented solutions, only one set of instance variables is defined for any ancestor class, regardless of the number of paths by which the class is reached in the inheritance graph. This semantics potentially introduces problems because operations can be invoked twice on some instance variables.

4.4.2. Linear Solutions

This strategy is used in Flavors and in CommonLoops. These languages first flatten the inheritance graph without duplicates, and treat the result as a single inheritance. There are also a number of problems with this strategy. Firstly, unwanted methods can be selected in case of name conflicts. Secondly, a class may have a difficulty to communicate with its real parents if some other class is inserted before its real parent.

4.4.3 Tree Solutions

This strategy is adopted by CommonObjects. Although this strategy is similar to the graph-oriented solutions, there are two important differences:

(1) An attempt to inherit operations from more than one parent is always an error regardless of the sources of operation. The designer must revise the class definition to explicitly select one method or redefine the method. Also, in CommonObjects, the user can specify which operations of its parents are or are not included in its own external interface. To provide access to methods that are not inherited, a special construct *call-method* can be used within a method to invoke an operation, given the name of the parent and the name of the operation. This is similar to the *super* construct of Smalltalk.

(2) The inheritance graph is converted into a tree by duplicating nodes. For example, if two parents define an instance variable x, instances of the class will contain two instance variables x, one for each parent. There is no merging of instance variables even if a class is inherited more than once. Further, instance variables may not be accessed directly by a child class, but may be accessed only by invoking operations defined by the parent. To achieve the effect of direct instance variable access, these methods can be invoked by an operation called *call-method*.

There are similarities between the strategy of CommonObjects and Sina/st. Both of these languages aim at a stronger encapsulation than conventional object-oriented languages. However, there are also important differences:

(1) An attempt to inherit operations from more than one parent is not an error in Sina/st. An operation will be selected according to the precedence rules of the interface predicate.

(2) Both Sina/st and CommonObjects do not merge instance variables. They duplicate instances of ancestors. Although this approach supports encapsulation, it creates redundant objects. In Sina/st the pointer for an object X can be obtained by evaluating the expression &X; for pointer P, the object pointed to by it is given by the expression @P. A pointer object P for type T is declared in the object declaration part of a type as

objects @T as P;

This feature of Sina/st allows the programmer to merge the instance variables of ancestors, if the owners of the ancestor types provide an access to the pointers of these local variables. If the designer of a type needs to merge some of the instance variables of its ancestors for his own application, he should negotiate with the designers of the ancestor types, so that they provide operations to access the pointers of their instance variables. The descendant class, then, can alias these variables to each other. This approach does not weaken the encapsulation of objects, because any class can change its internal implementation without having an effect on its interface. In other words, descendant types only inherit the abstract specifications of their ancestors.

(3) Sina/st integrates everything into a simple construct: interface predicates. CommonObjects defines language features for inheritance specification and for method exclusion.

5. Comparison and Evaluation

In comparison to conventional object-oriented languages, Sina/st provides message predicates, declaration of variables at the interface of an object, local methods and mechanisms to group messages as an atomic construct. The abstraction mechanisms that have been introduced through out this paper are depicted by figure 9.

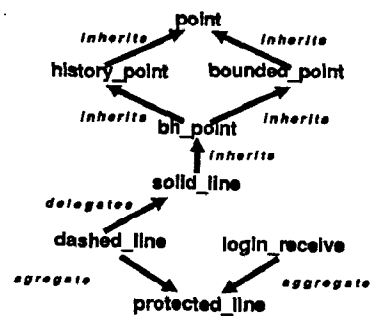

Figure 9. Hierarchy of data abstractions.

(***) We refer to the multiple inheritance extension of Smalltalk defined by [Borning82].

Although Sina/st can simulate most of the abstraction techniques that are available in conventional object-oriented languages, there are a number of fundamental differences between these languages and Sina/st:

(1) The most important difference is that Sina/st does not introduce code sharing strategies like inheritance as a language construct.

(2) In Sina/st, types can be defined upon hierarchically nested instance objects that belong to various types. This hierarchical structure is fundamentally different than hierarchical organization of classes, because in Sina/st every instance of a type has its own distinct hierarchy as these hierarchies are formed from instances instead of classes. For example, type *solid_line* inherits the abstract operations of types *bh_point, history_point, bounded_point* and *point*. This inheritance structure is simulated by creating instances of the corresponding types at the interfaces of objects and by making them public by means of appropriate predicate constructs. These instance objects may have their own state, and therefore, they define explicit and concrete object structures. Classical object-oriented languages define implicit and abstract classes embedded into an object's structure. Most of these languages also merge the instance variables through the inheritance hierarchy, thus creating flat object structures. It is our experience that in order to simulate "real-world" systems, one has to be able to construct objects nested within each other. In general, objects in "real-world" systems impose hierarchically organized encapsulations of data and processes, and Sina/st directly simulates such a structure.

(3) Most of the existing object-oriented concurrent languages define the unit of modularity and concurrency to be the same, and therefore can create only concurrency between objects. On the other hand, "real-world" objects are characterized by intra-object concurrency, and languages that would allow concurrency within an object could be more expressive. One objection to such a construct is the increased complexity. In another related paper [Tripathi&Aksit88] we have described in detail the use of SINA in programming several well-known synchronization problems in scheduling, communication, and resource management. Sina/st allows intra-object concurrency by defining hierarchically nested objects creating nested encapsulations. While this approach provides fine-grained parallelism and more expressiveness, it also does not increase the complexity because the concurrent activities are well-structured through the encapsulated hierarchies.

(4) Predicates are not restricted only to those constructs that are presented in this paper. One can build and integrate various useful disciplines into the structure of an object. We are currently experimenting with predicate constructs to build "communication abstractions" [Francez86] and "reflective architectures" [Maes87].

Although designed independently (****), the law-based systems [Minsky87] show similarities to the computational model of Sina/st. Both Sina/st and law-based systems argue that, starting from a very primitive foundation, one can establish various forms of abstractions and useful disciplines by carefully using these primitives. In law-based systems, the programmer defines the discipline under which his system operates. This technique is not restricted to object-oriented programming, and even does not assume any data encapsulation. Like in Sina/st, objects in law-based systems communicate with each other by sending messages. These messages are not delivered immediately, but intercepted by a Prolog interpreter which enforces the laws of the system expressed in terms of Prolog clauses. It is possible to simulate encapsulation, inheritance and other useful disciplines by defining appropriate laws. Both Sina/st and law-based systems claim to construct more complex systems from user defined rules. In law-based systems rules are Prolog clauses which define the system laws, in Sina/st rules are input interface predicates which specify the message protocols of objects. Although there is much similarity between these two systems, there are also a number of important differences:

(1) In Sina/st, every object has its own manager which is responsible for queuing, parsing, checking interface predicates and invoking operations. In law-based systems, rules are enforced by the system. As a consequence of this, Sina/st provides more modular software, because every object has its own predicates to enforce, and the system is only responsible for message transfer. Distributing the rules of the system to objects fits more into the object-oriented and distributed programming style.

(2) Sina/st's approach is also more resilient to failures than law-based systems. If an object crashes, its failure can be limited to the boundary of this object. However, in case of a centrally controlled system, a system failure will have an influence to all the elements of the system.

(3) As Minsky also admits [Minsky87], Prolog is a very powerful language and therefore, the programmer can shift the programming task to Prolog, and he leaves this choice to the programmer. We believe that it is dangerous to adopt Prolog for defining laws. Leaving the decision to a programmer to use small and simple laws is not acceptable from a software engineering point of view.

(4) Minsky does not support encapsulation as a fundamental language issue [Minsky87]. However, we consider encapsulation fundamental and our system implements types, objects, messages and encapsulation of objects as language features. Although encapsulation can be enforced by a law, it is more efficient to introduce it as a language feature instead of as a system law.

(5) Minsky introduces a number of message types with different semantics to implement the law-based system. Although data encapsulation and inheritance are not specified by a new language construct, laws alone would not be sufficient without introducing additional message constructs. Sina/st, however, has uniform message syntax and semantics. Minsky claims that the user only deals with one type of message, and the others are hidden in the implementation of the system. However, the user need to be aware of these message types in order to understand how the system works.

(****) The early version of the Sina/st language has been designed in 1985 [Aksit85]. Due to the confidentially of the project at that time, however, no external publications were submitted.

6. Conclusions

In this paper we have presented a new data abstraction mechanism in an object-oriented model of computing. The model of data abstraction presented here has been included in the Sina/st programming language. We have shown that by means of a simple message predicate construct, one can simulate various forms of abstractions without selecting one among them as a language feature. The computation model of Sina/st also provides a stronger data encapsulation than conventional object-oriented languages.

ACKNOWLEDGEMENTS

The Sina/st language has been implemented by a group of students on the SUN 3 workstation as their Master's degree projects at the University of Twente. This implementation is based on Smalltalk-V2.2. Ruud Nijhuis has implemented the Sina/st compiler to translate Sina/st type declarations to Smalltalk data structures. The object manager and concurrent processing constructs have been implemented by Hans Bank. Gerard van Wageningen has implemented the Sina/st interpreter. Delegations and atomic constructs have been implemented by Jan Willem Dijkstra. The user interface of Sina/st is currently being developed by Willem Veldkamp.

REFERENCES

[Agha86]
G. A. Agha, Actors:A Model of Concurrent Computation in Distributed Systems, (eds) P. H. Winston et al, MIT PRESS, 1986

[Aksit85]
M. Aksit, Introduction to Sina-0: An Object-Oriented Language for Distributed Systems, Int. Rep: 8522.043, Oce Research lab.- the Netherlands, July 1985

[Borning82]
A. Borning and D. Ingalls, Multiple Inheritance in Smalltalk-80, Proc. AAAI, pp 234-237, 1982

[Carnese84]
D. J. Carnese, Multiple Inheritance in Contemporary Programming Languages, MIT Lab. of Comp. Science MIT/LCS/TR-328, September 1984

[Francez86]
N. Francez and et al, Script: A Communication Abstraction Mechanism, and its Verification, Science of Computer Programming, Vol. 6, No. 1, pp 35-88, 1986

[Goldberg83]
A. Goldberg and D. Robson, Smalltalk-80, The Language and its Implementation, Adison Wesley, 1983

[Kempf87]
J. Kempf and et al, Experience with Commonloops, OOPSLA'87 Proceedings, pp 214-226, October 1987.

[Kim87]
W. Kim and et al, Composite Object Support in an Object-Oriented Database System, OOPSLA'87 Proceedings, pp. 118-125, October 1987

[Lieberman86]
H. Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Sys., ACM OOPSLA'86 Proceedings, pp. 214-223, September 1986

[Maes87]
P. Maes, Concepts and Experiments in Computational Reflection, OOPSLA'87 Proceedings, pp. 147-155, October 1987

[Minsky87]
N. H. Minsky and D. Rozenshtein, A Law-Based Approach to Object-Oriented Programming, OOPSLA'87 Proceedings, pp. 482-493, October 1987

[Moon86]
D. A. Moon, Object-Oriented Programming with Flavors, OOPSLA'86 Proceedings, pp. 1-8, September 1986

[Nguyen86]
V. Nguyen and B. Hailpern, A Generalized Object Model, Sigplan Notices, Vol. 21, No. 10, pp. 78-87, October 1986

[Rumbaugh87]
J. Rumbaugh, Relations as Semantic Constructs in an Object-Oriented Language, OOPSLA'87 Proceedings, pp 466-481, October 1987.

[Schaffert86]
C. Schaffert and et al, An Introduction to Trellis/Owl, OOPSLA'86 Proceedings, pp. 9-16, September 1986

[Smit77]
J. M. Smit and D. C. P. Smit, Database Abstractions: Aggregation and Generalization, ACM TODBS, Vol. 2, No. 2, pp 105-133, June 1977

[Snyder86a]
A. Snyder, Encapsulation and in Object-Oriented Programming Languages, Proceedings of OOPSLA86, pp. 38-45, September 1986

[Snyder86b]
A. Snyder, CommonObjects: An Overview, Sigplan Notices, pp. 19-28, October 1986

[Stein87]
L. A. Stein, Delegation is Inheritance, OOPSLA'87 Proceedings, pp 138-146, October 1987

[Tripathi&Aksit88]
A. Tripathi and M. Aksit, Communication, Scheduling, and Resource Management in SINA, *to appear in the Journal of Object-Oriented Programming*, Vol. 1. No 3. (July/August 1988).

[Wegner87]
P. Wegner, Dimensions in Object-Based Language Design, OOPSLA'87 Proceedings, pp. 168-182, October 1987