A Generalized Object Model

Van Nguyen
(914-789-7798; vnguyen@ibm.com)

Brent Hailpern
(914-789-7799; bth@ibm.com)

IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

ABSTRACT

Since the introduction of Smalltalk, the object-oriented paradigm has become a popular technique for the organization of data and programs. In this paradigm, objects (collections of data and programs) are organized in a class hierarchy. These classes collect similar objects and serve as a repository for methods (programs) shared by those objects. We present a new simple model of objects that allows multi-dimensional inheritance. Objects, in our model, behave like communicating processes in Hoare's CSP language, but with a different message-passing mechanism. This leads to a simple formal semantics for objects and inheritance.

1. Introduction

1.1 Object-oriented programming

There are two main approaches to structuring of data within programs: type theory and object oriented. In the type-theory approach, each datum is considered to be an element of a type. Complex types are formed out of basic types by mathematical operations such as cartesian product, disjoint sum, and function space. This approach results in elegant mathematical theories. Conventional imperative programming languages, such as Pascal, use a modified version of this approach. These languages provide primitive types (for example, boolean, character, and integer) and type constructors (such as, arrays and records).

The object-oriented approach combines three properties: encapsulation, inheritance, and organization. *Abstract data types* encapsulate data and operations into objects (for example, in Modula these objects are called *modules*). Each such object has its own private memory and local functions, resulting in modularity and information hiding. The more general *object-oriented* technique further organizes this set of encapsulated objects into a hierarchy (usually called a class or subclass-superclass hierarchy). This hierarchy serves to organize similar objects into groups called *classes* and then to organize similar classes into *superclasses*. Objects at any level of the hierarchy *inherit* all attributes of higher-level objects. This inheritance makes it possible for similar objects to share program codes. During execution, the search for an attribute begins at some level of the hierarchy and proceeds to the top - taking the first instance of the attribute that is found. Hence, an attribute at a lower-level hides an attribute of the same name at a higher level. This short circuit of the search

allows a subclass to customize the more general attributes of its superclass. Smalltalk is the best known language in this category.

1.2 Background

It is beyond the scope of this paper to detail all the different approaches to object-oriented systems. Some approaches are described in [D, GR, CBLL, Ha, LH, Ca]. A survey of the field can be found in [R]. A comparison of abstract data types and object-oriented programming can be found in [HW].

Object-oriented programming originated with Simula [D, DDH]. A Simula class is an abstract data type. In Simula, one class X can be a refinement of another class Y. That is, X *inherits* all the attributes and actions of Y. In addition, X can add attributes and actions of its own. Dahl called this program concatenation and he described it as a textual substitution, rather than a runtime inheritance.

Smalltalk [GR] is probably the best known object-oriented language. In Smalltalk, there are two basic units: objects and classes. Classes contain function definitions (called methods) and data declarations. Every object is an instance of some class. That is, an object contains exactly those variables defined by it class. All operations on an object are defined in the class of that object, or in the superclass of that class, or in the superclass of the superclass, and so on. The top-level superclass is called *Object*. All classes are refinements of the superclass *Object* in that they add new or different methods or allow for more variables in their instances. Of course, classes themselves can be considered objects and are instances of their metaclasses. As an object, a class cannot contain it own specialized methods. These special methods are kept in the metaclass of the class.

Smalltalk objects interact by exchanging messages. In addition to message passing, different objects of a class can share variables, called *class variables*. Class variables are defined in the metaclass of the class and are accessible to any method defined in the class.

Smallworld [LH] is a shell language that uses the object-oriented paradigm. It is not intended to be a programming language, but rather a system for organizing files, programs and system commands. Its object model is different from that of Smalltalk in that an object is an independent entity and is not an instance of its class. That is, an object is a collection of properties: each property has a name and a value. Objects are grouped into classes (as defined by the class property of each object) solely for organization purposes. The member objects of a class need not have the same or even similar structures. A Smallworld object can define its own properties and methods, where methods are merely properties with the suffix *.method*. If an object does not have a requested property p or method m, then it asks its class for property *inherited:p* or method *inherited:m*. If not found, the request follows the superclass chain, until it reaches the top-level class, *universe*. Smallworld classes are normal objects (with class=*class*) and hence they can customize their own methods, eliminating the need for metaclasses. Furthermore, classes can provide inherited properties, which permit default values to be inherited from a class.

Smalltalk and Smallworld support only *single inheritance*. That is, each class has a unique superclass. The resulting class tree is often overly restrictive. Consider, for example, the following structure [Ca], where we consider *age* to be the function that returns the value of the age of the object.

```
class object  = <age>
class vehicle = <age, speed>
class machine = <age, fuel>
class car     = <age, speed, fuel>
```

In this example, *vehicle* and *machine* are subclasses of *object*. The class *car* is a subclass of both *vehicle* and *machine*, since it inherits methods from each. Any method to decrease the amount of fuel in a car should come from the machine description. Similarly any increase in speed should be manipulated by the vehicle description. In *multiple inheritance* systems [CBLL, CA, SBK], a class can inherit methods from more than one superclasses. Thus the subclass-superclass relation is no longer constrained to form a tree, but can form a directed acyclic graph.

Cardelli [Ca] gives a formal semantics of multiple inheritance. However, his notion of inheritance is type inheritance, and not class inheritance. That is, suppose that objects a_1 and a_2 are in the same class, and method m_1 of a_1 and method m_2 of a_2 are both inherited from the same method of the class. Then all that is required is that m_1 and m_2 are of the same type. They can be different methods!

The conventional approach to managing multiple inheritance is to allow a class to have many superclasses. Either all method names of the superclasses must be distinct, or a priority order is placed on the superclasses to resolve name conflicts. But even this model of multiple inheritance can be too restrictive. Sometimes inheritance can also depend on the *context* of the request: that is, on the object that requests the method as well as on the method itself. We term this *multi-dimensional inheritance*. For example, an object that manages a database may inherit a method from another object to answer some query from a supervisor, but it will not process the same query from an ordinary user. For another example, consider an object that manages a database of employee histories in some company. If a user requests his ranking in the company, then different methods (which may result in different inheritance paths) are used, depending on whether the user is a manager or a programmer.

The distinction between objects and classes (and metaclasses) forces the organization of the objects into a tree, DAG, or collection of intertwined trees. One is forced to assume the existence of a largest superclass and a class of all classes. This makes it cumbersome to expand a system of objects or to integrate two existing systems. For instance, if there are two hierarchies of objects and we want to integrate them, then we have to decide whether to create a new largest superclass or to use one of the two existing largest superclasses. In either case, the structures of several objects will have to be modified to permit this change.

In this paper we propose a new model of objects that handles multi-dimensional inheritance. The only primitive notions in the model are objects and inheritance; such concepts as classes and superclasses can be derived from them. This results in both simplicity and generality. Our model can simulate the object models of Smalltalk, Smallworld, and some other models in a clean and simple way.

The model we describe in this paper should be viewed as an experiment in language design. We are attempting to refine the notions of "object" and "inherit" exclusive of the notions of "class" and "type". Because we are deliberately excluding notions of type, some of our assumptions are simplistic and unsafe. However to understand what "inheritance" and "type" mean in combination, we must make an attempt to understand them in isolation. We intend to reintroduce some notions of type and organization in future refinements of this

work. In some sense, you can consider this work to be an *assembly language* for object-oriented systems, with all the connotations of type freedom and unsafe operations included. On the other hand, we are building from a sound semantics of objects and message-passing, as described in section 3.

The rest of the paper is organized as follows. Section 2 describes our model. In Section 3, we show how to interpret objects in the model as communicating processes, and we describe briefly a formal semantics for objects and inheritance. Section 4 will show how some other systems can be simulated by our scheme.

2. Our model

The central notion of our model is the *object*. An object consists of a set of methods, a set of local procedures, and a private memory (that is, a set of private variables). The variables and local procedures are not visible outside that object. There is no shared memory among objects. There are two types of methods: simple and inherited. A *simple* method has an executable program associated with it. An *inherited* method contains the names of other objects from which it inherits methods. Associated with each method is a *guard* of the form (**request**, *id*, *action*) or (**inherit**, *id*, *action*), where *id* represents the object requesting the method specified by *action*.

Objects can send (receive) requests and replies to (from) other objects. Objects can also inherit methods from other objects. A *request* includes the following fields

(**request**, *source*, *destination*, *action*, *parameters*)

where *source* is the id of the requester, *destination* is the id of the receiver of the request, *action* is the action to be performed on the receiver, and *parameters* are the parameters that are to be used when performing the action. An *inheritance* includes the fields

(**inherit**, *source*, *destination*, *action*, *parameters*)

where *source* is the id of the (original) sender of the inheritance, *destination* is the id of the receiver of the inheritance, *action* is the action to be performed and *parameters* are the parameters that are to be used when performing the action. A *reply* is of the form

(*object name*, *result*)

where *object name* is the name of the object receiving the reply.

When an object sends a request, it suspends all of its request activity until it receives a reply for that request. After an object receives a request, it searches the guards of its set of methods to find one that matches the tuple (**request**, *source*, *action*) of the request. If a match is found and it is simple, then the method is executed and the reply is sent to the requester. If an inherited method is found, then an inheritance is sent to a new object, using the id of the original source.

Conceptually, the inheritance is a request for the code body of the method to be executed (by the receiver of the original request). The new object then searches the guards of its set of methods to find one that matches the tuple (**inherit**, *source*, *action*) of the inheritance. If a match is found and it is simple, then the method code body is returned to the receiver of the original request, who then executes that body. If an inherited method is found, then a new inheritance is sent to another object, still using the id of the original requester.

If no guard matches the request, then an error is returned. This process is repeated until a simple method (or no match) is found.

An object can process only one request at a time, to avoid interference on its private memory. This is an instance of the well-known critical-section problem in distributed computing: if one allows concurrent access to shared variables, then unpredictable behavior can occur as a result of unrestricted access to these shared variables. The solution is to serialize access to the collection of shared variables (as we have done here) or to include explicit interprocess synchronization that guarantees atomic access to the shared resources.

On the other hand, an object can handle several inheritances simultaneously because its memory is not affected. Such an object is acting only as a code repository. For the sake of efficiency, we may actually desire the code repository to act as an agent to execute the code body: that is the reason we include the parameters in the inheritance. The execution of these code bodies must be independent of all other executions within the repository object, including independent executions of the same code body. To preserve our local semantics, the executions must use the private variables of the receiver of the original request. If this optimization is used, then we assume that a capability to these private variables can be sent with the inheritance.

In general, requests and inheritance could be combined by allowing some requests to return values and others to return functions. We prefer to separate these two concepts, because of the different blocking strategies, communication paradigms, and abilities to affect private memory.

One must keep the notion of locality of memory space clearly in mind when considering the evaluation of a request in our model. During the execution of a method, the method may send requests to other objects or make local procedure calls. All variable names in a simple method can refer only to the private memory of the object that receives the original request, any local variables of the method, the names of other objects, and the formal parameters of the action. Note that once a code body of an inherited method reaches a requestor, that code body can refer to the private memory of the requestor. That is, all methods *executing* within an object consider the private variables of that object to be global. This clearly has the potential for problems, should the imported code refer to non-existent variables or to variables intended for some other use. We intend to modify this naive assumption in future refinements.

Besides allowing exact matches on guards, we also permit wild cards, represented by *?*. The wild card *?* represents either **request** or **inherit**, or any object, or any action. For example, the guard (**request**, ?, *show*) would be satisfied by any requester that asks for the show command. The guard (?, *supervisor*, ?) will match any request or inheritance by an object named *supervisor*. We also have a distinguished id *user* representing requests from the user.[1] Suppose, for example, that we have a Smalltalk-like object that has no methods of its own: all methods are inherited from its class. That would be represented by:

```
(?, ?, ?) -> INHERIT FROM class {an object name}
```

On the other hand, in Smallworld, an object could have a special output format for the show command:

---

[1] Other wild cards might be considered: for example, the wild card *?o* could matches the id of any other object, but not that of the user.

```
(REQUEST, ? , show) -> {text of show function}
```

Furthermore, an object could show itself one way (request directly from a user), but provide a shared code for showing its member objects:

```
(REQUEST, user, show) -> {function to show myself}
(INHERIT, ?, show)    -> {function for member objects}
```

When several guards match an input message, the most specific guard is chosen. For instance, the guard (**request**, *mailer*, *send*) supersedes (**request**, ?, *send*) . Because some guards are incomparable, there may be no most specific guard. In that case, a maximal guard, that is, one that is not less specific than any other guard, is chosen nondeterministically. As an alternative to the most-specific and nondeterministic strategy, we could use the Prolog approach: the programmer would order the guards, and the first guard that matches the message signature would be chosen.

Infinite looping can arise if the chain of inheritance forms a cycle. An algorithm is needed to detect this dynamically. A simple scheme is to record the inheritance path as it is traversed to check for repetitive nodes.

Instead of having a hierarchy of objects as in other object models, our model can be thought of as having a *network* of objects. This obviates the need for such entities as the largest superclass or the class of all classes. It also makes our model simpler and yet more general than most other object models. In particular, because a tree is but one instance of a general graph, the notions of class, superclass and metaclass can be accommodated in this model. See Section 4 for details.

In order to permit multiple inheritance we allow inherited methods to refer to a list of other objects. If such a guard is chosen, an inheritance would be sent to the first object in the list. Should a simple method be returned, then that method would be executed and the inherited method would be complete. Should an error be returned, indicating that no simple method was found, then the next object in the list would be sent an inheritance. If the list is exhausted and no simple method is found, then an error would be returned. Note that our multiple inheritance is at the method level rather than at the (typical) object or class level. This is in keeping with our notion of multi-dimensional inheritance.

It is immediately obvious that this list-based multiple inheritance is only a simple case of a class of generalized inheritance techniques. One could employ a non-deterministic choice of objects to request methods from. More general control structures are also possible based on the state of the local variables of an object, the parameters of the inheritance, or the id of the original object. By identifying multiple inheritance as a particular syntactic object, we have simplified its discussion. Multiple inheritance is now an issue orthogonal to the structure of the network of objects and to the semantics of simple methods and passing messages.

3. Objects as processes

There are several ways of interpreting objects. One interpretation is that objects are records with possible functional components, as in [Ca]. Another interpretation, which seems more natural for our model, is that objects are communicating processes.

A *process* is an independent computing agent that interacts with its environment solely by passing messages through its input ports and output ports [Ho78]. Each process has a private memory that only it can access. Because processes interact with one another solely by passing messages, the way a process is implemented has no effect on other processes. Only the external behavior of a process matters. There has been extensive research on the formal semantics of processes [Ho83, NDGO, P].

In this interpretation, an object is a process having an input port and output port for receiving and sending requests and replies, and an input port and output port for sending and receiving inheritance. In other words, requests, inheritance, and replies are all messages. On the ports for sending and receiving inheritance, *asynchronous message passing* is used. An object can send an inheritance any time, and the message is queued on the inheritance-input port of the receiving object. The protocol for the ports for sending and receiving requests and replies is equivalent to *remote procedure calls*. In this case, after an object sends a request to another object, it cannot resume execution until it receives a reply, because its local state may not be consistent.

By modeling objects and inheritance as communicating processes and messages, we are able to give a simple, formal semantics of objects and inheritance using existing semantics for processes, with minor modifications [NDGO]. Informally, in this approach, a process is specified solely by its set of (external) behaviors. A *behavior* is an infinite sequence of observations, where each *observation* is a record of the history of communication events on the process up to some point in time. Thus a behavior is an abstract representation of the communication history in some execution of the process over time. The set of behaviors of a network of processes can be composed from the behaviors of the component processes of the network.

4. Simulating other systems

4.1 Smalltalk [GR]

In Smalltalk, objects have no methods of their own. An object inherits all of its methods from its class. All objects in a class must have the same set of variable names. Using our model, an object x is a Smalltalk class of some other object y if y has $class = x$, y has no simple methods, and y inherits all methods from x. Furthermore, all members of x must have the same set of private variables. A Smalltalk class x is a subclass of another Smalltalk class y if x inherits methods from only y. Metaclasses are similar.

Classes treat requests from (member) objects with a simple method or through inheritance to the superclass. User requests to a class are inherited from the metaclass of the class. Class variables can be simulated as follows. Associated with each class variable x are two methods *getval*\$$x$ and *putval*\$$x$, which reads and writes on x, respectively. To read (write on) x, an object sends a request to the class and asks it to execute the method *getval*\$$x$ (*putval*\$$x$).

Note that reading the value of a class variable is semantically similar to inheriting a method – consider the inherited variable a nullary function that returns its value. Writing to a class variable is fundamentally different. It involves changing a value in a remote name

space. Access to such a remote space must be serialized with respect to other inheritances and other actions in that space.[2]

```
object:
     (REQUEST,?,?) -> INHERIT FROM class
class:
     (INHERIT,?,m) -> {return method m}
     ...
     (INHERIT,?,?)  -> INHERIT FROM superclass
     (REQUEST,?,?)  -> INHERIT FROM class
metaclass:
     (INHERIT,?,n) -> {return method n}
     ...
     (INHERIT,?,?)  -> INHERIT FROM metaclass
     (REQUEST,?,?)  -> INHERIT FROM "Metaclass"
```

4.2 Smallworld [LH]

Smallworld objects export all of their variables and can have their own methods. We represent the values of the variables as two methods per variable x: *getval$x*, *putval$x* . As in Smalltalk, unknown operations follow the class/superclass path. In our model, an object y is a Smallworld class of some other object z if z inherits methods from only y. Here are some examples of objects and classes in Smallworld.

```
object:
  (REQUEST,?,m) -> {perform method m}
  ...
  (REQUEST,?,?) -> INHERIT FROM class
class:
  (INHERIT,?,n) -> {return inherited method n}
  ...
  (INHERIT,?,?)  -> INHERIT FROM superclass
  (REQUEST,?,k) -> {perform method k}
  ...
  (REQUEST,?,?)  -> INHERIT FROM class
```

5. Conclusions

We have described a generalized model of objects that supports multi-dimensional inheritance. This model is simple yet general. The notions of classes and metaclasses are shown to be definable in terms of the two primitive notions objects and inheritance. Objects are organized into networks rather than hierarchies, thus obviates the need for assuming the existence of a largest superclass and a class of all classes. We also show how objects can be interpreted as communicating processes, thus makes it possible to give a simple formal semantics for objects and inheritance.

We have, in some sense, sacrificed fast algorithms for storing and manipulating hierarchies (trees) in exchange for the increased expressive power of general graphs. We need to

---

[2] The terminology may be confusing here, so we will relate the terms we use to those of Fig 16.1 of [GR]. We use the term "class" to mean the "instance relationship" as designated by dashed lines. We use the term "superclass" to mean the "class hierarchy" as designated by solid grey lines. Finally, we use the term "metaclass" to mean the "metaclass hierarchy" as designated by solid black lines.

show that we can perform aggregating operations quickly in this general setting: for example, "show me all objects that consider x to be their class." Alternatively, we must provide special fast properties, such as class and superclass. These special properties could have compile-time links rather than run-time searches.

We have also noticed an analogy in our work to that of database theory. One can draw a correspondence between Smalltalk (or Smallworld) and hierarchical databases: they both involve entities collected in a tree-structured organization. Our model corresponds closely to the network database model: requests must navigate through a network of information to satisfy a query. Interestingly, Korth [K] is completing the analogy by developing an object theory based on the relational database model.

After this paper was written, we learned about the Emerald system at the University of Washington [BL1, BL2]. Their system does not use our notion of inheritance, but we find a great deal of similarity between our notion of "object" and theirs. In addition, their specification of interfaces is similar to the direction we would like to take in that area. A more extensive comparison between our object model and that of Emerald will appear in our future papers on this topic.

Our future research is to refine our model to reintroduce notions of type-safety of parameters, organization of objects, specification of methods, and verification of correctness. We want to study *dynamic inheritance* in our model: that is, to allow a method to change the inheritance specifications of the methods within an object. We also want to investigate alternative inheritance strategies, where we allow the *inherit from* operation to be just another statement in the method language. This modification would allow a subclass to inherit an operation from a superclass and then customize the answer, or to involve many different inherited operations in resolving a single request.

6. *Acknowledgments*

We would like to thank Rob Strom for the formulation of inheritance as the returning of a code body (in contrast to the conventional notion of remote execution). We also thank Peter Wegner for relating the idea of alternative inheritance strategies and Lee Hoevel for suggesting the notion of dynamic inheritance to us. Thanks go to Harold Ossher and Jan Stone for helpful comments on an earlier draft of the paper.

REFERENCES

[B1] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. To appear in *Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland), ACM, September 1986.

[B2] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. Technical report 86-02-04, University of Washington, Seattle, February 1986. To appear in *IEEE Transaction on Software Engineering*.

[Ca] L. Cardelli. A semantics of multiple inheritance. Unpublished manuscript, AT&T Bell Laboratories, 1985.

[CA] G. Curry and R. Ayers. Experience with traits in the Xerox Star workstation. *IEEE Transactions on Software Engineering* SE-10 (5):519-527, 1984.

[CBLL] G. Curry, L. Baer, D. Lipkie, and B. Lee. Traits: an approach to multiple-inheritance subclassing. *SIGOA Conference on Office Information Systems*, pages 1-9. ACM, 1982.

[DDH] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, 1972.

[D] O. Dahl and K. Nygaard. Simula, an Algol-based simulation language. *Communications of the ACM* 9:671-678, 1966.

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

[Ha] W. Harrison. A program development environment for programming by refinement and reuse. Technical Report RC 11352, IBM Research, Yorktown Heights, NY, 1985.

[HW] J. Hendler and P. Wegner. Viewing object-oriented programming as an enhancement of data-abstraction methodology. *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, volume 2, pages 117-125. Western Periodicals, 1986.

[Ho78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM* 21 (8):666-677, 1978.

[Ho83] C. A. R. Hoare. Notes on communicating sequential processes. Technical Monograph PRG-33, Programming Research Group, Oxford University Computing Laboratory, 1983.

[K] Henry F. Korth. Extending the scope of relational languages. *IEEE Software* 3 (1):19-28, January 1986.

[LH] M. R. Laff and B. Hailpern. SW 2: An object-based programming environment. *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments.* Available as *SIGPLAN Notices* 20 (7):1-11, 1985.

[NDGO] V. Nguyen, A. Demers, D. Gries, and S. Owicki. A model and temporal proof system for networks of processes. *Distributed Computing* 1 (1):7-25, 1986.

[P] V. Pratt. On the composition of processes. *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 213-223. ACM, 1982.

[R] T. Rentsch. Object-oriented programming. *SIGPLAN Notices* 17(9):74-86, 1981.

[SBK] M. Stefik, D. Bobrow, and K. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software* 3(1):10-18, 1986.