Introducing Hybrid: A Unified Object-Oriented System

O.M. Nierstrasz

Institute of Computer Science
Research Centre of Crete

UUCP: mcvax!ariadne!oscar (Crete)
mcvax!cernvax!cui!oscar (Geneva)

ABSTRACT

Hybrid is a data abstraction language that attempts to unify a number of object-oriented concepts into a single, coherent system. In this paper we give an overview of our object model, describe a number of the language constructs, and briefly discuss the issue of object management.

1. Introduction

Hybrid is a programming language that attempts to unify a number of concepts that we consider to be integral to the ''object-oriented'' paradigm. These include:

1. data abstraction with classification, aggregation and specialization
2. atomic events
3. concurrency control à la monitors
4. automatic triggering
5. object persistency
6. location-transparent object addressing

Our goal is to present these concepts in a natural, consistent manner in a small, high-level language. The language should be general-purpose, yet provide the programmer with adequate mechanisms for controlling low-level issues, such as process-switching, concurrency and locking granularity, when these are required. There should be very few assumptions about the data classes and operations supported, so that the language can be truly extendible.

We intend our system to be useful to those who wish to rapidly develop distributed applications such as office forms systems [Tsic82], ''intelligent mail'' systems [Hogg85], and so on. A useful system should have a rich selection of powerful object classes, such as multimedia documents with version control, user interface objects with arbitrary key-binding and window management, role objects for encapsulating security policies, and so on, as outlined in [Nier85b]. Ideally, programming of new applications would often be a simple matter of putting existing objects together in new ways (much as one can put together ''scripts'' of tools in an environment like UNIX†).

In this paper we present an overview of Hybrid, and a brief discussion of some of the more interesting language and implementation issues. A draft language definition is nearing completion, and we will be starting the implementation of an interpreter written in the C programming language [Kern78]. Future generations will be written in Hybrid itself.

In *IEEE Database Engineering*, vol. 8, no. 4, Dec. 1985, pp. 49-57.

† UNIX is a trademark of Bell Laboratories.

2. Overview

The notion which is most fundamental to the design of Hybrid is that of data abstraction. Simply taken, it means that the representation of a data object, and the implementation of the operations it supports should be hidden. The only legitimate interface to the object should be through its operations. Programming languages that support some form of data abstraction are growing in number quite rapidly, and are becoming difficult to enumerate. Some of these are : Simula [Birt73], Smalltalk [Gold83], CLU [Lisk77], Argus [Lisk83], Zetalisp [Wein81], C++ [Stro84], OOPC (Objective C) [Cox83], Galileo [Alba85], Modula [Wirt83], Oz [Nier85a], Taxis [Mylo80], OPAL [Ahls84], Smallworld [Laff85] and BETA [Kris83].

In Hybrid, ultimately everything is an object. The initiative to do things is also encapsulated in objects, consequently we do not have ‘‘objects’’ and ‘‘programs that manipulate objects’’, but just objects, and sometimes these objects attempt to communicate with one another. Naturally, the most active objects take their orders from the outside world (i.e. users, etc.).

An *object* is an *instance* of an *object class*. An object class is the specification of an abstract data type. The specification includes a *contents* portion, which describes the *instance variables* that distinguish individual object instances, and a *behaviour* portion, which describes the code shared by all instances of the same class.

An *integer* object, for example, might contain a single instance variable, namely a 32-bit value. Various instances of this class would each be represented by such a 32-bit value, and all the instances would share the code implementing the operations that integers support.

Instance variables are names for *acquaintances*, other objects that can be communicated with. An object that is properly contained in the contents, i.e. is stored locally, is a *dependent*, or *child*. Children are stored in *dependent variables*. Other acquaintances are known through *reference variables*, which reference more distant objects. Syntactically, there is little difference in the way the two kinds of variables are used. Internally, however, reference variables store *object identifiers*, or *oids*, which the system uses to keep track of objects.

In the case of integers, the 32-bit values would undoubtedly be dependent variables. A *mailbox* object, on the other hand, would probably know the messages it ‘‘contains’’ as external acquaintances.

Any object can thus be seen as an aggregate of its children plus a number of (non-child) acquaintances. Top-level objects (those with no parents) are called *independent objects*. The *descendents* of an object are its children and, recursively, the children of its descendents. A *domain* is a collection containing a single independent object, called the *domain parent*, and all its descendents. Conceptually it is most convenient to think of a domain as being a single object, namely the domain parent. For complex objects, such as documents, however, one may wish to deal with individual components (tables, figures, paragraphs) as objects in their own right.

Across a network, the universe of objects can be partitioned into a number of object *environments*, each of which is a logical node. Each environment has a single *object manager* that performs the duties of an operating system. The environments will normally correspond to physical nodes, but this is not a strict requirement. An object wishing to communicate with an acquaintance in another environment can do so through an *agent*, a local representative of that acquaintance. Objects that actually move from one environment to another leave a local agent behind so that they can be located, if necessary. Oids are globally unique, so there can never be confusion between a local object, and one that has immigrated.

3. Language

An object specification consists of header, contents and behaviour sections. The header contains the class name, a list of parameters (optional) and the name of the superclass whose contents and behaviour are inherited by the class being defined.

The contents portion consists purely of instance variable declarations. These are mapped to the representation of an object instance. In addition to the dependent and reference variables mentioned above, we have *sequence* variables, which are useful for manipulating sequences of references. (Sequences are a language construct rather than an object class. In order to effectively manipulate a sequence, it should be assigned to an appropriate class, such as a *list*, or an *array*.) In the example below, *talktime* is a dependent variable of class *date*, *who* is an acquaintance of class *mailer*, and *msgseq* is a sequence of references to messages:

var talktime : *date*;
ref who : *mailer*;
seq msgseq : *message*;

The behaviour section describes the processes that live in an object instance. All processes are part of some object's behaviour. This is consistent with the idea that there are no external ''programs'' manipulating objects, but only objects talking to each other.

The behaviour of an object consists of a single main procedure (labeled **start**) and a number of procedures and operations. The **start** procedure may split into a number of concurrent blocks. For a given object, each of these blocks is assigned its own process. Typically most of these processes are idle, waiting for requests from other objects. An object whose processes are all idle is normally retired to secondary storage.

Objects communicate by sending messages. A common type of message is a request to perform an operation. Two other message types are used to return from an operation (with an optional return value), and to report an exception (error) in the execution of the operation.

Depending on the situation, most message exchanges resemble procedure calls or remote procedure calls. The difference lies in the possibility of concurrent requests. Normally an object will queue requests as they come in, but a high-priority request may require immediate attention (especially in the case of real-time applications). Within a domain, where there is no real concurrency, internal requests can be handled as straight procedure calls.

The procedures of an object's behaviour are for that object's use alone. The interface available to acquaintances is the set of *operations* supported by that object. An object announces its readiness to perform an operation with the **accept** statement:

accept init ;

causes the running process to block, waiting for an acquaintance to call the *init* operation. A list of operations may be given, or the keyword **any** can be used to state that any operation can be invoked. The **select** statement can be used to specify follow-up actions after performing an operation, or, with the aid of an **else** clause, to prevent the process from blocking:

```
select
        accept opA ;
        /* followup activity for opA */ ;
or
        accept opB ;
        /* followup activity for oB */ ;
else
        /* no requests, so do something else */ ;
end
```

When no **else** clause exists, the **select** blocks. A **select** statement resembles Dijkstra's guarded commands [Dijk75]. The **accept** functions as a guard, resembling the input commands of Hoare's communicating sequential processes [Hoar78]. The **select accept** combination is also used in Ada [Andr83].

Automatic triggering of processes is made possible with the **while await** statement:

while (mbox.empty) **await** (mbox.insert) ;

If the boolean expression after the **while** evaluates to *true*, then the running process blocks until one of the operations following the **await** is performed on the specified object. When notification is received, the expression is re-evaluated. The **while await** statement is atomic, in the sense that the system guarantees that the awaited operations cannot be ''lost'' in between the evaluation of the condition and the actual waiting.

The onus is on the programmer to indicate what operations may affect the condition. This is perhaps a nuisance, but the degree of control has its benefits:

while (x < y) **await** (x.incr, y.decr) ;

is surely preferable to:

while (x < y) **await** ((x,y).**any**) ;

The **while await** statement is important for detecting changes to acquaintances without having to either poll them, or modify their behaviour to perform the necessary notification. It may be used as a guard for a **select**.

Operations with alphabetic names are invoked by indicating an object, the operation, and a sequence of arguments:

mbox.insert(msg);

Operations may also be invoked using non-alphabetic operator names. Operator-overloading is important if one is to be able to program with objects in a concise, natural manner. The scheme used in Hybrid is to declare operators explicitly as **prefix**, **postfix**, or **infix**. The precedence rules dictate that prefix operators always bind more closely than postfix, and postfix more than infix. In addition, there can be one **index** operation, which is invoked by using (square) brackets. Together with the ''dot'' notation for invoking operations with alphabetic names, the precedence is as follows:

prefix >> { postfix, index, dot } >> infix

An expression such as:

*x++ - y[n].total

would be evaluated as:

((*x)++) - ((y[n]).total)

regardless of the semantics of the various operations. What remains is to identify operators appearing in a cascade between two expressions. This can be especially troublesome when an operator has multiple interpretations as, say, both prefix and infix, or perhaps even postfix as well. The rule used here, in lieu of disambiguating parentheses, is to start at the end of the cascade, going backwards, assuming prefix operators up to, but not including, the earliest possible infix. The remaining operators (if any) must be postfix. So:

x ++ += - y

would parse as:

(x ++) += (- y)

As an extreme example, suppose ''@'' has all three interpretations. Then:

x @ @ @ @ @ y

would parse as:

x @ (@ (@ (@ (@ y))))

that is, one infix, and the rest prefix.

Object specifications can be parameterized. This is useful for defining objects that are to serve mainly as ''containers'' for other objects. The class of the contained object is listed as a parameter in the header of the specification:

stack (paramclass : *paramsuper*) : *object*

Elsewhere in the *stack* specification, the *paramclass* parameter can be used as though it were an actual class. The only operations that are allowed, however, on *paramclass* objects, are those inherited from the class *paramsuper*. Similarly, instances of stacks cannot assign a class to *paramclass* that is not a specialization of *paramsuper*. When a variable of class *stack* is declared, the parameter is given as, for example:

var jobstack : *stack* **of** *job*;

Although strong-typing is enforced in Hybrid, there are mechanisms for handling objects whose (precise) class is not known at compile-time. Suppose, for example, that mail messages can serve as containers for arbitrary objects. A clever mail-handler could unpackage certain kinds of messages containing objects of known classes. In the specification of the mail-handler, all one knows for certain is that message contents are of the generic class *object*. The specific class can be determined at run-time by using the **class** statement:

```
class {
thing : document =>
        folder.file(thing);
thing : meeting =>
        calendar.enter(thing);
thing : object =>
        /* default, if others fail */ ;
}
```

The *file* operation requires a *document* argument, but it is known to be invoked only if *thing* is verified (at run-time) to be of that class. Type-checking for the code within the various **class** cases can be done at compile-time.

Assignment in Hybrid is performed as follows:

x ← 5 ;

means, ‘‘the variable *x* is now a name for the object 5’’. This is different from:

x := 5 ;

which means, ‘‘apply the infix operation := to the object named by *x*, and send it the argument 5’’. In the first case, a new object is named; in the second, an existing object is modified. When an object is assigned to a variable, either the object itself is moved, or an oid is created and copied. This depends on whether the variable is a dependent or a reference variable.

Objects can be assigned several at a time:

(x,y) ← circle.centre ;

Furthermore, a sequence of unspecified length may be passed as an argument to an operation, or as a return value from an operation. *Sequence* variables are used to name these object. If *s* is a sequence variable, then:

(x,s) ← s ;

assigns the head of *s* to *x*, and the remainder back to *s*. Generally a more satisfactory solution is to use the sequence to initialize a *list*, or some other suitable container class, and then use the *list* operations to access the elements of the sequence. Alternatively, one may iterate through a sequence with a **for** statement:

```
for x in (s) {
        /* do your stuff */ ;
}
```

Sections of code can be made ‘‘atomic’’ by declaring them as an *event* block. Before entering an event, the states of the objects used within the event, i.e., its *resources*, are sampled and saved. If the event *commits*, the saved states are discarded. If it *aborts*, the saved states are restored. During the event, the intermediate states are not visible to non-participants. Events may be nested, in the fashion described by Moss in his Ph.D. dissertation [Moss81]. A parent event, detecting the failure of a child, may either choose to abort itself, or may attempt some other action.

Events are especially important when negotiating a transaction across a network or updating stable storage. In either case, interruption of the event, due to a crash, for example, could leave objects in inconsistent states. Events can be used to make such transactions atomic.

4. Object management

Objects in Hybrid are persistent. That is, once created, an object continues to exist until it is explicitly destroyed. In order to maintain a consistent view of objects in a given environment, we distinguish between stable and volatile storage. Stable storage contains a complete, consistent representation of all objects in the environment at some point in time. Volatile storage contains ''working copies'' of currently active objects. In the event of a crash, the contents of volatile storage (i.e. virtual memory) is discarded. Stable storage must therefore be updated very carefully, incrementally creating new, consistent views of the environment. Stable storage can be thought of as the permanent object database.

Depending on the requirements of the applications, the frequency with which stable storage is updated can vary. This means that many objects can be created and destroyed in between such updates, so that these objects are never written to stable storage. Where an atomic event is involved, it is, of course, important to update stable storage in a fashion that preserves the integrity of the event. For example, an event that spans several object environments, and several physical nodes, must, if it commits, have its effects observed reliably at all the nodes. Updating the stable storage of these several environments must also take place as a single, atomic event. Two-phase locking is typically used to implement this sort of behaviour [Moss81, Verh78].

The object manager is also responsible for bringing needed objects into memory and resolving object identifiers to actual memory addresses. Object identifiers can be thought of as capabilities for addressing objects [Fabr74]. (Oids are actually capabilities for sending messages, i.e. for performing operations. One thus distinguishes not merely between, say, read and write operations, but between all of the various operations supported by an object. This is the same philosophy adopted in the Hydra operating system [Wulf74].)

A hash table is maintained for the objects currently in memory. The first time an active object attempts to address an acquaintance, the object manager does a lookup to see if the required object is already in memory. If not, it must be brought in. That acquaintance is then marked as being ''open'' for communication, and the oid is translated into a memory address. As long as the connection stays open, no further lookups are needed. Connections can be closed when an object becomes inactive, i.e. when there are no more outstanding messages for it, and all its processes are blocked. Inactive objects may be retired to stable storage. In fact, objects that have been inactive for only a brief period may quite reasonably become active again in a short time, so it makes sense only to retire objects that have been inactive for a while. (This is analogous to paging policies in operating systems.)

Part of the object manager's task is to keep track of the correspondence between the representations of instances, and the code that implements their behaviour. Since the latter is shared, care must be taken when modifications are made the specification of an object class. In fact, a fair degree of intelligence should be built into the *class* meta-class. In particular, some sort of version-control is required to protect existing object instances when new versions of classes are installed. Furthermore, some degree of cleverness is needed to handle objects that move from one environment to another, since the class definitions may not be identical.

The object manager must negotiate concurrent accesses to objects. The problem is somewhat simplified by insisting that, for any given domain, there is never more than one process active. Domains are, in this respect, similar to monitors [Hoar74]. The arrival of a message (a request for service) interrupts the domain, which then decides whether it can handle the request immediately or not. Since the process issuing a request blocks if its request is not handled immediately, there is always a possibility of deadlock. When an object is participating in an atomic event, it is, strictly speaking, not permitted to handle any external requests, since the intermediate states of the event must not be visible. If deadlock occurs between several concurrent events, a ''victim'' must be chosen, and restarted. There is a substantial body of literature on this problem, and many schemes exist for handling it [Bern81, Kohl81].

A final issue of interest is that of garbage collection. The approach taken in Hybrid is that objects themselves are ultimately responsible for their own existence. A request to an object for it to commit suicide can well be ignored by that object, if it decides that the time is simply not right. Similarly, it is not up to the object manager to decide when objects are ''garbage''. On the other hand, if a passive object is no longer referenced by any other object (except by system objects, such as the object manager), then this is probably a good hint that the object has become garbage. The object manager therefore maintains a reference count for objects, and when that count drops to zero, the object in question is notified. It may then decide whether to commit suicide, or possibly initiate some other action.

5. Conclusions

Hybrid is a system for programming with abstract data types. There is a uniform view of all objects, so we do not distinguish between ‘‘objects’’ and ‘‘programs’’ -- instead, every process is an integral part of the behaviour of an object. An idle object can be activated by sending it a message, i.e. invoking an operation. Objects can also be automatically triggered into action by having them wait for a precondition with a **while await** statement.

Objects are persistent, meaning that the system automatically saves consistent states of the object environment. Objects that are idle are normally kept in the stable storage object database. They are brought into memory only when they are activated by a message or a trigger condition.

Objects are referenced by unique object identifiers. The complete object universe consists of many object environments, each with its own object manager, communicating over a network. To protect the integrity of objects taking part in inter-node transactions (where failure of some component of the network is possible), atomic events are available as a programming construct. Events are useful for preventing inconsistent states of the object environment from being seen by objects contending for the same resources.

A draft language specification is nearly complete. A prototype implementation written in C will be starting shortly.

6. References

[Ahls84] M. Ahlsen, A. Bjornerstedt, S. Britts, C. Hulten and L. Soderlund, ‘‘An Architecture for Object Management in OIS’’, ACM TOOIS, vol. 2, no. 3, pp. 173-196, July 1984.

[Alba85] A. Albano, L. Cardelli and R. Orsini, ‘‘Galileo: A Strongly-Typed, Interactive Conceptual Language’’, ACM TODS, vol. 10, no. 2, pp. 230-260, June 1985.

[Andr83] G.R. Andrews and F.B. Schneider, ‘‘Concepts and Notations for Concurrent Programming’’, ACM Computing Surveys, vol. 15, no. 1, pp. 3-43, March 1983.

[Bern81] P.A. Bernstein and N. Goodman, ‘‘Concurrency Control in Distributed Database Systems’’, ACM Computing Surveys, vol. 13, no. 2, pp. 185-221, June 1981.

[Birt73] G. Birtwistle, O. Dahl, B. Myhrtag and K. Nygaard, *Simula Begin*, Auerbach Press, Philadelphia, 1973.

[Cox83] B.J. Cox, ‘‘The Object Oriented Pre-Compiler’’, SIGPLAN Notices, vol. 18, no. 1, pp. 15-22, January 1983.

[Dijk75] E.W. Dijkstra, ‘‘Guarded commands, nondeterminacy, and formal derivation of programs’’, CACM, vol. 18, no. 8, pp. 453-457, Aug 1975.

[Fabr74] R.S. Fabry, ‘‘Capability-Based Addressing’’, CACM, vol. 17, no. 7, pp. 403-412, July 1974.

[Gold83] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, May 1983.

[Hoar74] C.A.R. Hoare, ‘‘Monitors: An Operating System Structuring Concept’’, CACM, vol. 17, no. 10, pp. 549-557, Oct 1974.

[Hoar78] C.A.R. Hoare, ‘‘Communicating Sequential Processes’’, CACM, vol. 21, no. 8, pp. 666-677, August 1978.

[Hogg85] J. Hogg, ‘‘Intelligent Message Systems’’, in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, pp. 113-134, Springer Verlag, Heidelberg, 1985.

[Kern78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall Software Series, 1978.

[Kohl81] W.H. Kohler, ‘‘A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems’’, ACM Computing Surveys, vol. 13, no. 2, pp. 149-183, June 1981.

[Kris83] B.B. Kristensen, O.L. Madsen, B. Moeller-Pedersen and K. Nygaard;, ‘‘Abstraction Mechanisms in the BETA Programming Language’’, 10th ACM symposium on the principles of programming languages, Norwegian Computing Center, Oslo, 1983.

[Laff85] M.R. Laff and B. Hailpern, ‘‘SW 2 -- An Object-based Programming Environment’’, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1985.

[Lisk77] B. Liskov, A. Snyder, R. Atkinson and C. Schaffert, ‘‘Abstraction Mechanisms in CLU’’, CACM, vol. 20, no. 8, pp. 564-576, August 1977.

[Lisk83] B. Liskov and R. Scheifler, ‘‘Guardians and Actions: Linguistic Support for Robust, Distributed Programs’’, ACM TOPLAS, vol. 5, no. 3, pp. 381-404, July 1983.

[Moss81] J. Eliot B. Moss, ‘‘Nested Transactions: An Approach to Reliable Distributed Computing’’, Ph.D. thesis, MIT/LCS/TR-260, MIT Dept EE and CS, April 1981.

[Mylo80] J. Mylopoulos, P.A. Bernstein and H.K.T. Wong, ‘‘TAXIS: A Language Facility for Designing Database-Intensive Applications’’, ACM TODS, vol. 5, no. 2, pp. 185-207, June 1980.

[Nier85a] O.M. Nierstrasz, ‘‘An Object-Oriented System’’, in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, pp. 167-190, Springer Verlag, Heidelberg, 1985.

[Nier85b] O.M. Nierstrasz and D.C. Tsichritzis, ‘‘An Object-Oriented Environment for OIS Applications’’, Proceedings, Conference on Very Large Data Bases, pp. 335-345, Stockholm, August 1985.

[Stro84] B. Stroustrup, ‘‘Data Abstraction in C’’, Computing Science Technical Report #109, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, January 1984.

[Tsic82] D.C. Tsichritzis, F. Rabitti, S.J. Gibbs, O.M. Nierstrasz and J. Hogg, ‘‘A System for Managing Structured Messages’’, IEEE Transactions on Communications, vol. Com-30, no. 1, pp. 66-73, January 1982.

[Verh78] J.S.M. Verhofstad, ‘‘Recovery Techniques for Database Systems’’, ACM Computing Surveys, vol. 10, no. 2, pp. 167-195, June 1978.

[Wein81] D. Weinreb and D. Moon, *The Lisp Machine Manual*, Symbolics Inc., 1981.

[Wirt83] N. Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin, 1983.

[Wulf74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack, ‘‘HYDRA: The Kernel of a Multiprocessor Operating System’’, CACM, vol. 17, no. 6, pp. 337-345, June 1974.