A Tour of Hybrid[1]

O.M. Nierstrasz

University of Geneva
Centre Universitaire d'Informatique
12 Rue du Lac, CH-1207 Geneva, Switzerland
E-mail: oscar@cui.unige.ch, oscar@cgeuge51.bitnet
mcvax!cernvax!cui!oscar

Abstract

Hybrid is a strongly-typed, concurrent, object-oriented programming language in which objects are active entities. In this paper we provide an overview of the language constructs, paying particular attention to the mechanisms for programming concurrent applications, and we describe our experiences in developing a prototype implementation of the language and its run-time environment.

1 Introduction

Hybrid is an object-oriented programming language that addresses the development of applications concerned with concurrency, distribution and code reusability. The design of the language is an attempt to integrate three distinct notions of *objects*:

1. objects as instances of reusable *object classes*
2. objects as *typed entities*
3. objects as independent *active entities*

The main innovation in the language is the concurrency model, which provides for a uniform message-passing paradigm for communication between active objects consistent with strong-typing, and supports mechanisms for creating, interleaving and scheduling threads of control.

This paper gives an overview of the language, with particular attention to object types and active objects. A prototype implementation of a Hybrid compiler and run-time environment for active objects is also outlined.

2 Object Types

An application written in Hybrid consists of a collection of cooperating active objects, possibly distributed amongst a number of separate object environments. An object may be created either as an independent "top-level" object, called a *domain*, or it may be a dependent "part" (instance

[1]In *Actes des mardis objets du CRIN*, CRIN 89-R-072, G. Masini, A. Napoli, D. Colnet, D. Lonard and K. Tombre (Eds.), pp. 237-248, Centre de Recherche en Informatique de Nancy, Vandoeuvre-lès-Nancy, 1989.

variable) of another object within a domain. Domains define the granularity of concurrency, and so are comparable to monitors [Hoare 1974]. (The concurrency model is further described in section 3.)

Every object in Hybrid is an instance of an object type. Type definitions have the following general form:

```
type typeName parameters :
    typeSpec ;
private
    realization
```

The *typeSpec* describes the public interface to instances of the type. The interface is typically a set of operations that may be invoked (including the specification of argument and return types), but may also include "visible instance variables". In addition to named operations, one may define and overload a set of infix and prefix operators recognized by the language.

The *realization* part of a type definition describes the private instance variables, the implementation of the operations (i.e., the methods), and any private operations. Type parameters defined in the interface may be used anywhere in the specification or realization as though they were bound to actual types.

2.1 Type specifications

Type specifications are described using a number of type constructors, the most general of which is `abstract`, which requires the programmer to explicitly provide the list of public operations and visible instance variables associated with the type. For example,

```
type buffer of itemType :
        abstract {
                put : itemType -> ;
                get : -> itemType ;
        } ;
```

defines the interface to a type `buffer` supporting the operations `put` and `get`. (The realization is not shown.) The argument type of `put` and the return type of `get` are that of the parameter `itemType`. Another object that requires an instance of a buffer must bind the parameter to an actual type (i.e., one with a realization).

Abstract types may also define a set of *infix* and *prefix* operators, and may overload the *indexing* operator denoted by square brackets (`[]`). Operators are constructed from a fixed alphabet of operator symbols. The language distinguishes between *priority* operators (namely `*`, `/` and `%`), *relational* operators terminating in a question mark (`?`) and yielding a Boolean value, *assignment* operators terminating in an equal sign (`=`), and parsed right-to-left, and all other operators, parsed left-to-right.

The other constructors are `inherits`, for defining subtypes that inherit operations from multiple parents, `enum` for defining enumerated types, `oid` for defining object identifiers, `array` for defining

homogeneous arrays, and `record` and `variant` for defining records and variant types. A type may also be defined as a range of integer values (ranges of values from enumerated types are not yet handled).

For all of the type constructors except `abstract`, the realization is typically omitted, since it can be inferred from the *typeSpec*. For example, the realization of an `array` is automatically supplied by the compiler. In the case of the `inherits` constructor, the methods and instance variables inherited from parents may be overridden by the subtypes.

One may also define abstract types with incomplete or empty realizations, but these types (called *virtual* types) cannot be instantiated. A subtype inheriting from a virtual type is also virtual, unless it supplies the missing methods in its realization.

The interface to an object is its *effective* type. The *actual* type of an object is determined by its realization. A type T_1 *conforms* to another type T_2 if supports at least the same interface, i.e., if it supports at least the same set of operations with the same specifications. We say that T_1 is a *subtype* of T_2, even if it does not inherit anything from T_2. Inheritance is therefore purely a code reusability mechanism in Hybrid, and only accidentally establishes a subtype relationship.

Effective types and subtypes are used to determine whether expressions are type-correct. With dynamic binding, actual types may not be known till run-time.

2.2 Expressions

Expressions have the general form:

<*target*> <*operation*> <*arguments*>

The target and arguments may themselves be subexpressions. The actual form of the expression may vary, depending on whether the operation is named by an identifier, or by one of the infix or prefix operators. The former looks like:

```
b.put(value)
```

whereas the latter may as complicated as:

```
n := a * ++b + c
```

which would be parsed as:

```
n := ((a * (++b)) + c)
```

Note that variable binding is different from assignment. Assignment operators are defined by the methods of an object type, whereas the binding operator (`<-`) binds variable names to values. For example, in the expression:

```
a := b <- c
```

the name `b` will be bound to a copy of the object instance currently named by `c`. Then the instance named by `a` will execute a method corresponding to the operator `:=` with the argument named by `b`. Presumably (but not necessarily), `a` will try to make itself look like `b`. In this example, `b` is dynamically re-bound, whereas `a` is not.

Expressions are *type-correct* if operation invocations are consistent with the effective types of the target and argument subexpressions. Variables may be dynamically bound to the value of any expression that conforms to the declared type of the variable.

Type casting is required to change the effective type of an expression to a more general type. For example, consider:

```
scratchPad.insert(s:graphicalObject)
```

where `s` is a variable of type `spline`, and the `insert` operation of the `scratchPad` expects a `graphicalObject` argument. Then type-casting will tell the compiler to verify that `spline` is a subtype of `graphicalObject`.

In the implementation, this step also guarantees that the appropriate method lookup table will exist so that the type-cast object can efficiently respond to messages intended for objects of the type it conforms to. Once type-casting has been performed, there is only a small, fixed overhead in looking up the method for, say, a *display* operation.

2.3 Statements

Statements, unlike expressions, have no type or value. A simple statement consists of an expression followed by a semi-colon. Compound statements are a series of statements enclosed in braces (`{ ... }`), and may include local (automatic) variable declarations. An example of a declaration is:

```
var s : spline ;
```

Hybrid has an `if` statement and a `switch` statement for selectively executing code. Repetition is provided by a `loop` statement, which may be repeated with a `continue` statement, or exited with a `break` statement. A `block` is similar to a `loop`, except that it can only be exited, not repeated. In case of nested loops or blocks, a label may be supplied to either `break` or `continue`.

Hybrid also supports a `check` statement for disambiguating variant types at run-time, and for determining whether an object actually belongs to a subtype of its effective type. For example:

```
var x : graphicalObject ;
...
check (g :? spline) {
        ...
}
else {
        # complain ..
}
```

will determine if the actual type of the current value bound to `g` conforms to the more specific type `spline`. Upon success of the check statement, `g` will be re-declared to be of type `spline` for the body of the compound statement that follows.

The `return` statement is used to terminate a method. The expression supplied to it must conform to the method's declared return value. The `end` statement terminates a thread of control, and may only occur within the method of a *reflex* (see below).

A more detailed description of Hybrid exists in [Nierstrasz 1987a].

3 Communication and Concurrency

Objects are *active* when they are responding to a message. Since all objects are instances of object types, this means that objects are active when they are responding to an operation invocation, or when they themselves receive a response to request they have issued.

The basic model of communication is that of remote procedure calls. Messages between objects are generally either *call* messages, requesting an object to execute one of its methods, or *return* messages sent after the successful completion of a method. (Exceptions were envisaged as a necessary alternative to *return* messages, but were not included in the initial language design.) We can therefore trace a thread of control, called an *activity*, as a sequence of *call* and *return* messages between objects, whether they communicate within a domain or between domains.

New activities are created by invoking a special kind of operation called a *reflex*. When a reflex is invoked, a *start* message is sent to the object, and accepted as soon as the object's domain is idle. Since reflexes do not return anything, the effect is to initiate a new activity. The method for a reflex is terminated by an `end` statement.

Messages may be delivered either synchronously, when communication is between objects within the same domain, or asynchronously, when communicating objects are independent. A call to a remote object is made through an *object identifier* (i.e., of type `oid`), which takes care of delivering the message. When an object sends a *call* message to a remote target, the object's domain ordinarily *blocks* until a response is received. (Recursive calls, related to the blocking activity, are permitted.) An activity can always be viewed as being at a unique location, either within an object executing a method, or buffered in a message queue. Similarly, domains can always be viewed as being in one of three states: *idle*, *running*, or *blocked*.

Two additional mechanisms are required in order to be able to program interesting active objects. *Delay queues* are used to schedule activities when there are operations that cannot always be immediately performed. A simple example is a *get* from an empty buffer. These operations are declared as *using* a named delay queue, and the object manages the queue of buffered messages by *opening* and *closing* the queue during the execution of other methods. The delay queue is typically used to represent either the availability of a resource, or the status of an awaited condition, much in the same way that condition variables are used in monitors. The main difference is that opening or closing a delay queue does not entail an immediate transfer of control, as is the case with *waits* and *signals* [Hoare 1974].

Delegation is a mechanism for interleaving activities. An expression of the form:

`delegate` (*target op args*)

```
type item : abstract { ...  }
private {
var n : integer ; # = no. of items in stock

order: (r: integer) -> integer ;
        uses avail ; # open iff n>0
{
        if (r <? n) {
                n -= r ; # fill the order
        }
        else {
                r := n ; # fill as much as we can
                n := 0 ;
                avail.close() ; # delay future orders
        }
        return(r) ;
}

add: (s: integer) -> ;
{
        n += s ;
        avail.open() ; # assumes s>0
}

} # end of item
```

Figure 1: resource management using a delay queue.

will always be evaluated by asynchronous message-passing, and will leave the calling domain *idle*, that is, free to accept messages related to other activities. The context of the delegated expression is saved, and later resumed when the *return* message is eventually received. Delegation is typically needed for objects that manage multiple activities, such as an "administrator" object that forwards tasks to a set of "worker" objects. Aside from interleaving of activities, delegated expressions behave just like non-delegated expressions.

In figure 1 we see how to schedule requests for a resource by using a delay queue to represent the precondition for service. An `item` object keeps track of the number of items of a certain kind that are in stock. It will service orders as long as there are at least some items in stock, even though it may not be able to completely fill an order. (A "filled" order has at least one allocated item.) Whenever an item is out of stock, requests will be delayed.

Figure 2 shows part of the definition of a `clerk` object that looks up item names, and forwards orders to the appropriate `item` object. Since `clerk` objects may process several orders concurrently, and should not be blocked if an item happens to be out of stock, the order request is forwarded by delegation. The context of the current activity is saved at the point where delegation occurs, and is resumed when the order is filled. Only when the return message is received will a value be

```
type clerk : abstract { ...  }
private {
var item_list : list [string] of item ; # lookup table
process_orders : (f: order_form) -> ;
{
        ...
        # order, but don't block:
        filled := delegate(item_list[item_name].order(r)) ;
        ...
}
} # end of clerk
```

Figure 2: administration by delegation.

assigned to the variable `filled`.

Note that it is also possible to design an `item` object that will only return completely filled orders by introducing a `backorder` object that waits for the number of items required for the current back order. When the `item` object detects that it cannot completely fill an order, it delays all future requests (by closing its `avail` queue), tells the `backorder` object how many items to wait for, and delegates the current request to the `backorder` object, notifying it whenever new items arrive.

The operational semantics of delegation and delay queues is discussed in [Nierstrasz 1987b]. Other examples are given in [Nierstrasz 1987c].

4 Implementation

The Hybrid execution model is that of a distributed collection of object environments, each of which provides support for persistent active objects and for communication between objects in different environments. The prototype implementation is currently restricted to a single object environment, but with support for multiple users.

The Hybrid object manager effectively functions as an "object server" for users' client processes. In the sample applications implemented using the prototype, the user processes are responsible for connecting to object manager, and for managing the user interaction objects (e.g., windows). Objects in the client's environment have corresponding "shadow" objects in the Hybrid object environment, which forward messages to the client.

The object manager is implemented as a single UNIX[2]process that manages the workspace of active objects. Persistence is provided by storing the workspace in a file. The workspace is therefore limited by the size of virtual memory. Pseudo-concurrency is provided by light-weight processes implemented using a coroutine extension to the C language.

The system consists of three main components, the Hybrid *compiler*, the *type manager*, and the *run-time system*. After considering the alternatives, it appeared that the fastest and most

[2]UNIX is a trademark of AT&T Bell Laboratories.

flexible way to implement the compiler was to use the C programming language as a high-level "assembler". Dynamic linking was not considered a high-priority item for the prototype, so the present implementation does not integrate the Hybrid compiler into the object manager. We therefore distinguish between the compile-time and run-time views of the system.

Hybrid type definitions are translated to C, compiled into run-time libraries, and linked in with the object manager. The type manager keeps track of a database of all information concerning object types, other than the actual executable code for the methods. The type database is stored directly in the persistent workspace. The type manager provides the mechanisms for the realization of multiple inheritance, code resubality, type parameterization, overloading and version management. The compiler communicates with the type manager in order to verify type-correctness of new type definitions, and generates information concerning new types to be stored in the type database for later use.

The system implements Hybrid activities as light-weight processes, and domains as shared, passive monitor-like objects. Since the target environment of the prototype was basically a shared memory with pseudo-concurrent processes, this approach was more natural (and efficient) than trying to directly simulate message-passing. The message-passing semantics of Hybrid's concurrency constructs are nevertheless preserved. In order to extend this approach to work in a distributed environment, we would require several light-weight processes to implement a Hybrid activity (i.e., one per environment involved in a computation).

The run-time system mediates between active objects and the client processes. Communication with clients is supported by providing special object types that know how to communicate with the outside world. These types, as well as all basic Hybrid types, exist in the run-time type library. The type manager is responsible for the method lookup tables needed to support dynamic binding, and for the information needed to create and delete objects.

A skeleton parser (i.e., recognizer and pretty-printer) and the routines for managing the persistent workspace were implemented by Oscar Nierstrasz. The Hybrid compiler and the type manager were implemented by Dimitri Konstantas. The run-time system was implemented by Michalis Papathomas. The total implementation effort comprised roughly two man-years over the period from March 1987 to May 1988.

The source code lines of the major components of the Hybrid prototype are of the following sizes:

Compiler	18,102 lines
Type Manager	10,016 lines
Thread Manager	5,497 lines
Basic User Interface	5,426 lines
Run-time Type Manager Interface	1,882 lines
Persistent Workspace Module	1,969 lines

In addition, there were two smaller components dealing with user interface and initialization that were needed for the test applications. The total size of the source code is 44,927 lines of C code.

A detailed report on the implementation can be found in [Konstantas, et al. 1988].

5 Conclusions

Hybrid was conceived as an experiment in integrating several existing object models in order to provide the benefits of object-oriented reusability mechanisms, strong-typing, and constructs for manipulating active objects within a single language.

Despite our efforts to keep the language simple and small, the project turned out to require a great deal more effort than originally expected. Part of the problem was due to the lack of a formal semantics for Hybrid. A number of constructs that were thought to be orthogonal turned out to interfere with one another. For example, delegation may interfere with dynamic binding, since interleaving activities are free to execute methods that will re-bind instance variables participating in other activities. The other main difficulty was due to the original implementation strategy, which was to implement Hybrid in terms of an abstract machine. This turned out to be an impractical approach for a prototype, because it would require designing and thoroughly debugging the abstract machine before we could attack the main problem of implementing Hybrid itself. Translating to C proved to be a far more efficient strategy.

Some of our problems would have gone away if more time had been spent on the language design, but there were many issues that would not have been uncovered without an early prototype. The feedback from the implementation effort will be invaluable for ironing out language design problems, and will also provide a forum for testing whether our ideas were valid or not. One of the surprises was that the reusability mechanisms, the strong-typing, and the concurrency constructs were not as orthogonal as was originally thought. Since their semantics were defined independently, the interference was not discovered until the implementation had started.

The overwhelming lesson from the point of view of language design was that exception-handling must be a fundamental part of the language if we are to make sense of strong-typing in a distributed environment of active objects. Time and again, when semantic difficulties were encountered, it was apparent that exceptions were needed to notify objects when unexpected events occurred during execution, and that the exceptions that could be raised must be part of an object's interface.

Future work will be in several directions. First, the prototype needs to be extended to handle distributed object environments. Aside from handling communication between remote objects, which should not entail any great difficulties, there is a problem guaranteeing global persistence. If remote object environments fail, there is a possibility of messages being lost, and activities being "broken". Furthermore, there may be some difficulties in distributed type management, since objects passed between environments must be able to carry their methods with them.

Second, the language needs some re-thinking and re-design. Exception handling needs to be introduced. A more flexible approach to type-casting will accommodate objects with multiple views, and will eliminate the need for `variant` types and `check` statements, by allowing programmers to define procedures for expanding the interface to a type. The present approach to dynamic binding is far too primitive and costly. An alternative approach that seems promising is to allow the programmer to identify which variables may be dynamically bound. This is to be contrasted with Simula [Birtwistle et al. 1973], in which dynamic binding is restricted to "virtual functions", rather than to the variables themselves.

Third, we are looking at environments and tools that can help programmers design reusable objects, and help them find reusable objects relevant to an application being built [Arapis 1988; Pintado 1988].

Finally, we are looking at formal models and tools that can help in the design and specification of new languages like Hybrid [Nierstrasz 1988].

References

[Arapis and Kappel 1988] C. Arapis and G. Kappel, "An Object Software Base", in *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1988.

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

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

[Konstantas, et al. 1988] D. Konstantas, O.M. Nierstrasz and M. Papathomas, "An Implementation of Hybrid, a Concurrent Object-Oriented Language", in *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1988.

[Nierstrasz 1987a] O.M. Nierstrasz, "Hybrid – A Language for Programming with Active Objects", in *Objects and Things*, ed. D.C. Tsichritzis, pp. 15-42, Centre Universitaire d'Informatique, University of Geneva, March 1987.

[Nierstrasz 1987b] O.M. Nierstrasz, "Triggering Active Objects", in *Objects and Things*, ed. D.C. Tsichritzis, pp. 43-78, Centre Universitaire d'Informatique, University of Geneva, March 1987.

[Nierstrasz 1987c] O.M. Nierstrasz, "Active Objects in Hybrid", ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 243-253, Dec 1987.

[Nierstrasz 1988] O. Nierstrasz, "Mapping Object Descriptions to Behaviours", in *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1988.

[Pintado and Tsichritzis 1988] X. Pintado and D. Tsichritzis, "An Affinity Browser", in *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1988.