A Tour of Hybrid

A Language for Programming with Active Objects*

Oscar Nierstrasz†

Abstract

Object-oriented programming is a powerful paradigm for organizing software into reusable components. There have been several attempts to adapt and extend this paradigm to the programming of concurrent and distributed applications. Hybrid is a language whose design attempts to retain multiple inheritance, genericity and strong-typing, and incorporate a notion of active objects. Objects in Hybrid are potentially active entities that communicate with one another through a message-passing protocol loosely based on remote procedure calls. Non-blocking calls and *delay queues* are the two basic mechanisms for interleaving and scheduling activities. A prototype implementation of a compiler and run-time system for Hybrid have been completed. We shall review aspects of the language design and attempt to evaluate its shortcomings. We conclude with a list of requirements that we pose as a challenge for the design of future concurrent object-oriented languages.

*.In *Advances in Object-Oriented Software Engineering*, ed. D. Mandrioli and B. Meyer, Prentice-Hall, 1992, pp. 167-182.

†.*Author's current address:* Institut für Informatik und angewandte Mathematik (IAM), University of Berne , Länggassstrasse 51, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.4618. *E-mail:* oscar@iam.unibe.ch. *WWW:* http://www.iam.unibe.ch/~oscar.

1 Introduction

An increasing number of today's software systems can be best described as "open systems," that is, systems that evolve to keep up with changing requirements and available technology. Open systems are frequently physically distributed, and may run on a heterogeneous collection of machines. Object-oriented programming is one of very few approaches that shows promise as a better way to develop open systems. There are two important reasons for this:

- Encapsulation of data and operations into software "objects" improves the maintainability of complex systems by decomposing them into manageable pieces with well-defined functionality. Object-oriented programming can thus be seen as a logical continuation of structured and modular programming.
- Instantiation, class inheritance and genericity enhance software reusability and encourage the careful design of truly reusable object classes. Object-oriented programming in combination with good object design can therefore improve the adaptability of open systems by shifting the emphasis from reprogramming to reuse and reconfiguration.

We claim that object-oriented programming can help us not only to cope with the complexity and evolution of open systems, but that it can also serve as a good paradigm for the programming of distributed and concurrent open systems. The key idea is that an object can be viewed as an entity that provides its clients with a service; when that object is servicing a request, it may do so in parallel with other activities taking place in the system. Such an object is said to be "active." Active objects may be physically distributed, their precise location perhaps unknown to clients.

Hybrid is an experimental object-oriented programming language that attempts 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.

In this paper we will discuss the problem of extending the object-oriented paradigm to accommodate active objects. We will provide an overview of Hybrid, paying particular attention to its concurrency model. We conclude with an evaluation of the language, a list of requirements for concurrent object-oriented language design, and topics for further research.

2 Objects and Threads

The minimal requirements for a programming language to be accepted as being "object-oriented" are that the language must support objects, classes and inheritance [16] where:

1. an *object* has a hidden representation (typically a set of *instance variables*) and a visible interface (typically a set of operations, whose implementation – or *methods* – may vary from object to object),
2. an *object class* defines the shared behaviour (i.e., instance variables and methods) of a set of objects, and
3. *class inheritance* can be used to define new classes (*subclasses*) that inherit the behaviour of existing classes, and may augment it with new instance variables and methods.

Modern object-oriented languages often add to this basic list by providing some combination of strong-typing, multiple inheritance, genericity (i.e., parameterized classes), concurrency control mechanisms, and persistent objects.

At first glance, incorporating concurrency into an object-oriented language seems straightforward, after all, objects communicate by "message-passing." However, objects are traditionally viewed as being passive entities with an operational interface, whereas communicating processes are seen as active entities with a stream or message-oriented interface. In fact, if we look at existing concurrency mechanisms as they might apply to objects, we see that they can be classified into two world views:

1. Active entities (processes) share and manipulate passive objects.
2. Active objects communicate and synchronize by message-passing.

This distinction is analogous to that made by Andrews and Schneider [2] who classify approaches as being either *procedure-oriented* or *message-oriented*.

We propose that these approaches be unified by thinking instead in terms of *objects and threads*. Objects are "real" in the sense that they have state and behaviour, whereas threads are virtual, being simply execution traces. Any object is "active" whenever a thread enters it. In the first scenario above, passive objects become active when processes (threads) access them. In the second scenario, objects create or pass on threads by sending messages, and objects become active when they accept messages. If an object becomes idle whenever it sends a message, then the threads correspond exactly to the message-passing traces.

If we compare various concurrency control mechanisms with respect to this reference model of objects and threads, we can gain some insight into the real differences between them. Criteria for comparison include:

- the granularity of the objects,
- the number of threads that may be simultaneously active within an object (single thread objects are "atomic"),
- how a thread is defined,
- how threads are created and destroyed,
- how long threads may be,
- how threads are synchronized (how does an object decide which thread may enter it).

It should be emphasized that what exactly a thread is will be open to interpretation for any given language model. For example, in actor systems [1], threads are arguably the length of a single message-passing event. In real actor applications, however, it is undoubtedly more useful to conceptually group chains of events into longer threads. (After all, this is what the programmer does!)

We further propose that the notion of a thread is extremely important as a programming concept for structuring concurrent computations, and that concurrency control mechanisms should be built upon that notion. (As a disclaimer, we acknowledge that this may not hold for specialized applications where massive parallelism is required.) In the case of object-oriented programming, where the paradigm of an object as "server" accessible by message passing prevails, a thread corresponds by default to a "remote procedure call" trace (i.e., a balanced sequence of *call* and *return* messages).

With Hybrid, we chose to adopt message passing as the paradigm for communication (whether or not "real" message passing takes place in the implementation). Single-thread "atomic" objects are called *domains*, and may be defined by the programmer to be as large or as small as desired. Thread creation is by invocation of a special *reflex* operation, which determines the resulting chain of *call* and *return* messages. A thread is always either present in an executing object, or frozen in a message. An object becomes active by accepting a message. Synchronisation and interleaving of threads is accomplished by the *delay queue* and *delegation* mechanisms described in §4.

We shall now at Hybrid in some detail before returning at the end of this paper to some of the requirements for incorporating concurrency mechanisms into an object-oriented language.

3 An Overview of Hybrid

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 variable) of another object within a domain. Domains define the granularity of concurrency, and so are comparable to monitors [3]. (The concurrency model is further described in §4.)

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.

3.1 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 generic 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 presently supported).

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 *T1 conforms* to another type *T2* if it 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 *T1* is a *subtype* of *T2,* even if it does not inherit anything from *T2*. 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.

3.2 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 be 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 rebound, 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.

3.3 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.

Hybrid has both 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 [8].

4 Communication and Concurrency in Hybrid

Objects are *active* while they are responding to a message. Since all objects are instances of object types, this means that objects are active when 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* [3].

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

    delegate(*target op args)*

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.

```
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.

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 assigned to the variable filled.

```
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.

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 are discussed in [9]. Other examples are given in [10].

5 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 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 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 reusability, 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 Michael 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 [5].

6 Observations

Although the Hybrid project has thus far demonstrated that an object-oriented approach to concurrency is both viable and implementable, we also feel, however, that there are several problems to be solved before we can arrive at a realistic concurrent object-oriented language that will be appropriate for programming open systems.

The first problem we encountered was the lack of useful formalisms for defining the semantics of a concurrent object-oriented language. The semantics of Hybrid's concurrency mechanisms were defined semi-formally, using an ad hoc model, independently of type model and other aspects of the language. The net effect was that interference between supposedly orthogonal mechanisms was discovered rather late in the game. 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. These problems are reported in [14] and [15]. Interference between concurrency mechanisms and inheritance has been independently reported by Kafura and Lee [4].

Related to this problem was the lack of good tools for prototyping languages. The implementation effort required for prototyping Hybrid was far too great to allow the language to evolve together with its implementation. (This is analogous to the evolution problem posed by open systems mentioned in the introduction.) In retrospect, a more promising approach would be to define Hybrid's semantics by mapping its language constructs to a formal, executable notation for describing concurrent behaviours, as outlined in [12].

One difficult design decision in any object-oriented language is what the first class values of the language shall be. The principle of homogeneity present notably in Smalltalk is that "everything" should be an object, in particular, object classes and, in certain cases, executable code (i.e., Smalltalk's "block expressions"). The importance of classes being objects should not be underestimated in the context of open systems: it is crucial that systems be able to evolve while they are running. In order to be able to instantiate objects of new or modified classes, it must be possible within the language to communicate with a class object that was not known at compile time.

In Hybrid, we attempted to apply the principle of homogeneity, but found that certain kinds of objects, notably delay queues and primitive objects like integers, could not be instantiated and manipulated in the same way as programmer-defined objects.

By far the most serious omission in Hybrid was the lack of an exception handling mechanism. The omission was intentional, not because we felt it was an unimportant issue, but rather because we believed it would be easier to evaluate exception handling approaches once we had experience with a running prototype. In fact, exception handling is essential if concurrency and strong-typing are to be meaningfully integrated into a useful object-oriented programming language. If we accept the view that any object is essentially an entity that provides a service to client objects, and that an object type is a description of the contract between the client and the server object with respect to these services, then without exceptions as an integral part of that contract, there is no way for an object to notify its client when the contract cannot be honored. For realistic concurrent applications, it must be possible for clients to catch and handle exceptions.

It is not our goal to survey exception handling mechanisms here. Nevertheless, we shall briefly list some of the requirements that a reasonable scheme would have to meet to satisfy our needs:

- Any operation may fail, raising an exception.
- The exceptions that may be raised are part of the type of an operation. Exceptions are themselves typed.
- Clients may define their own handlers, or inherit those of their own clients. There is always a default handler.
- Exception-handling should be no more expensive than message-passing (or procedure calls).
- The responsibility of a handler is to repair damage, not to provide an alternative execution path. (See also [6].)
- Methods should not have to depend upon exceptions to implement control flow. (It should always be possible to write code that does not require an exception to implement, say, loop termination.)

Since exception handling indicates a break from the normal flow of control, and should occur exceptionally (!), economy rather than generality should be a design criterion. Mechanisms should be motivated by real examples. The design choices include such questions as:

- Should exceptions have optional values associated with them (i.e., to inform the client what went wrong)?
- What actions may be taken by a handler (e.g., retry, resume, abort/re-raise, return, ...)?
- Can exceptions be raised within a handler, and, if so, what happens?

A related problem is that of *signaling*, though in this case it is less clear how a satisfactory solution may be arrived at. In some concurrent applications it is convenient to split up work amongst a number of cooperating objects. If, for example, a set of objects are working in parallel to solve a problem using several different approaches, the first to succeed may need to notify the others that the job is finished. Signals could be viewed as a kind of exception, but it seems more natural to view signals as a special kind of "express" message, as in ABCL/1 [17]. Again, we feel that proposals for new mechanisms should be well-motivated both by economy of function and by real examples.

Yet more difficult is the problem of how to encapsulate concurrent behaviour. Even though the abstraction of an RPC thread is extremely useful for structuring most concurrent computations, its limitations are only too obvious when higher level abstractions are called for. Concurrent subactivities and recoverable atomic transactions are two examples of useful concurrent control abstractions that are unpleasant to program directly using Hybrid's delay queue and delegation primitives. Although the object paradigm serves well to encapsulate certain kinds of concurrent behaviours (e.g., triggers, workers and administrators, etc.), it fails to capture encapsulation of control abstractions. A transaction cannot be viewed as an "object" in the usual sense, since it does not support an interface of operations. A satisfactory solution would allow for the addition of new control abstractions to the language, much in the same way that programmers may add new object classes. Well-designed, reusable control abstractions would eliminate the need for most programmers to have to deal with low-level synchronisation issues.

Finally, we note that Hybrid, like most programming languages, does not *scale* well. By this we mean that programming languages are typically classified as being either good for "prototyping" or for production, but not both. The division is generally made along the lines of dynamic vs. static binding, weak vs. strong typing, and interpretation vs. compilation. The only concession to scaling that is commonly made is in languages like Lisp and Pascal that may be either interpreted or compiled. Languages like Simula and C++ offer a choice between static and dynamic binding through the use of the "virtual" function declaration, but such decisions are frozen in the class definition. No language that we know of offers a choice between weak and strong typing (or between run-time and static type checking). We believe that scaling will be increasingly important in the development of open systems, not only to ease the transition from prototyping to production development (which can be accomplished by other means), but mainly to accommodate varying needs and system evolution. Both static and dynamic binding of the same object classes and operations can be simultaneously required by different applications, the first for efficiency reasons and the second for genericity. Although static type-checking is generally desirable, for evolving and open systems it is not practical to require all applications to be statically type-checked, since there will be no way for existing objects to communicate with new ones without re-compilation.

7 Conclusions

We have argued that a reference model of "objects and threads" should be used to guide the development of concurrent object-oriented programming languages, and we have shown how this model manifests itself in *Hybrid* an experimental language for programming with active objects. Although we can claim partial success with Hybrid, we are still a long way from raising the level of concurrent programming to the same degree that objects raise the level of sequential programming. We can summarize our conclusions in the following list of requirements for concurrent, object-oriented programming languages:

- A computational model for concurrently executing objects is needed for properly defining the semantics of new languages. Better tools for prototyping languages are needed to support research in this direction. We have developed an executable notation for specifying concurrent behaviour, which is based on process calculus [7], and we are using this notation to explore various semantic models for active objects [12].
- A mechanism for encapsulating concurrent control abstractions should be supported. Object classes do not always provide the best mechanism for encapsulating concurrent behaviour [13].
- Concurrency control mechanisms and object-oriented features can interfere in unexpected ways [14], [15]. Formal approaches appear promising as a means to better integrating concurrency and object-orientation.
- A realistic programming language must support exception handling for active objects.
- Signals (express messages) should be supported.
- A choice between dynamic and static binding of variables and operations should be offered for all object classes.
- A choice between run-time and static type-checking should be offered to support the evolution of open applications.

Acknowledgements

A shorter version of this paper has previously appeared as [11]. The author would like to thank the Centre de Recherche en Informatique de Nancy (CRIN) for their permission to re-use this material.

References

[1] G.A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

[2] 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.

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

[4] D.G. Kafura and K.H. Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages," Proceedings of the Third European Conference on Object-oriented Programming, pp. 131-145, Cambridge University Press, Nottingham, July 10-14, 1989.

[5] 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, pp. 61-105, Centre Universitaire d'Informatique, University of Geneva, June 1988.

[6] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

[7] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[8] 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.

[9] 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.

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

[11] O.M. Nierstrasz, "A Tour of Hybrid," in *Les Mardis Objets du CRIN, CRIN 89-R-072*, ed. G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre, pp. 237-248, Centre de Recherche en Informatique de Nancy, Vandoeuvre-lès-Nancy, 1989.

[12] O.M. Nierstrasz, "A Guide to Specifying Concurrent Behaviour with Abacus," in *Object Management*, ed. D.C. Tsichritzis, pp. 267-293, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[13] O.M. Nierstrasz and M. Papathomas, "Viewing Objects as Patterns of Communicating Agents," ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP '90, vol. 25, no. 10, pp. 38-43, Oct 1990.

[14] M. Papathomas, "Concurrency Issues in Object-Oriented Programming Languages," in *Object Oriented Development*, ed. D.C. Tsichritzis, pp. 207-245, Centre Universitaire d'Informatique, University of Geneva, July 1989.

[15] M. Papathomas and D. Konstantas, "Integrating Concurrency and Object-Oriented Programming – An Evaluation of Hybrid," in *Object Management*, ed. D.C. Tsichritzis, pp. 229-244, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[16] P. Wegner, "Dimensions of Object-Based Language Design," ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 168-182, Dec 1987.

[17] A. Yonezawa, J-P Briot and E. Shibayama, "Object-Oriented Concurrent Programming in ABCL/1," ACM SIGPLAN Notices, Proceedings OOPSLA '86, vol. 21, no. 11, pp. 258-268, Nov 1986.