Composing Active Objects

The Next 700 Concurrent Object-Oriented Languages*

Oscar Nierstrasz

University of Geneva†

Abstract

Many of the shortcomings of present-day object-oriented programming languages can be traced to two phenomena: (i) the lack of general support for software composition, and (ii) the semantic interference between language features addressing operational and compositional aspects of object-oriented programming. To remedy this situation, we propose the development of a "pattern language" for active objects in which objects and, more generally, applications, are constructed by composing software patterns. A "pattern" can be any reusable software abstraction, including functions, objects, classes and generics. In this paper we seek to establish both informal requirements for a pattern language and a formal basis for defining the semantics of patterns. First, we identify some basic requirements for supporting object composition and we review the principal language design choices with respect to these requirements. We then survey the various problems of semantic interference in existing languages. Next, we present a formal "object calculus" and show how it can be used to define the semantics of patterns in much the same way that the λ calculus can be used to give meaning to constructs of functional programming languages. We conclude by summarizing the principle open problems that remain to define a practical pattern language for active objects.

*.In: *Research Directions in Object-Based Concurrency,* ed. G. Agha, P. Wegner, A. Yonezawa, MIT Press, 1993, pp. 151-171.

†.*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

Object-oriented programming languages can either be characterized in terms of the features they support, or in terms of the way these features are used to change the nature of programming. We shall argue in this paper that too much emphasis has traditionally been placed on the details of specific object-oriented features and not enough on the more general topic of software composition. This leads us to propose the development of a "pattern language" for active objects that generalizes the principles of object-oriented programming to the definition and composition of software patterns.

If we consider the way in which object-oriented programming languages (OOPLs) are used, we see two complementary, yet distinct roles they play:

1. Applications consist of systems of cooperating and communicating objects, each with well-defined responsibilities.
2. Applications are compositions of pre-packaged software components (i.e., classes, abstract classes, etc.) that fit together to define an object system.

The first of these roles is a natural extension of the principles of modularity and data abstraction, inspired by the world of modelling and simulation and introduced with the language Simula [7]. The second role emerged with the definition of the Smalltalk language and environment [18]. Here, the emphasis is on the composition of applications from reusable software, facilitated by the definition of standard interfaces and components.

Although Smalltalk has demonstrated with some degree of success the reuse potential offered by OOPLs, and other languages have duplicated or improved upon these efforts, object-oriented programming has not revolutionized software development. Aside from the fact that the development of a library of truly reusable software components for a new application domain can entail a significant investment of effort, we see two technical reasons why present-day OOPLs fall short in this area: First, existing OOPLs emphasize *programming* over software composition. Reuse, whether implemented by inheritance or by other mechanisms, is achieved only by programming new objects, not by functionally composing existing objects or object parts. Second, language features that support composition and reuse often *interfere* with other features and mechanisms concerned with operational aspects of objects, such as concurrency and persistence. Both these problems can also be seen as side-effects of viewing OOP features as "add-ons" to existing languages, rather than as a fundamentally different programming paradigm in which software composition plays the leading role.

We propose that an effective object-oriented programming language will support application construction from reusable software patterns much in the same way that architectural designs can be composed from established architectural patterns [2]. By a software *pattern*, we mean any reusable software abstraction, that is, any identifiable piece of software that may be fruitfully exploited in multiple contexts. By *software composition* we mean the construction of applications by means of operators that bind patterns to one another. Patterns, in order to be useful, typically contain free variables that will be bound by actual parameters or by the environment in which they are instantiated, but we shall not initially make any assumptions about what precise mechanisms are used to define them. Rather, we point out that existing programming languages typically provide some specific pattern definition mechanisms but usually not any general purpose ones. For example, expressions, statements, functions, procedures, macros, generics, packages, modules, classes, and so on, are all specific kinds of patterns, but very few languages arguably provide programmers with the ability to define new kinds of patterns appropriate for specific problem domains. Similarly, few languages provide support for software composition that goes beyond the composition of expressions and statements, which is to say, low-level programming.

We take it for granted here that the operational view of applications as collections of cooperating and communicating objects has clear advantages, and in particular that such systems are inherently concurrent (whether or not the concurrency is exploited by a parallel or distributed implementation). What we are more concerned with is how we can build such systems by composing software patterns. Our goal in this paper is to explore the theme of a pattern language for active objects by investigating the interplay between the operational and compositional aspects of objects. More specifically, in section 2 we develop some informal requirements for OOPL design by considering the impact of the choice of the operational view of concurrent objects on software composition and reuse. Then, in section 3 we shall review the more significant problems of semantic interference between OOPL features and argue that the root of these problems generally lies in the confusion between the operational and compositional issues. As a concrete example, we shall look at some of these problems as they are manifested in *Hybrid*, a concurrent OOPL whose design attempts to integrate various seemingly orthogonal OOPL features. Finally, in section 4, we propose the development of a pattern language for active objects in which the semantics of higher-level patterns is given by a mapping to a formal "object calculus" called OC. We present OC through examples that illustrate the composition of active objects from more primitive patterns, and we conclude by identifying some of the key open research problems in the development of a pattern language for active objects.

2 Requirements for Composable Active Objects

The key features characterizing object-oriented languages are generally accepted to be [49]:

1. *Objects:* support for *encapsulation* of service and state as objects;
2. *Classes:* the ability to *instantiate* objects from class templates;
3. *Inheritance:* the ability to define new object templates by *incremental modification* of existing ones [50].

The operational view of objects is that of entities with private state which provide services to clients by exchanging messages. The compositional view is supported by classes and inheritance. A class is a software pattern that can be used to create an object by binding its instantiation parameters, or it can be used to derive a new (sub) class by binding self and super to a new environment including the instance variables and methods introduced by the subclass.

The operational view affects the compositional view in essentially two ways: First, the contexts in which an object may be instantiated should depend on the external message-passing interface of the object and not the internal details of the object's implementation. In particular, the way in which a client interacts with a given object should conform to a standard protocol that does not depend on the presence or absence of other clients. Second, classes should be composed of patterns representing object parts (i.e., instance variables, methods, interface specifications, constraints, etc.) in such a way that new classes can be derived without necessarily having to override (re-bind) inherited patterns. More specifically, the derivation interface should be abstract, so that a subclass can be derived without requiring detailed knowledge of the implementation of the superclass. In either case, compositionality is impaired if operational details are not sufficiently abstract.

In this section we will discuss a number of basic requirements on the operational view of concurrent objects that affect compositionality, and we will review various language design alternatives with respect to their compatibility with these requirements. We will concentrate on the client/server interaction, as these issues are somewhat better understood, and say less about the interplay between concurrency and inheritance, as this is a topic of much active research. The following requirements summarize the operational features that should be transparent in the client/server contract [40] (see also the dissertation of Papathomas [41]):

- *Object autonomy:* the internal state of an active object should be automatically protected from concurrent requests without the need for clients to explicitly synchronize with one another;
- *Internal concurrency:* it should be possible to replace a sequential implementation of an active object with a concurrent one without affecting clients;
- *Request scheduling transparency:* handling of local delays (i.e., scheduling the servicing of requests based on an object's internal state or on the nature of the request) should be transparent to the client;
- *Reply scheduling transparency:* handling of remote delays (i.e., scheduling of the receipt of a reply with respect to other ongoing tasks) should be transparent to the service provider.

A fifth requirement, which is not very well understood in the presence of inheritance, is a topic of much active research and study, as we shall see in section 3:

- *Composable synchronization policies:* mechanisms for defining synchronization policies should support the composition of new policies from existing ones.

If we accept these requirements as evaluation criteria, we can compare and rank various OOPL design alternatives with respect to the degree to which they satisfy them. Papathomas [41] has proposed a classification scheme for object-based concurrent languages based on the following three aspects: (i) *Object Models:* How is object consistency maintained in the presence of concurrency? (ii) *Internal Concurrency:* What means do objects have to manage multiple threads of control? (iii) *Client-Server Interaction:* What degree of control do objects have over the interaction protocol?

We shall now summarize from the observations presented in [40] concerning the evaluation of various basic OOPL design choices.

2.1 Object Autonomy

As a first cut, concurrent object-based languages can be distinguished according to the kind of object model they support, i.e., the means by which objects protect their internal state in the presence of concurrent threads. There are basically three different approaches, but, in a few rare cases, different approaches may be combined in a particular object model:

- *The Orthogonal Approach:* Synchronization is independent of object encapsulation. Mechanisms such as locks or semaphores must be used to synchronize concurrent invocations or internal consistency may be violated [8], [18], [32];
- *The Heterogeneous Approach:* Both active and passive objects are provided. Passive objects are protected by virtue of being used only within single-threaded active objects [14], [20];
- *The Homogeneous Approach:* All objects are "active" and have control over the synchronization of concurrent requests [3],[34],[46].

The orthogonal approach fails to support object autonomy. Since no implicit protection of objects' states is provided, only specially designed objects can be safely used in the presence of concurrent clients. Objects that behave correctly in a sequential setting may "break" if exposed to concurrent requests. The heterogeneous approach suffers because one cannot typically interchange implementations of active and passive objects. The existence of two different kinds of objects limits code reusability, since it may either force duplicate (active and passive) implementations of essentially the same object, or unnatural decomposition of applications in order to maximally exploit active objects. In some cases, a mixed model is supported, as in Argus [27], where active objects (*guardians*) may be internally concurrent, and so must synchronize accesses to passive objects (*clusters*) as in the orthogonal approach. The homogeneous approach does not suffer from these defects, but there is a potential performance problem if active objects are too "heavyweight" in contrast to passive objects. It may be possible, however, for a compiler to generate automatically more efficient code for active objects occurring in contexts that require no synchronization [39].

2.2 Internal Concurrency

We can make various distinctions according to the means for coping with internal concurrency [49]:

- *Sequential Objects* have a single active thread of control [5], [3], [53];
- *Quasi-Concurrent Objects* explicitly interleave multiple threads of control [34];
- *Concurrent Objects* support internally concurrent threads. We can further distinguish the cases in which thread creation is either:
    - o *Client-driven:* threads are implicitly created upon the receipt of a request, as in the orthogonal object model;
    - o *Server-driven:* in which case objects must explicitly create a new internal thread (for example, using the *become* primitive of actor languages [1]).

Purely sequential objects that follow a strict RPC protocol overly restrict potential concurrency as they are unable to handle remote delays [28]. Additional concurrency can only be introduced by delegating tasks to proxy objects (e.g., Cboxes [52] and future variables [53]), that will reply to the original client (i.e., by abandoning the strict RPC protocol). Even quasi-concurrent objects must make use of additional active objects to achieve true concurrency. Client-driven concurrency has the disadvantages of the orthogonal object model. Server-driven internal concurrency has the clearest advantages in that concurrent activities are properly encapsulated: there is no need to artificially introduce additional *external* objects just to distribute the work load.

2.3 Client/Server Interaction

Finally, we can distinguish languages according to the forms of client/server interaction supported. The main concerns from the point of view of the client and the server are, respectively, reply scheduling (the control the client has over the delivery of the reply) and request scheduling (the control the server has over the acceptance of requests).

From the server's point of view, the key issue is whether requests are accepted unconditionally or not:

- *Unconditional acceptance:* No synchronization with the state of the object occurs [8], [18], [32].
- *Conditional acceptance:*
    - o *Explicit acceptance:* An explicit *accept* statement is used to indicate which requests will be served [5],[3],[53].
    - o *Reflective computation:* The arrival of a message triggers a reflective computation in the associated "meta-object" which is capable of directly manipulating messages, mailboxes, etc. [11],[48].
    - o *Activation conditions:* Implicit or explicit conditions govern the acceptance of requests.
        - — *Representation specific:* Conditions are expressed in terms of the object's internal state [24],[34],[47].
        - — *Abstract:* Message acceptance depends on the *abstract state* of an object [10],[20],[46].

The various approaches to conditional acceptance are generally equivalent in expressive power, provided that there is some way to delay servicing of a request based on the contents of the request message. The main difference between these approaches from the point of view of composition is the extent to which the synchronization policy of the server can be understood in abstract terms to facilitate reusability and incremental modification. In this respect the approach of abstract states (whether by means of *enabled sets* [46], *behaviour abstraction* [20],[21] or *synchronizing actions* [33], etc.) appears to be the most promising, though it is not clear how best to accommodate (1) local delays based on message contents, which potentially entail an infinite number of "abstract" states, and (2) incremental modification, which extends and alters the graph of transitions between abstract states.

From the client's point of view, the most important distinction is the interaction protocol supported:

- *One-way Message-Passing:* higher-level protocols are explicitly programmed [10];
- *Request/Reply:* every request necessarily entails a reply.
    - *RPC:* a requesting thread blocks until a reply is received. Internal threads may be sequential [3], quasi-concurrent [34] or concurrent [47], as described above;
    - *Proxies:* sending of requests and receipt of replies may be delegated to (external) proxy objects [14],[52],[53].

Reply scheduling is essentially concerned with three issues: (1) the interleaving of client activities, (2) controlling the reply address, and (3) obtaining the reply. With one-way message-passing (whether synchronous or asynchronous) clients have complete control over all three issues, at the expense of abstraction. In particular, servers are obliged to keep track of the reply address. A protocol in which requests implicitly entail replies eliminates this problem. RPC in combination with purely sequential objects exhibits the problems mentioned earlier, but is quite flexible in combination with concurrent and quasi-concurrent objects. Concurrent RPC has the advantage over both proxies and quasi-concurrent RPC in that additional external objects do not need to be introduced in order to split up a logically concurrent task. Internally concurrent threads are properly encapsulated.

2.4 Observations

After eliminating approaches that fail to adequately support one requirement or another, we are left looking for: (i) a homogeneous model of active objects (ii) that supports server-driven internal concurrency (and hence hierarchical composition of active objects), (iii) a client-server interaction protocol that is by default one of remote procedure calls, (iv) but which provides both clients and servers transparent control over the scheduling of requests and replies. The main technical difficulty appears to be the specification of synchronization policies in such a way as to adequately support incremental modification.

3 Language Feature Integration

Now that we have established some basic requirements for OOPL design, let us consider the interplay between various language features.

Wegner [49],[51] has identified the following dimensions as a design space for object-based languages: objects, classes and types, inheritance and delegation, data abstraction, strong typing, concurrency and persistence. Semantic interference between object-oriented dimensions has been noted by numerous researchers. Let us simplify the problem space by considering only the following four dimensions: *encapsulation* (objects + classes), *inheritance*, *strong typing* and *concurrency.* As we shall see, several of the difficulties that arise are due to interference between the operational model, in which the client/server relationship is that of message-passing between objects, and the compositional model, which supports very different client/server relationships, such as inheritance between classes and containment for instance variables.

We shall first discuss interference in general, and then we will look at particular examples of interference of language features in Hybrid, a concurrent object-oriented language.

3.1 Orthogonality and Interference of Object-Oriented Dimensions

Inheritance vs. Encapsulation

The main difficulty with inheritance in object-oriented languages is that, in order to define meaningful extensions to inherited behaviour, subclasses must "violate encapsulation" of their parent classes [44],[43] (i.e., they must be able to access instance variables otherwise hidden from clients of superclass instances). Furthermore, once an inheritance hierarchy has been defined, modifications to superclasses can affect subclasses in unexpected ways, thus limiting the freedom to modify the implementation of classes and, once again, "violating encapsulation".

In both cases the problem arises from a failure to distinguish between the *two* kinds of client of a class: other instances, and subclasses. Various solutions have been proposed, including:

- Provide a common interface to both kinds of client (which ignores the fact that different clients typically have different needs);
- Separate interfaces for ordinary clients and subclasses (including private class methods, etc.);
- Provide mechanisms to extend inherited behaviour in a controlled fashion, such as the *inner* construct of Simula [7] and Beta [25] and the "before" and "after" methods of CLOS [31];
- Adopt a more compositional approach as an alternative to inheritance, in which the dependencies of class templates (or *habitats* [43]) are made explicit.

As a general principle, we conclude that binding interfaces of any software pattern should be explicit, and different classes of clients should be distinguished.

Types vs. Encapsulation

Through types we mean to express the abstract interface of objects. Type-checking is a means to increase our confidence that we are using objects in a consistent way. It is convenient to view types as partial specifications of behaviour, that is, as *predicates* describing those properties of an object that we are interested in as clients. In this view, subtypes are simply stronger predicates. An object, which is an instance of a particular class, can thus be viewed as simultaneously belonging to all types whose properties it holds. This justifies dynamic binding in object-oriented languages that allow one to bind variables to any instance that satisfies the type constraint of the variable.

Various polymorphic type theories based on the λ calculus have been developed for type-checking programming languages (see, for example, [12]), but they consider a type to stand for a set of *values*. Such a theory can be applied to languages that support functional objects (i.e., in which "objects" are tuples of values and functions). Types may be refined to subtypes by applying the contra-variant rule: input parameters to functions may be generalized and return values may be refined. (This is the essence of polymorphism: that instances of the supertype may be replaced by instances of the refined type without causing surprises for clients).

In most OOPLs, however, objects are not values, but entities with changing state. The arguments to operations are used not only to compute return values, but may be used to change the state of the object. In languages with updatable attributes, for example, attribute types cannot be refined, as they serve as both inputs and outputs [13].

More difficult to handle is the problem of changing *roles* [42]: an object may have multiple roles with respect to different clients, and these roles may change during its lifetime. A type theory that would deal with state change and multiple roles must take into account certain temporal properties of objects.

Types vs. Inheritance

Most difficulties with types and inheritance are due to a confusion between types and classes. It is commonly assumed that the class (inheritance) hierarchy and the type hierarchy must correspond, and that the type of a subclass will always be a subtype of the type of the superclass. If we view types as interface specifications, then it is clear that classes unrelated by inheritance may share a common interface. In this case, we may have type equivalences and subtype relationships across the class hierarchy.

Conversely, in some languages, such as Eiffel, classes may be derived by applying the co-variant rule (which allows both input parameters as well as return values to be of a refined type). In this case subclasses will not satisfy the type constraint of the superclass, and there will be no subtyping relationship between the two classes. Furthermore, if it is possible for a subclass to "hide" part of an inherited interface, then there will also be no subtype relationship between superclasses and subclasses. For example, in Eiffel, one may use inheritance to define a FixedStack class that inherits its interface from the (abstract) class Stack and its internal representation from the (implementation) class Array and hiding the operations inherited from Array. But if we would now bind a FixedStack instance to a variable of type Array, the Array operations would be exposed and encapsulation of the FixedStack instance would be violated.

Inheritance and subtyping must therefore be seen as independent [16]. In some languages, such as POOL-I [4], types and classes are separate concepts.

Concurrency vs. Encapsulation

As we have seen already in section 2, the view of objects as encapsulations of service and state must be treated delicately in the presence of concurrency. In the simplest case, encapsulation of passive objects is violated by concurrent clients that do not synchronize their requests. Conversely, encapsulation of clients is compromised if it is their responsibility to explicitly synchronize requests to shared, passive objects. We conclude that, at least, some model of active objects is desirable.

The situation is complicated by the need for objects that are capable of coping with local and remote delays by means of scheduling requests and replies. As we argued in section 2, request and reply scheduling should be transparent to, respectively, clients and servers, as well as to "nearby" objects. In Hybrid [39], for example, objects that make use of such scheduling mechanisms cannot be used with impunity as instance variables, as the use of these mechanisms will also affect the enclosing object, possibly violating encapsulation.

Concurrency vs. Inheritance

A very similar situation exists when we consider inheritance, where the "nearby" objects are subclass instances. Here the difficulty is that inherited behaviour may be very sensitive to cooperation between methods making use of concurrency mechanisms to support request/reply scheduling. It can, in general, be very difficult to extend the inherited behaviour in a consistent way. Superclass encapsulation is thus violated by the need to expose implementation details, and reusability is compromised since inherited methods cannot synchronize with subclass methods [20].

Various approaches to address this problem essentially make the compositional (i.e., inheritance) interface more explicit. As mentioned in section 2, such approaches are typically based on abstract states and the separation of the synchronization policy from the implementation of methods.

Concurrency vs. Types

If we indeed view types as partial specifications of behaviour, then it is clear that merely characterizing the message-passing interface of an object does not go very far. In particular, when we are concerned with software composition, we are interested in knowing when one object is substitutable for another in a given context, that is, when an object is "plug-compatible" with another. To this end, we must be able to characterize certain temporal and concurrent properties that can affect the various clients of an object. A short, non-exhaustive list would include:

- *Non-uniform service availability:* the fact that requests to perform certain operations may be delayed or rejected depending on the state of the object;
- *Non-client/server protocols:* active objects may conform to alternative message-passing protocols such as early reply [28] and send/receive/reply [17];
- *Multiple concurrent clients:* objects capable of handling multiple concurrent requests may not work correctly if placed in a sequential environment (e.g., a communication buffer), and, conversely, clients may not work correctly if they expect a strictly sequential behaviour from a service provider (e.g., if they do not expect the state to change in between one message exchange and another).

What is at issue is that the client/server contract cannot always be reduced to a static message-passing interface that specifies only the requests accepted, the consequent replies, and the types of values they may contain. In general, it will be necessary to express some temporal properties of the message-passing interface in order to reason about valid object compositions. Note that we do not seek to fully specify object behaviours and prove that their composition correctly implements an abstract specification. Rather, we only wish to verify some weaker safety and liveness conditions: when is it safe for an object to send certain messages, and what kind of replies can be expected. For example, a stack and a queue would both satisfy the type constraint that only specifies the possible orderings of put and get operations, though their fully abstract specifications would differ.

Understanding plug-compatibility in terms of the visible interactions an object may exhibit appears promising as a means to abstract away from implementation details [35],[37]. Modal logics [26] and propositional temporal logic [6] appear useful as formalisms for expressing abstract properties of concurrent systems, and can also be used to reason about changing roles [6], though it is not yet clear whether they will prove appropriate as a formalism for developing a type theory.

3.2 Semantic Interference in Hybrid

Let us briefly consider how some of these problems of semantic interference manifest themselves an a particular language. *Hybrid* is a concurrent OOPL whose design attempts to integrate cleanly various seemingly orthogonal language features [34]. In particular, the prototype implementation provides [22]:

- o objects + classes + inheritance
- o strong typing + genericity
- o concurrency
- o persistence

Features currently being implemented include:

- o homogeneity (“everything” is an object)
- o distribution (see [23])

Hybrid is based on a model of active objects. Active objects (as opposed to passive objects) have complete control over the scheduling of incoming requests. They may, therefore, not only ensure mutual exclusion of method activations (if required), but they may also exercise a fine degree of control over local and remote delays and internal concurrency.

Concurrent activity in Hybrid is delimited by *domains*. Each domain is essentially a quasi-concurrent process, that is, at most one thread of control may be running at any time within a domain, but mechanisms are provided for switching attention from one thread to another. Domains are always mapped to a single, autonomous active object (i.e., instance variables cannot have their own domain, nor can a group of autonomous objects belong to a single domain). A domain, by default, handles exactly one request at a time. A domain only accepts new requests while in the *idle* state. While servicing a request, a domain is either *busy* or *waiting* (if there is a pending call to another domain).

A logical thread of control that traces the logically connected requests and replies between domains is called an *activity.* (We introduce this term to avoid confusion with the term “thread,” which is usually understood to be a property of a single process only.) Domains permit recursive calls as such requests will be associated to the currently running activity. In Figure 1 we see an activity as a trace of *call* (white) and *reply* (black) messages between domains. New activities are created by calling special operations, called *reflexes*, that yield no return value. The effect of calling a reflex is to place a new request in the target’s message queue, and then to continue without waiting for a reply.

Domains can exercise a fine degree of control over the acceptance of requests by associating services (operations) to *delay queues*. Such queues can be either *open* or *closed*. When a domain is idle, it will accept requests only from its open queues (Figure 2). Local delays that do not depend on message contents are easy to capture using delay queues. For example, a bounded buffer can be modelled by associating the put and get operations to notFull and notEmpty delay queues.

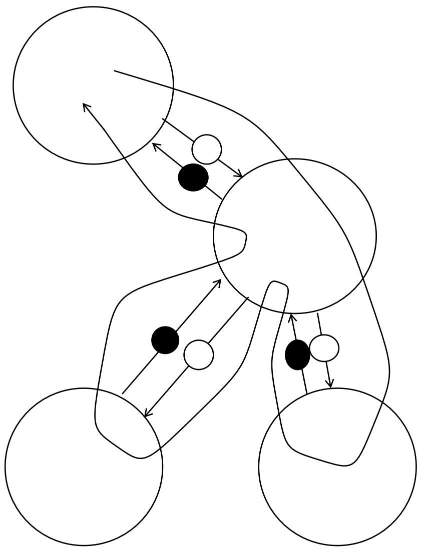
Figure 1 Domains and Activities

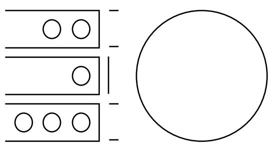
Figure 2 Delay Queues

Domains may be quasi-concurrent. By use of the mechanism of *delegated calls*, a domain may interleave the servicing of requests: a delegated call to another domain causes the calling domain to immediately become *idle* instead of waiting for its reply. The calling context is temporarily saved, and is restored when the reply is available. For example, in Figure 3 we see an object servicing a request on behalf of activity α delegate a call to another domain, allowing it to switch its attention temporarily to request β.

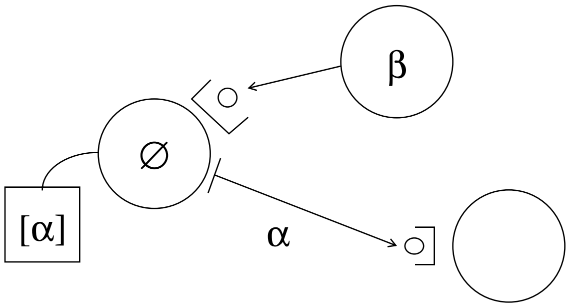

Figure 3 Delegated Calls

Delegated calls are very expressive, and can be used to handle both remote delays and, in combination with delay queues, local delays that depend on message contents. Internal concurrency is not directly supported in Hybrid, but it is possible to simulate it by means of delegated calls to "worker" objects.

Although Hybrid succeeded in demonstrating that object-oriented features could be combined with concurrency, persistence and strong-typing in a single programming language, it cannot be claimed that these features were fully integrated. In particular, the central issues of composition and reuse have been addressed in only a limited way. Let us consider some of the problems encountered:

- *Homogeneity:* although it was intended that all objects should have the same first-class status, it is not at all clear how to treat active objects as first-class values; similarly, delay queues are object-like, but cannot be consistently treated as other objects;
- *Types:* strong-typing on the basis of signatures (operation names and argument and return types) tells us nothing about the behaviour of concurrent and mutable objects, and hence tells us very little of use concerning substitutability of objects with superficially similar interfaces [37];
- *Composition:* objects implemented in terms of delay queues and delegated calls cannot be (re)used in arbitrary contexts without potentially introducing deadlocks (through closed delay queues) or violating encapsulation (by exposing an enclosing object during a delegated call);
- *Inheritance:* by the same token, it can be difficult or impossible for a subclass to incrementally modify the synchronization policy of a superclass in a consistent way [20], since inherited methods have no way of accessing delay queues introduced by subclasses without being overridden;
- *Encapsulation:* it is not generally possible to define higher level patterns of behaviour using only classes, inheritance and genericity; for example, mechanisms to support internal concurrency, internal triggers, and transactions, although they can be programmed in Hybrid, apparently cannot be packaged in a general way.

Note that each of these issues is concerned with some aspect of *composition* of concurrent behaviour.

In a more practical vein, the Hybrid project suffered from (1) the lack of any generally accepted computational model for active objects suitable for defining the semantics of language constructs, and (2) the lack of good prototyping tools for experimenting with language design alternatives. These observations have led us to search for a computational model and notation for active objects that can be used as an executable target for language specifications [35],[36],[38]. At the same time, it is essential to be very clear about the informal goals and requirements for language design [40] and to study the problems of semantic interference from these viewpoints.

4 Towards a Pattern Language for Active Objects

We have presented the object-oriented paradigm as a two-level programming model in which an application can be viewed both operationally as a configuration of cooperating active objects, and functionally as a composition of software patterns that yields this configuration. A programming language to support this paradigm should therefore address equally well the operational aspects of active objects and the functional aspects of specifying and composing software patterns.

We propose the development of a pattern language for active objects which would be used according to the three levels of Figure 4:

1. *Computational Model:* at this level the operational behaviour of software patterns is specified by means of a formal "object calculus"; the object calculus gives meaning to patterns much in the same way that the λ calculus is used to give semantics to sequential programming languages [45]: it provides both a formal notation for the semantics of language features and a possible operational interpretation of their behaviour;
2. *Language Primitives:* at the language level we identify the particular behavioural patterns that define our object model, the kinds of objects we can construct, and the kinds of abstractions that will be appropriate for composing objects;
3. *Software Components:* finally, in terms of the language primitives provided, we are able to define software components that can be composed to produce running applications.

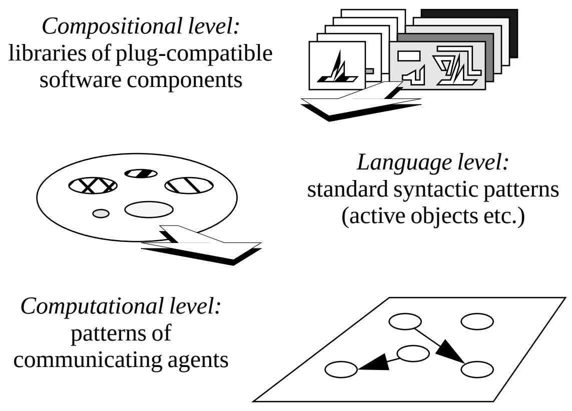

Figure 4 Viewing Application Development in Terms of Hierarchies of Patterns

Applications would be constructed by composing reusable software patterns much in the way that architectural designs can be composed from established architectural patterns [2]. Language designers would use the pattern language to develop the primitive patterns of object models. Designers of application frameworks and libraries would define the standard interfaces and patterns for particular application domains. Application developers would produce new applications by composing the pre-defined application patterns.

In this section we shall present a concrete object calculus, called OC, and we shall illustrate how higher-level patterns for composing active objects can be defined by a mapping to OC. We shall conclude by summarizing the principal research problems that remain to defining a practical pattern language.

4.1 OC — An Object Calculus

The object calculus that we shall present informally here is based on recent developments in process calculi. In particular, it extends the π calculus [30] to tuple-based communication, and to functional application and communication of expressions as in the λ calculus. For details on these extensions and a description of the formal semantics of OC, see [38].

The terms of OC consist of a set of expressions representing *agents, A*. Agents are composed of a set of *names, N*, a set of *input offers, X*, and a set of output offers, or communicable *values, V*. We let *a, b, c* range over *A*, *n, m* over *N*, *x* over *X*, and *v* over *V*.

An agent is an entity that may offer to (synchronously) input or output some message tuple, or may silently change state (if it is itself composed of communicating agents). A communication may take place if the input and output offers of two agents match. The syntax of input and output offers are, respectively:

$$x ::= n \mid n? \mid (x,\ldots,x)$$
$$v ::= n \mid a \mid (v,\ldots,v)$$

The notation "?" indicates the introduction of a new variable name. Input and output offers match if they correspond structurally and if values appearing in the input offer are equal to the corresponding values in the output offer. Variables match any value and are subsequently bound to the values they match. (So, x? matches y and (x,x?) matches (x,y), but (x,x) does not match (x,y).)

The syntax of agents is as follows (loosest to tightest binding):

$$a ::= a\&a \mid n{:=}a \mid a|a \mid x \rightarrow a \mid v \,\hat{}\, a \mid a@v \mid n \setminus a \mid n \mid \mathsf{nil}$$

Concurrent composition is &, recursion is :=, (left-preferential) choice is |, abstraction (input) is →, output is ^, (functional) application is @, restriction (of local names) is \ and the inactive agent is given by nil. A name *n* stands for an agent only if it has been bound to an agent expression by a communication or by a recursive definition.

As an example, consider the following specification of a counting semaphore:

```
csem := (p→nil & v→csem)
```

csem is a recursively defined composite agent. It consist of two concurrently composed agents, p→nil, which can accept the message p and become nil, and v→csem, which can accept v and become csem.

Note that csem accurately captures the semantics of a counting semaphore. A resource may be claimed by communicating a p, yielding the system:

```
nil & v→csem
```

To recover the original state of csem, a v must be sent. With each additional v communicated, a new copy of p→nil is introduced.

We can use the counting semaphore agent to restrict access to a resource that repeatedly accepts print messages:

```
resource := print → resource
```

Two clients synchronize via the semaphore before accessing the resource:

```
c1 := p^print^print^v^nil
c2 := p^print^print^print^v^nil
```

So, c1 may output p and become print^print^v^nil. In the system:

```
csem & resource & c1 & c2
```

either of c1 or c2 (non-deterministically) claims the resource, and the other must wait until the semaphore is released again.

A binary semaphore is very similar, but is a sequential rather than a concurrent agent:

```
bsem := p→v→bsem | v→bsem
```

The choice operator | selects either one input or the other. So bsem can input p to become v→bsem or v to become bsem.

Functional composition is achieved by means of the @ operator. It is similar to remote communication via ^ except that no competition can take place. The left operand of @ is treated as though it were a function and its right operand as its argument. The target of @ may not communicate with any external agents but must eventually accept the argument as input (if it can), or deadlock.

To understand how @ works, let us now consider an agent that models the behaviour of the Linda *tuple space* [15]. Linda provides a small set of primitives to allow concurrent processes to communicate and synchronize by writing and reading tuples to a so-called tuple space. A process may write a tuple using the non-blocking *out* primitive, and may read a tuple either destructively with the *in* primitive, or non-destructively with the *rd* primitive. Both read primitives block if no matching tuple exists. The following agent, linda, supports these three primitives:

```
linda := (out,t?) → ( linda & tuple@t )
tuple := t? → ( (in,t)^nil | (rd,t)^tuple@t )
```

When linda receives a request to create a new tuple, it replaces itself by a system including a copy of itself and an agent that implements the behaviour of a tuple. The tuple is instantiated by forcing tuple to accept the value t as input, i.e., by treating it as a function. The resulting agent can either (destructively) output (in,t) and become nil, or (non-destructively) send (rd,t) and become tuple@t again.

The last operator to introduce is restriction (\), which defines the scope of a new name. If that name is ever communicated outside that scope, then it must be distinguished by renaming (i.e., by α-conversion) to a unique name. For example, a generator of unique identifiers could be specified as:

```
idgen := x \ ((new,x)^idgen)
```

Since the name x occurs within its own local scope, it is guaranteed to be globally unique upon communication to clients.

4.2 Modeling Active Objects with OC

In much the same way that denotational semantics can be attributed to sequential programming languages by mapping their syntactic features to λ calculus terms [45], semantics can be given to concurrent languages by mapping their constructs to process calculi [29]. We shall illustrate this approach not by defining a complete programming language, but by defining some derived operators of OC that might correspond to constructs of a concurrent OOPL.

Let us start by considering the following example of a template for an object that recursively computes factorials (modelled after the standard example used to illustrate concepts of actor languages [1], [19]):

```
recFact := self? →
        [fact, n?] ⇒
                become(recFact) ;
                if (n=0)
                then return(1)
                else   self ⇐ [fact,n-1]
                        >> (k? → (n × k) >> (x? → return(x)))
```

The agent recFact accepts a name (standing for an object identifier) as input, and becomes an object with self bound to its oid. The body of the object is specified using a number of derived operators (⇒, ;, become, etc.). The object accepts requests ([fact,n?]) to compute factorials, and eventually replies to clients with the result. Upon receiving a request, it immediately "becomes" a copy of itself so that it may start to accept new requests concurrently with the processing of the pending request. This is possible since instances of recFact carry no state information other than that related to the pending request.

If the request is to compute the factorial of 0, it returns the result 1, otherwise it sends itself a request to compute the factorial of n-1. We may informally read the expression E >>(x? → A) as let x = E in A. Then the result of the subexpression self ⇐ [fact,n-1] is multiplied by n, and that result is in turn sent back to the client.

To create a new copy of recFact and ask it to compute the factorial of 10, we may now write:

new(recFact) >> (rf? → (rf ⇐ [fact,10] >> done))

In the following encoding of the derived operators, we adopt the convention that requests from clients to objects are always of the form:

(*oid*,*request*,*args*,*rid*)

where *oid* is the object's oid, *request* is the name of the service, *args* its arguments, and *rid* the client's request identifier. Replies will be of the form:

(*rid*,result,val)

where *val* is the value returned. Furthermore, self and client will be reserved keywords standing respectively for an object's oid and the rid of the current request. Now let us consider the semantics of each of the derived operators.

The operator ⇒ simply expands the request form [M,A] to the appropriate input offer, binding client in the body:

$\llbracket [M,A] \Rightarrow S \rrbracket = (\text{self}, M, \llbracket A \rrbracket, \text{client?}) \rightarrow \llbracket S \rrbracket$

Statements are agents that take a continuation agent as input:

$\llbracket Stmt \,;\, Cont \rrbracket = \llbracket Stmt \rrbracket @ \llbracket Cont \rrbracket$

The become statement instantiates concurrent copies of its continuation and the object template O with self as the oid.

$\llbracket \text{become}(O) \rrbracket = \text{cont?} \rightarrow (\llbracket O \rrbracket @\text{self} \;\&\; \text{cont})$

The if then else construct applies two possible continuations to a Boolean agent that selects one of them:

$\llbracket \text{if } B \text{ then } C1 \text{ else } C2 \rrbracket = \llbracket B \rrbracket @ (\llbracket C1 \rrbracket, \llbracket C2 \rrbracket)$

Booleans are encoded as follows:

$\llbracket \text{true} \rrbracket = (\text{a?},\text{b?}) \rightarrow \text{a}$

$\llbracket \text{false} \rrbracket = (\text{a?},\text{b?}) \rightarrow \text{b}$

The operators +, -, ×, /, =, and >> are just syntactic sugar for applying a binary tuple with the operator in the first position:

$\llbracket A \; Op \; B \rrbracket = \llbracket A \rrbracket @ (Op, \llbracket B \rrbracket)$, where $Op \in \{+, -, \times, /, =, >>\}$

The representation of numbers is closely modelled after the standard encoding in the λ calculus. We shall not repeat the encoding here (see [38] for details) but remark only that n=0 returns a Boolean agent, n-1 and n × k return integer agents, and n>>econt evaluates to econt@n, where econt is an expression continuation of the form n? → cont for some name n. So E >> (x? → cont) eventually binds x to the value of E in cont.

The return operator is straightforward:

$\llbracket \text{return}(V) \rrbracket = (\text{client},\text{result},\llbracket V \rrbracket)\text{\^{}nil}$

The request/reply protocol is encapsulated in the following construct:

$\llbracket X \Leftarrow [M,A] \rrbracket = (>>,\text{econt?}) \rightarrow$
  $\text{rid} \setminus ((\llbracket X \rrbracket, M, \llbracket A \rrbracket, \text{rid})$
    $\text{\^{}} (\text{rid},\text{result},\text{value?}) \rightarrow \text{econt@value})$

$X \Leftarrow [M,A]$ takes as input the expression continuation that will eventually receive the result. A new request identifier rid is then specified so that the request can be distinguished from those of other clients. The request is then dispatched to the object with oid X, the return value is accepted, and the value is applied to the expression continuation econt.

Finally, we have:

$\llbracket \text{new}(O) \rrbracket = (>>,\text{econt?}) \rightarrow \text{id} \setminus (\llbracket O \rrbracket @\text{id} \;\&\; \text{econt@id})$

which instantiates an object with new oid id from template O and passes id to its expression continuation, and:

$\llbracket \text{done} \rrbracket = \text{x?} \rightarrow \text{nil}$

which can be used as an expression terminator.

It should be noted that the derived operators presented here are intended only to illustrate, through a simple, yet non-trivial example, the general approach of defining language constructs by means of a mapping to an object calculus. We do not mean to suggest that these operators are (or are not) especially well-suited for general-purpose composition of active objects.

4.3 Open problems

An object calculus would serve as only the lowest level of a pattern language. A pattern language would consist of a core language corresponding to an object calculus, a set of pre-defined syntactic patterns, and a pattern declaration mechanism for binding syntactic patterns to their interpretation as behavioural patterns. We can identify three main lines of activity towards developing a practical pattern language:

- *Theoretical:* When are two patterns behaviourally equivalent? Is there a simple "normal form" for patterns to which they can be transformed? What is an appropriate type theory for "plug-compatibility" of patterns?
- *Experimental:* What object models meet our requirements for composing active objects? What are suitable basic patterns at the language-level? What constitutes a well-designed pattern library?
- *Pragmatical:* What are effective strategies for implementing a pattern language on conventional hardware? How can parallel and distributed architecture be exploited? What are the interoperability issues when communicating with applications written in other languages?

Equivalence and Plug-compatibility

There is a well-developed body of literature on behavioural equivalences for process calculi. See, for example, [29] for an introduction. Nevertheless, this is an area of much active research, and it is far from clear what “tests” should determine when two terms of a process calculus specify the “same” behaviour.

Aside from purely theoretical concerns, a well-defined notion of equivalence has several important practical applications. Garbage collection can be performed if we can determine that a system is equivalent to another one with certain agents removed. Optimization is possible if we can determine that certain behavioural patterns are equivalent to other ones that can be implemented more efficiently. During compilation, certain expressions may be pre-evaluated if we can show that lazy and eager evaluation will give the same results.

A closely related subject is that of a type theory for composing patterns. If we consider a type as a partial specification of behaviour, that is, as a *contract* between a client and a provider of services, and if we can verify these contracts statically, then we can reason about the plug-compatibility of patterns. In the examples we have seen in OC, the composition of agents that do not “fit” simply results in a deadlocked agent. (For example, if 10 then a else b will deadlock since 10 is not a Boolean agent and will not be able to accept the message (a,b).) In general, an adequate type theory for pattern composition should take into account not just static aspects of services provided at a particular point in time, but also dynamic aspects of properties associated with future reachable states of agents. As such, we imagine that modal and temporal logics will be useful in developing such a theory (recall the discussion above in section 3).

Object Models

An object model for a pattern language determines the kinds of messages exchanged by objects, the structure of objects and their constituent parts, and the primitive patterns that can be used to compose objects and to define higher-level patterns. A suitable object model should satisfy the various requirements we have posed in section 2 and should address the problems of semantic interference identified in section 3. Ultimately, the acid test of a good object model will be quality of the application-level patterns that can be developed and the ease with which applications can be built by composing such patterns.

Implementation

We have introduced OC as an executable specification language for the development, rapid prototyping and evaluation of OOPL design choices. We have also proposed that an object calculus like OC be used as the kernel of a pattern language. Although a prototype implementation of an OC interpreter exists [38], it is by no means efficient enough to use as the kernel of a real programming language. As there is much in common with the formal foundations of functional programming languages, we expect that much of the research that has been performed to develop efficient implementations of functional languages will also apply to the implementation of a pattern language.

There are, of course a number of traditional pragmatic issues to be studied: interpretation vs. compilation, type-checking, version management, interoperability with other languages, programming and debugging tools, and so on. In the arena of more exotic implementation issues, we may consider visual formalisms for active objects and composing software patterns, and interactive composition and monitoring of applications by means of “active graphics” whose formal semantics may be given by graphical transformation rules much in the same way that the structural operational semantics of process calculi are given by transition rules.

5 Concluding Remarks

We have presented the view that present-day object-oriented programming languages emphasize too much *programming* at the expense of *software composition*. To remedy this situation, we propose the development of a *pattern language* for active objects that addresses both the operational view of applications as collections of communication objects and the functional view of applications as compositions of plug-compatible software patterns.

As set of concrete steps towards this goal, we have (i) identified a number of language requirements for composing active objects, (ii) reviewed the principal classes of OOPLs with respect to these requirements, (iii) reviewed the principal difficulties of semantic interference between OOP language features, (iv) proposed an *object calculus* as a formal basis for specifying, understanding, prototyping and evaluating OOPLs, and (v) outlined a program for the development of a pattern language for active objects.

The key difficulties to be resolved appear to be (i) the development of an object model that supports well the composition of objects with non-trivial synchronization policies, (ii) the development of basic language-level patterns to support composition and reuse, (iii) the development of a type theory for plug-compatibility of patterns.

Acknowledgements

The author would like to acknowledge the contribution of Michael Papathomas, whose work on evaluating language design alternatives is summarized in section 2.

Sincere thanks are also due to Peter Wegner for his careful reading of various versions of the manuscript and for suggesting significant improvements to the presentation of this material.

References

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

[2] C. Alexander, S. Ishakawa and M. Silverstein, *A Pattern Language*, Oxford University Press, New York, 1977.

[3] P. America, “POOL-T: A Parallel Object-Oriented Language,” in *Object-Oriented Concurrent Programming*, ed. A. Yonezawa, M. Tokoro, pp. 199-220, The MIT Press, Cambridge, Massachusetts, 1987.

[4] P. America, “A Parallel Object-Oriented Language with Inheritance and Subtyping,” Proceedings OOPSLA/ECOOP ’90, ACM SIGPLAN Notices, vol. 25, no. 10, pp. 161-168, Oct 1990.

[5] American National Standards Institute, Inc., *The Programming Language Ada Reference Manual,* LNCS 155, Springer-Verlag, 1983.

[6] C. Arapis, "Temporal Specifications of Object Behaviour," in *Proceedings Third International Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems<Family Times,* ed. B. Thalheim, J. Demetrovics, H.-D. Gerhardt, LNCS 495, pp. 308-324, Springer-Verlag, Rostock, Germany, May 1991.

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

[8] A. Black, N. Hutchinson, E. Jul and H. Levy, "Object Structure in the Emerald System," Proceedings OOPSLA '86, ACM SIGPLAN Notices, vol. 21, no. 11, pp. 78-86, Nov 1986.

[9] T. Bloom, "Evaluating Synchronization Mechanisms," Proceedings of the Seventh Symposium on Operating Systems Principles, pp. 24-32, Pacific Grove, CA, Dec 10-12, 1979.

[10] J. van den Bos and C. Laffra, "PROCOL — A Parallel Object Language with Protocols," ACM SIGPLAN Notices, Proceedings OOPSLA '89, vol. 24, no. 10, pp. 95-102, Oct 1989.

[11] J-P. Briot, "Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment," in *Proceedings ECOOP '89*, pp. 109-129, Cambridge University Press, Nottingham, July 10-14, 1989.

[12] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," ACM Computing Surveys, vol. 17, no. 4, pp. 471-522, Dec 1985.

[13] L. Cardelli, "A Semantics of Multiple Inheritance," Information and Computation, vol. 76, pp. 138-164, 1988.

[14] D. Caromel, "Concurrency and Reusability: From Sequential to Parallel," Journal of Object-Oriented Programming, vol. 3, no. 3, pp. 34-42, Sept/Oct 1990.

[15] N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," ACM Computing Surveys, vol. 21, no. 3, pp. 323-357, Sept 1989.

[16] Wm. Cook, W. Hill and P. Canning, "Inheritance is not Subtyping," in *Proceedings POPL '90*, San Francisco, Jan 17-19, 1990.

[17] W.M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept," Software — Practice and Experience, vol. 11, pp. 435-466, 1981.

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

[19] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," Artificial Intelligence, vol. 8, no. 3, pp. 323-364, June 1977.

[20] D.G. Kafura and K.H. Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages," in *Proceedings ECOOP '89*, pp. 131-145, Cambridge University Press, Nottingham, July 10-14, 1989.

[21] D.G. Kafura and G. Lavender, "Recent Progress in Combining Actor-Based Concurrency with Object-Oriented Programming," ACM OOPS Messenger, Proceedings OOPSLA/ECOOP 90 workshop on Object-Based Concurrent Systems, vol. 2, no. 2, pp. 55-58, April 1991.

[22] D. Konstantas, O.M. Nierstrasz and M. Papathomas, "An Implementation of Hybrid, a Concurrent Object-Oriented Language," in *Active Object Environments,* ed. D. Tsichritzis, pp. 61-105, Centre Universitaire d'Informatique, University of Geneva, June 1988.

[23] D. Konstantas, "A Dynamically Scalable Distributed Object-Oriented System," in *Object Management,* ed. D. Tsichritzis, pp. 245-254, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[24] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin and X. Rousset de Pina, "Design and Implementation of an Object-Oriented Strongly Typed Language for Distributed Applications," Journal of Object-Oriented Programming, vol. 3, no. 3, pp. 11-22, Sept/Oct 1990.

[25] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen and K. Nygaard, "The BETA Programming Language," in *Research Directions in Object-Oriented Programming,* ed. B. Shriver, P. Wegner, pp. 7-48, The MIT Press, Cambridge, Massachusetts, 1987.

[26] K.G. Larsen and Liu Xinxin, "Compositionality Through an Operational Semantics of Contexts," in *Proceedings ICALP '90,* ed. M.S. Paterson, LNCS 443, pp. 526-539, Springer-Verlag, Warwick U., July 1990.

[27] 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.

[28] B. Liskov, M. Herlihy and L. Gilbert, "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing," in *Proceedings POPL '86,* St. Petersburg Beach, Florida, Jan 13-15, 1986.

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

[30] R. Milner, "Functions as Processes," in *Proceedings ICALP '90,* ed. M.S. Paterson, LNCS 443, pp. 167-180, Springer-Verlag, Warwick U., July 1990.

[31] D.A. Moon, "The Common Lisp Object-Oriented Programming Language Standard," in *Object-Oriented Concepts, Databases and Applications,* ed. W. Kim and F. Lochovsky, pp. 49-78, ACM Press and Addison-Wesley, 1989.

[32] J.E.B. Moss and W.H. Kohler, "Concurrency Features for the Trellis/Owl Language," in *Proceedings ECOOP '87,* pp. 223-232, Paris, France, June 15-17, 1987.

[33] C. Neusius, "Synchronizing Actions," in *Proceedings ECOOP '91,* ed. P. America, LNCS 512, pp. 118-132, Springer-Verlag, Geneva, Switzerland, July 15-19, 1991.

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

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

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

[37] O.M. Nierstrasz and M. Papathomas, "Towards a Type Theory for Active Objects," ACM OOPS Messenger, Proceedings OOPSLA/ECOOP 90 workshop on Object-Based Concurrent Systems, vol. 2, no. 2, pp. 89-93, April 1991.

[38] O.M. Nierstrasz, "Towards an Object Calculus," in *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing,* ed. M. Tokoro, O.M. Nierstrasz, P. Wegner, LNCS, Springer-Verlag, Geneva, Switzerland, July 15-16, 1991, to appear.

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

[40] M. Papathomas and O.M. Nierstrasz, "Supporting Software Reuse in Concurrent Object-Oriented Languages: Exploring the Language Design Space," in *Object Composition*, ed. D. Tsichritzis, pp. 189-204, Centre Universitaire d'Informatique, University of Geneva, June 1991.

[41] M. Papathomas, "Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming," Ph.D. thesis, Dept. of Computer Science, University of Geneva, 1992, forthcoming.

[42] B. Pernici, "Objects with Roles," Proceedings ACM-IEEE Conference of Office Information Systems (COIS), Boston, April 1990.

[43] R.K. Raj and H.M. Levy, "A Compositional Model for Software Reuse," in *Proceedings ECOOP '89*, pp. 3-24, Cambridge University Press, Nottingham, July 10-14, 1989.

[44] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," Proceedings OOPSLA '86, ACM SIGPLAN Notices, vol. 21, no. 11, pp. 38-45, Nov 1986.

[45] J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[46] C. Tomlinson and V. Singh, "Inheritance and Synchronization with Enabled Sets," Proceedings OOPSLA '89, ACM SIGPLAN Notices, vol. 24, no. 10, pp. 103-112, Oct 1989.

[47] A. Tripathi and M. Aksit, "Communication, Scheduling and Resource Management in SINA," Journal of Object-Oriented Programming, vol. 2, no. 4, pp. 24-36, Nov/Dec 1988.

[48] T. Watanabe and A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language," Proceedings OOPSLA '88, ACM SIGPLAN Notices, vol. 23, no. 11, pp. 306-315, Nov 1988.

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

[50] P. Wegner and S. B. Zdonik, "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like," in *Proceedings ECOOP '88*, ed. S. Gjessing and K. Nygaard, LNCS 322, pp. 55-77, Springer Verlag, Oslo, August 15-17, 1988.

[51] P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," ACM OOPS Messenger, vol. 1, no. 1, pp. 7-87, Aug. 1990.

[52] Y. Yokote and M. Tokoro, "Concurrent Programming in ConcurrentSmalltalk," in *Object-Oriented Concurrent Programming*, ed. A. Yonezawa, M. Tokoro, pp. 129-158, The MIT Press, Cambridge, Massachusetts, 1987.

[53] A. Yonezawa, E. Shibayama, T.Takada and Y. Honda, "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1," in *Object-Oriented Concurrent Programming*, ed. A. Yonezawa, M. Tokoro, pp. 55-89, The MIT Press, Cambridge, Massachusetts, 1987.