This is the published version of the publication, made available in accordance with the publisher’s policy.

The Next 700 Concurrent Object-Oriented Languages: Reflections on the Future of Object-Based Concurrency

Nierstrasz, Oscar

How to cite


This publication URL:  https://archive-ouverte.unige.ch/unige:158090

© This document is protected by copyright. Please refer to copyright holder(s) for terms of use.
The Next 700
Concurrent Object-Oriented Languages

Reflections on the Future of Object-Based Concurrency

Oscar Nierstrasz

Abstract

There has been a flurry of activity in recent years to extend existing languages with object-oriented features, and to extend object-oriented concepts and languages with seemingly orthogonal features, such as concurrency and persistence, to improve their expressive power and potential as a solution to the "software crisis". In many cases these integration efforts have uncovered various forms of semantic interference between features. We claim that the majority of these difficulties are concerned with the very aspect of object-orientation that we seek most urgently to exploit, namely software *compositionality*. We shall review the problems of integrating concurrency and object-oriented features from this viewpoint and discuss some of the more important requirements to be met. Finally, we propose a view of objects as *patterns* of communicating agents that suggests the development of a class of concurrent object-oriented languages parameterized by patterns that address the needs of particular application domains.

1. Introduction

During the short history of object-oriented technology dating from Simula-67 [7] to the present, the key breakthrough leading towards its recognition as an important potential force in improving the effectiveness of software engineering was achieved by the Smalltalk project [27][28], which demonstrated that objects can be useful not only for modelling and simulation, but they also provide a very powerful paradigm for software composition and reuse. Since then, there has been a continuing effort to raise object-oriented programming from its perceived status as an novel technique limited to rapid prototyping to that of a credible technology for production software development.

In addition to various improvements to the performance of object-oriented languages and systems, there have been equally many efforts to extend existing programming languages with object oriented features to improve their effectiveness, and many efforts to extend object-oriented languages to address various issues such as strong-typing, concurrency, distribution, persistence and database (set-oriented) functionality. We are left with the impression, however, that many of these efforts at conceptual integration have concentrated on improved *expressive power* as the primary goal rather than on supporting software composition and reuse. In fact, many of the more difficult problems that have been encountered are concerned with the semantic interference between *computational* aspects of objects and mechanisms for composition and reuse, such as classes and inheritance. Issues that at first glance appeared to be orthogonal, such as inheritance and concurrency, can be in fact quite difficult to integrate cleanly.

1. Submitted for publication.
One lesson to draw from these experiences is that object-oriented programming is not a panacea. Not only does it not necessarily help to “add objects” to your favourite programming language, but it is even difficult to add them in such a way that they might help! A second, and perhaps more important, lesson is that we seem to have lost sight of why we are looking towards object-oriented mechanisms as a way of improving our programming environments: we are still placing too much emphasis on programming rather than on software composition as the primary means to developing applications. Reusability of software components, of designs, of requirements models and specifications, of experience is sufficiently poorly understood today that most purported object-oriented analysis, design and development methodologies pay little or no attention to design for reuse. Only recently has it been recognized that we must take a very different view of the software life-cycle, of software management and evolution, and of the economics of software development, if we are to achieve “frameworks” of truly reusable and composable software [25][20][44].

Although the topic of a methodology for the development of reusable object-oriented software is beyond the scope of this discussion, we claim that the designer of the programming languages must pay particular attention to the integration of mechanisms for software composition with other language features if we are to expect any reusable software to ever be written. To this end, we propose a view of concurrent object-oriented languages in which objects can be seen as autonomous communicating agents, and in which the fundamental mechanisms for software composition are “patterns”, which are essentially functions for building objects. Such languages would integrate object-oriented and functional programming in that applications can be seen as functionally composed configurations of communicating objects. Not only can “frameworks”, or libraries of compatible patterns, be defined to address various application domains, but the language primitives for defining objects, such as classes, functions, procedures, control structures and mechanisms for communication and synchronisation can also be viewed as patterns. Furthermore, by varying the set of basic patterns according to the special needs of a given application domain, we effectively can provide a number of different, but interoperable, languages, each obeying the compositional and behavioural rules of its basic patterns.

We shall start by reviewing some of the problems in integrating various language features, taking as a particular example Hybrid, an experimental concurrent object-oriented language developed at the University of Geneva [51][36][57]. We shall then develop some requirements for the design of concurrent object-based languages that specifically address software reuse. On the basis of these requirements we shall propose a view of objects as patterns of communicating agents. We conclude with some remarks on directions for further research.

2. Orthogonality and Interference of Object-Oriented Dimensions

As we seek to understand and characterize trends in the design of object-oriented languages, it is essential for us to pin down both what features of these languages are of interest and why they are important. To this end it is helpful to begin with a characterization of the various features that object-based languages may exhibit and which combinations of features define interesting language classes. Wegner [66][68] identifies 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. To this list we can add \textit{homogeneity}, the principle that "everything" must be an object: in particular, classes are seen as instances of \textit{metaclasses}, which is especially important to support reflection and evolution in the context of persistence.

By adding features in a particular order, we can identify some interesting language classes:

- \textit{Object-Based}: objects encapsulate a collection of services accessible by a "message-passing" interface; typically, objects have an identity and a mutable state (though there are languages that support "functional objects")

- \textit{Object-Oriented}: objects can be instantiated from classes; subclasses can be defined by inheritance

- \textit{Fully Object-Oriented}: classes are also objects; new classes may be created at run-time

- \textit{Strong-typed}: all expressions are guaranteed to be type-consistent (run-time type-checking may be required)

- \textit{Concurrent}: several objects can service requests concurrently; a single object may also service several requests concurrently

- \textit{Persistent}: the lifetime of an object may be longer than a single user session

At this point it is especially important to distinguish between the features concerned essentially with objects, and those concerned essentially with expressive power and run-time issues. We can see then that objects, classes and inheritance are the basic concepts of object-oriented programming and that homogeneity, concurrency and persistence are concerned with expressive power. Strong typing falls in between, as a means to increase one's confidence in the use of objects.

In fact, if we consider the \textit{intent} of object-oriented mechanisms, we can say that the essence of OOP is \textit{encapsulation} (objects), \textit{instantiation} (by classes, prototypes etc.) and \textit{incremental modification} [67] (by inheritance, delegation, genericity etc.). These mechanisms, then, support both decomposition of applications into cooperating objects through encapsulation, and composition and reuse by instantiation and incremental modification.

Our interest in object-oriented technology is motivated by a desire to exploit these mechanisms in the construction of reactive applications, where concurrency, persistence and evolution are critical issues.

\subsection*{2.1 Hybrid}

\textit{Hybrid} is a concurrent object-oriented programming language that attempts to integrate cleanly the features listed above [51]. In particular, the prototype implementation provides [36]:

- objects + classes + inheritance

- strong typing + genericity

- concurrency + persistence

Features planned in the design but not yet implemented include:

- homogeneity + distribution (see [37])
Hybrid is based on a model of active objects. Passive objects have no means to synchronize concurrent requests: each concurrent request causes a particular method to be activated and may interfere with the servicing of other requests. Active objects, by contrast, 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 [43] and internal concurrency.

Active objects in Hybrid are called domains. Each domain is an autonomous quasi-concurrent "top-level" object. 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). Domains permit recursive calls as such requests are perceived as belonging to the same "activity" (i.e., logical thread of control). 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 have no return value.

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 (see 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.

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.
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 cleanly 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; furthermore, 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 [54].

- **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 [34].
• *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 [52][53]. At the same time, it is essential to be very clear about the informal goals and requirements for language design [58] and to study the problems of semantic interference from these viewpoints.

### 2.2 Semantic Interference

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 object model, in which the client/server relationship is that of message-passing between objects, and the composition model, which supports very different client/server relationships, such as inheritance between classes and containment for instance variables.

#### 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 parent classes [60][59] (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:

- providing 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.)

- C++ "friends", which allow one to define different kinds of client [61]

- mechanisms to extend inherited behaviour in a controlled fashion, such as the *inner construct* of Beta [39] and the "before" and "after" methods of CLOS [48].
Types vs. Encapsulation

Through types we mean to express the abstract interface of objects. Type-checking is a means of increasing 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 (such as Hybrid) that allow one to bind variables to any instance that satisfies the type constraint of the variable.

There is a nice theory of polymorphic types restricted to functional objects [14]. Unfortunately there are certain anomalies that appear when we consider mutable objects [21]. In particular, it is not possible to refine “attributes” (which may be modelled by separate “get” and “put” operations), as a client may both want to use an attribute and to modify it. (A refined attribute cannot be updated to a less-specific client-supplied value!)

Types vs. Inheritance

Most difficulties with types and inheritance are due to a confusion between types and classes. If we view types as interface specifications, then it is clear that classes unrelated by inheritance may share a common interface.

By the same token, if it is possible for a subclass to “hide” part of an inherited interface, then there need not even be a 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 [19]. In Hybrid, as in POOL-I [4], types and classes are separate concepts.

Concurrency vs. Encapsulation

As we have seen already in Hybrid, our view of objects as encapsulations of service and state must be revised 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 request scheduling (the ability to delay servicing of requests based on an object’s current state and on the nature of the request) and reply scheduling (the ability of a client to switch its attention to another task while a request to a remote object is pending) [58]. In principle, request and reply scheduling should be transparent to, respectively, clients and servers, as well as to “nearby” objects. In Hy-
brid, however, we have seen that objects making use of delay queues and delegated calls cannot be used with impunity as instance variables, as the use of these mechanisms affects the entire domain. As a consequence, either the encapsulation of the domain is violated, or that of the instantiated object is compromised by the need to examine implementation details to guarantee correct functioning of the application.

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 [34].

Various approaches to address this problem essentially make the compositional (i.e., inheritance) interface more explicit. Such approaches are either based on additional parameterization, such as the inner construct of Beta [39], and “before” and “after” methods of CLOS [48], or on some abstraction of the synchronization policy based on abstract states [34][63].

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 [20]. To this end, we must be able to characterize certain temporal and concurrent properties that can affect the various clients of an object:

- **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 [43] and send/receive/reply [24]
- **Multiple concurrent clients**: objects capable of interleaving interactions with multiple concurrent clients may not work correctly if placed in a sequential environment, and, conversely, clients may be broken if they expect a non-interleaving behaviour

Notions of “contracts” between objects and their clients appear to be very promising [29] as do explicit distinction between the message-passing and compositional interfaces [59]. Understanding plug-compatibility in terms of the visible interactions an object may exhibit appears promising as a means to abstract away from implementation details [52][53]. Both modal logics [41] and linear logic [23][26] appear useful as formalisms for expressing abstract properties of concurrent systems, though it is not clear whether they will prove appropriate for reasoning about compositionality.
3. Requirements for Reusability

As stated at the beginning of the previous section, we can characterize the essence of object-oriented languages in terms of support for encapsulation (objects), instantiation (classes, etc.) and incremental modification (inheritance, etc.). These mechanisms essentially address issues of decomposition (i.e., organization), and composition (i.e., reuse). If we consider the various design alternatives for supporting concurrency in object-oriented languages, it is possible to identify some criteria for evaluating the degree to which a particular approach is consistent with object-oriented software composition. We shall summarize from our observations reported in [58].

It is helpful to distinguish approaches to concurrency according to the following classification scheme: (1) Object Models: How is object consistency maintained in the presence of concurrency? (2) Internal Concurrency: What means do objects have to manage multiple threads of control? (3) Client-Server Interaction: What degree of control do objects have over the interaction protocol?

3.1 Object Models

As a first cut, concurrent object-based languages can be distinguished according to the kind of object model they support [56][58]:

- **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][27][49].

- **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 [15][34].

- **The Homogeneous Approach**: All objects are "active" and have control over the synchronization of concurrent requests [3][63].

The orthogonal approach suffers from the various difficulties indicated earlier in Section 2. 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 [42], 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, as is the case with Hybrid [57].

3.2 Internal Concurrency

Next, we can make distinctions according to the means for coping with internal concurrency [66]:
• **Sequential Objects** have a single active thread of control [1][3][70].

• **Quasi-Concurrent Objects** explicitly interleave multiple threads of control (as in Hybrid).

• **Concurrent Objects** may be internally concurrent. We can further distinguish the cases in which thread creation is either:
  - *Unconstrained*: as in the orthogonal object model
  - *Controlled*: in which case must objects explicitly create a new internal thread (for example, using the become primitive of actor languages [2]).

Purely sequential objects that follow a strict RPC protocol overly restrict potential concurrency as they are unable to switch attention to other tasks while waiting for the reply to a remote request [43]. Additional concurrency can only be introduced by delegating tasks to proxy objects (e.g., Cboxes [69] and future variables [70]), 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. Unconstrained concurrency has the disadvantages of the orthogonal object model. Controlled 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.

### 3.3 Client-Server Interaction

Finally, we can distinguish languages according to the forms of client/server interaction supported. The main concerns (introduced above) 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 client’s point of view, the most important distinction is the interaction protocol supported:

* One-way Message-Passing: higher-level protocols are explicitly programmed [9].
* 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 [51] or concurrent [64], as described above.
  - **Proxies**: sending of requests and receipt of replies may be delegated to (external) proxy objects [15][69][70].

Reply scheduling is essentially concerned with three issues: (1) the interleaving of 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 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.

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][27][49].
- **Conditional acceptance**:
  - **Explicit acceptance**: An explicit `accept` statement is used to indicate which requests will be served [1][3][70].
  - **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. [12][65].
  - **Activation conditions**: Implicit or explicit conditions govern the acceptance of requests.
    - **Representation specific**: Conditions are expressed in terms of the object’s internal state [38][51][64].
    - **Abstract**: Message acceptance depends on the abstract state of an object [9][34][63].

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. In Hybrid, for example, this can only be done after accepting the request by making a delegated call to another (trigger) object that delays the activity associated with the request until the awaited precondition is satisfied [51]. The main difference between these approaches from the point of view of reusability is the extent to which the synchronization policy of the server can be understood in abstract terms to facilitate composition and incremental modification. In this respect the approach of abstract states (whether by means of enabled sets [63], behaviour abstraction [34][35] or protocol expressions [9], 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.

### 3.4 Some Requirements for Reusability

We can sum up our basic requirements as follows:

- **Mutual exclusion**: reusability is enhanced if the internal states of all objects are automatically protected from concurrent invocations, as in the homogeneous model of active objects.
- **Internal concurrency**: it should be possible to replace a sequential implementation of an active object by a concurrent one without affecting clients; it should be possible to properly encapsulate internal concurrency, i.e., without resort to external proxies.
• Reply scheduling transparency: a client should be able to switch its attention to another
task while waiting for a reply to an arbitrary request, i.e., transparently with respect to
the server.

• Request scheduling transparency: objects should be able to delay servicing of arbitrary
requests based on their current state and on the message contents, i.e., transparently with
respect to the client.

• Incremental modification: the synchronization policy of object classes must be suffi-
ciently abstract that it will be possible to extend "inherited" behaviour without reimple-
mentation. When previously defined classes are discovered not to be cleanly extensible
due to insufficient abstraction, it should be possible to define an equivalent implementa-
tion that is extensible in the desired way (possibly requiring reorganisation of the class
hierarchy [17][18][25]).

It seems, then, that we are looking for a homogeneous model of active objects that supports
internal concurrency (and hence hierarchical composition of active objects), a client-server in-
teraction protocol that is by default one of remote procedure calls, but which provides both cli-
ents and servers transparent control over the scheduling of requests and replies, and an abstrac-
tion mechanism that allows one to state explicitly the ways in which reusable software compo-
nents may be extended and composed.

4. The Next 700 Concurrent Object-Oriented Languages

We believe that concurrency and software composition are important enough to expect that fu-
ture generations of languages will be based on active objects. But what will such languages look
like, and what will distinguish them from one another?

There are essentially three sets of issues to take into consideration (see Figure 4):

1. Computational Model: at this level we are concerned with the semantics of concurrency,
communication, encapsulation and instantiation. Viewing computation in terms of the
interactions of systems of communicating concurrent agents is particularly appropriate.

2. Language Primitives: at the language level we identify the particular behavioural pat-
terns of interest, and to these we assign the syntactic patterns that will be the constructs
of the language. For instance, here we choose to view active objects as particular pat-
terns of communicating agents. The primitives we choose here should address the prob-
lems and requirements identified in the previous section.

3. Software Components: finally, in terms of the language primitives provided, we are able
to define software components that can be composed to produced running applications.
Reusability is measured in terms of the degree to which applications can be built out of
standard (library) components rather than user-defined components.

Within this scenario, even under the constraints we have listed so far, it is easy to imagine a vast
array of different languages each settling on a different set of primitive patterns to address a dif-
ferent problem domain. Clearly such a situation would discourage reuse, as the proliferation of
Oscar Nierstrasz

Compositional level: libraries of software components

Language level: standard syntactic patterns (active objects etc.)

Computational level: patterns of communicating agents

Figure 4 Viewing Application Development in Terms of Hierarchies of Patterns

languages introduces interoperability problems between libraries written with different ground rules in mind. (This is in fact the situation today, as application programmers struggle to make database applications, user interface toolkits and expert system shells talk to each other.)

We argue instead for a slightly different approach, in which a common computational model for active objects is combined with a pattern mechanism that effectively allows one to define a family of languages (in the sense of Peter Landin’s ISWIM proposal [40]) each characterized by a different set of standard patterns. Although we shall not here attempt to define a “pattern language for active objects”, we will at least give some hints as to what such a language might look like by illustrating how active objects can be viewed as patterns of communicating agents, and by listing a number of features that a pattern language should support to satisfy our requirements for integrating concurrency and software composition.

4.1 Viewing Objects as Patterns of Communicating Agents

We shall take the position that active objects are naturally modelled as communicating agents, in the spirit of in the spirit of CCS [45][46][47] and CSP [13][33]. Although recently several researchers have proposed alternative formalisms [6][10][30][47][50][62] that deal with such issues as communication of names and of agent specifications, equivalence between agents and unification with lambda calculus in different ways, what they all have in common is a well-defined formal framework for modelling concurrent systems in which one can reason in terms of externally visible behaviour of agents as opposed to their implementation. Agents with the
"same" external behaviour are equivalent (i.e., equal as specifications); agents that conform to a common protocol are "plug-compatible" [52][54].

**Abacus — a notation for specifying active agents**

For the purpose of this discussion, we shall adopt here the concept of agents provided by *Abacus* [53], an executable dialect of CCS that provides some additional primitives especially suited to modelling active objects. (The operational semantics of Abacus and a comparison with CCS is presented elsewhere [55].)

An agent is an entity capable of communicating with other agents. Agents may change state as a result of an external (communication) event or as a result of an internal event. Since we are primarily interested in the externally visible behaviour of an agent, by changing "state" we only mean that the behaviour of an agent may change as a result of either external or internal events. The terms "agent", "state" and "behaviour" are in a sense interchangeable, and only indicate a point of view. (I.e., an agent that "changes its state" also "changes its behaviour" or "becomes a new agent".)

At any point in time an agent may be capable of a variety of external or internal events. Internal events can, in general, be understood as internal communications within an agent that encapsulates a system of agents. An agent not capable of any internal event (in its current state) is stable [46]. An agent not capable of any events at all is dead. (The prototypical dead agent is called nil.) An agent expresses its desire to communicate by making an input or an output offer for given set of possible communications (messages). An event may take place if two agents make matching input and output offers. In general, in any given system of agents there may be many possible events that can take place, which is a source of both concurrency and non-determinism.

In Abacus, an agent *a* that offers to input the message *m* and become *b* would be expressed: \(a::m?b\). Another agent *c* that offers to output *m* and become *d* would be expressed: \(c:=m!d\). The concurrent composition of *a* and *c* is written: \(a&c\). In this system, only the communication event *m* is possible, yielding the replacement system \(b&d\).

We can specify both remote and local communication between agents. In Abacus *a::m* stands for the agent that results after *a* accepts *m* as input. *a* is not permitted to communicate with any other agent until it accepts *m*; if it is not willing to do so, it is dead. (For example, the agent \(\text{nil::m}\) is equivalent to \(\text{nil}\).) Local communication is analogous to application of arguments to functions, and will be used in this way later on.

A choice of offers is expressed (as in CCS) by the summation symbol: \(m?b + n?a\) describes an agent that will either input *m* and become *b* or input *n* and become *a*. (In Abacus this is the external choice operator of TCCS [22] — the agent simultaneously makes both offers.)

Both agents and messages may be parameterized. In Abacus identifiers beginning with upper case letters are variables (as in Prolog) and all others are literals. For example, we may represent the primitives of Linda [16] as follows:

\[
\begin{align*}
\text{linda} & := \text{[out,T]} \, ? \, (\text{linda} \, & \, \text{tuple(T)}). \\
\text{tuple(T)} & := \text{[in,T]} \, ! \, \text{nil} \, + \, \text{[rd,T]} \, ! \, \text{tuple(T)}.
\end{align*}
\]
Here Linda is an agent that will accept a message \[\text{[out,T]}\] for any value \(T\) (i.e., sent by an agent wishing to post the tuple \(T\) to the Linda "blackboard"), and replace itself by a copy of itself concurrently composed with a copy of \(\text{tuple}(T)\). This latter agent attempts to deliver either a \([\text{ln},T]\) message, which would be a destructive event, or a \([\text{rd},T]\) message, which would be non-destructive. (I.e., these messages would be delivered to agents attempting to remove or to read tuples on blackboard.) There may be many concurrent copies of the same \(\text{tuple}(T)\) agent in the same system. If there are also multiple agents concurrently competing for a single \(\text{tuple}(T)\), the successful event is defined non-deterministically.

**SCOL — A simple concurrent object language**

In order to specify programming language constructs, we must now identify the behavioural patterns of interest as parameterized agents. Let us consider a very simple hypothetical language that we shall call SCOL (for simple concurrent object language), whose grammar is as follows:

- `<initial statement>` := Initially `<statement>`
- `<classdef>` := `class <class name> [(<name list>)] is`<br>`{ (<selector>[(<name list>)] => `<statement>` ; ) }`
- `<statement>` := `skip` | `<expression>` | `<statement>`;<`statement>` | `return <expression>` | `become self` | `become <class name>[(<expression list>)]` | `if(<boolean>) then `<statement>` [ else `<statement>` ] | `let <binding list> In `<statement>`` | `spawn `<statement>`
- `<expression>` := `<number>` | `<name>` | `<expression list>` | `<expression> + `<expression>` | `<expression> - `<expression>` | `<expression> * `<expression>` | `<expression> / `<expression>` | `<name>` <= `<selector>[(<expression list>)]` | `self <= `<selector>[(<expression list>)]` | `new `<class name>[(<expression list>)]`
- `<binding>` := `<name>` = `<expression>`
- `<boolean>` := `<expression>` = `<expression>`
- `<class name>` := `<name>
- `<selector>` := `<name>`
- `<name list>` := `<name>` [ , `<name>` ]`
- `<expression list>` := `<expression>` [ , `<expression>` ]`
- `<binding list>` := `<binding>` [ , `<binding>` ]`

A SCOL program is a collection of class definitions and an initial statement. `<name>` and `<number>` are terminals. SCOL is loosely modelled after SAL, a simple actor language [2], but is based on synchronous message-passing (whereas actors communicate asynchronously) and supports a request/reply communication protocol. For example, in SCOL we may define the standard actor example of a recursive factorial server [32] as follows:

```plaintext
class recfact lsfact:n => become(recfact);
  if(n=0)
    then return(1)
  else return(n*(self<=fact:(n-1))) ;
Initially let r = new recfact in spawn(r<=fact:5); r<=fact:4
```

The class `recfact` is instantiated by means of the `new` statement and its identity is bound to the local name `r`. The `spawn` construct causes the enclosed statement to execute concurrently with whatever follows. In this case `r` will receive concurrent requests to compute the factorials of 4 and of 5. `r` will answer immediately the request is for the factorial of 0, otherwise it first sends itself a request to compute the factorial of `n-1` before responding with the correct result. A SCOL object busy servicing a request will not accept further requests until it has replied, unless a `become` statement is executed. This causes the object to become ready for another request while the servicing thread continues. Since `recfact` is defined recursively, it must use a `become` statement if it is not to deadlock. (Not really a desirable feature, but then SCOL is a toy language!)

In fact, `recfact` would be better defined as a purely functional object as it carries no state (in actor terminology, it would be a non-serial actor since it has no need to serialize its requests).

Perhaps the simplest example of a mutable object is:

```plaintext
class var(val) lsget => return(val) ;
    put:newval => become(var(newval)) ; return(newval) ;
```

Slightly more interesting is a one-slot buffer:

```plaintext
class empty lsget => become(empty) ; return(val) ;
class full(val) lsget => become(full(val)) ; return(val) ;
```

**Mapping SCOL to Abacus**

The constructs of a language like SCOL can be mapped in a fairly straightforward way to our computational model of communicating agents. There are essentially four different kinds of agents: classes, objects, statements and expressions; and there are three kinds of message: `[val,V]` for local communications (i.e., within an object), and `[call,Old,Request,Cld]` and `[reply,Cld,Value]` for remote communications (i.e., between objects).

An object is an agent with a unique global identifier (`Old`) that accepts messages of the form `[call,Old,Request,Cld]` and eventually replies (if all goes well!) with `[reply,Cld,Value]`. `Cld` uniquely identifies the client expression. An object is parameterized by its body, which is a set of simultaneously available services:

```plaintext
object(S1;S2) := object(S1) + object(S2) .
```

For each service, the object must know its own `Old` and its class, `Self`, which it must `become` after the reply has been generated. When a request is accepted, the object turns into an agent that executes the associated statement:

```plaintext
object(Stmt => Request) := [Old,Self] ? [call,Old,Request,Cld] ? stmt(Stmt,Id,Self,Cld) .
```

A class is simply an agent that, when supplied with an `Old`, instantiates an object:

```plaintext
class(Self,Body) := Old ? object(Body)::[Old,Self] .
```

There are two kinds of statement: services and actions; and there are two varieties of each: dependent and independent. A service is an agent that is obliged to deliver a reply to some client. An action is not. A dependent statement is obliged to `become` an object again when it has terminated, and an independent statement is not. We represent all statements by the parameterized
agent \texttt{stmt(Stmt,ld,Self,Client)}, where \texttt{Client} is \texttt{noclient} for actions and \texttt{Self} is \texttt{nil} for independent statements.

The \texttt{skip} statement is the easiest to define. When followed by another statement, it is equivalent to that statement:

\[
\texttt{stmt((skip;Stmt),Old,Self,Client)} := \texttt{stmt(Stmt,Old,Self,Client)} .
\]

When not so followed, however, it is defined only for actions (a service terminating with a \texttt{skip} would fail in its obligation to reply to its client):

\[
\texttt{stmt(skip,Old,Self,noclient)} := \texttt{Self::Old} .
\]

A \texttt{become} statement results in the splitting of a statement into an object and an independent statement:

\[
\texttt{stmt((become(B);Stmt),Old,Self,Client)} := B::Old \& \texttt{stmt(Stmt,Old,nil,Client)} .
\]

A special case occurs when the \texttt{become} statement is the last statement of an action (as with \texttt{skip}, a \texttt{become} statement may not terminate a service):

\[
\texttt{stmt(become(B),Old,Self,noclient)} := B::Old .
\]

A \texttt{spawn} statement, conversely, splits a service into an independent action and a service:

\[
\texttt{stmt((spawn(S);Stmt),Old,Self,Client)} := \texttt{stmt(S,Old,nil,nil,client)} \& \texttt{stmt(Stmt,Old,Self,Client)} .
\]

Expressions are computed by agents of the form \texttt{expr(Expr,Econt,Old)} that evaluate \texttt{Expr} and pass the value to the expression continuation \texttt{Econt} waiting for the result. (\texttt{Old} is also a parameter since expressions of the form \texttt{self \langle= Message} require the expression agent to know the \texttt{Old} of the object with which it is associated.)

The simplest expressions to evaluate are:

\[
\texttt{expr(Val,Econt,Old)} := \texttt{Econt::[Val,Val]} .
\]

where \texttt{Val} is a name or a number. A call to another object is generated as follows:

\[
\texttt{expr(X\langle= E, Econt,Old)} := \texttt{expr(E, call(X,Econt),Old)} .
\]

\[
\texttt{call(X,Econt)} := [\texttt{val,M}] \? [\texttt{id,Cid}] \? [\texttt{call,X,M,Cid}] ! [\texttt{reply,Cid,Reply}] ? \texttt{Econt::[val,Reply]} .
\]

The unique \texttt{Cid} or \texttt{Old} is generated by a special agent\footnote{We are cheating here slightly. A better way of handling the creation of new names is by means of so-called “alpha-conversion” rules (as in the lambda calculus \cite{lambda}) together with rules governing the migration of names between scopes \cite{scope}} for this purpose:

\[
\texttt{newld(Id)} := [\texttt{id,Id}] \! \texttt{newld(Id+1)} .
\]

A reply is generated by means of a \texttt{return} statement:

\[
\texttt{stmt(return(E),Old,Self,Client)} := \texttt{expr(E,reply(Self::Old,Client),Old)} .
\]

\[
\texttt{reply(Cont,Client)} := [\texttt{val,R}] \? [\texttt{reply,Client,R}] ! \texttt{Cont} .
\]

An early reply is possible, in which case the continuation is an action (i.e., with no client):
stmt((return(E);Stmt),Old,Self,Cllent) :=

expr(E,reply(stmt(Stmt,Old,Self,noclient),Cllent),Old).

The remaining agents that define the operational behaviour of SCOL are specified in a similar fashion.

4.2 Towards a Pattern Language for Active Objects

We have seen how programming language constructs for defining active objects can be viewed in terms of patterns of communicating agents. But the toy language we have used to illustrate this principle is a long way from meeting our requirements for a language integrating concurrency and object-oriented software composition.

What would such a language look like? What form of active objects would it provide? What kind of encapsulation mechanism is required to support fully component-oriented software development? Can a single framework provide the basis for a “family” of interoperable languages addressing different problem domains?

We propose an approach in which primitives for defining active objects are augmented by a “pattern” mechanism that allows one to define software components as higher-order functions (patterns) over objects. Patterns are intended to generalize the software composition mechanisms of object-oriented languages by viewing all such mechanisms in terms of functional composition of software templates. This view is supported by recent proposals that “unbundle” inheritance into explicit wrappers [31] or composable mixins [11] that provide greater control over software composition.

Let us first consider the characteristics of active objects. First and foremost, they are objects, and so they encapsulate both service and state. As we have previously argued, objects should support a request/reply protocol, in which every accepted request will eventually result in a reply (if the request cannot be fulfilled, the reply may be an exception). Similarly, an object making a request is obliged to eventually accept the reply (including exceptions, in which case the client should be prepared to repair the damage). Objects have a unique name through which their services can be requested (several objects with the same name constitute a single, concurrent object). Objects can be dynamically instantiated, which results in the generation of a new name.

Objects are active, which is to say that they are free to decide when to accept communications from other objects. Active objects may exhibit non-uniform service availability (such as the single-slot buffer above). They are also active in the sense that they may be internally concurrent. Whereas the objects of SCOL could be composed of several concurrent statements, in general objects may contain other objects, which implies the need for nested environments and appropriate scope rules. (The scope rules are especially critical, for software composition is essential an activity of binding free names to values and behaviours.) As we have seen with SCOL, the various parts of an active object exhibit different kinds of behaviour (the duty to respond to client or not, etc.). The primitives for defining these parts are the basic patterns of the language that determine what kinds of objects and components we can build and which configurations we can describe. For example, in SCOL we cannot describe configurations with pending requests (though of course such configurations are reachable from the initial ones).
Whereas the basic patterns of a language raise the level of computation from that of agents to active objects, software components raise the level further to that of particular problem domains. But how does one define “software components”? We have claimed that a single, general-purpose pattern mechanism should suffice, and in fact this is not hard to see if we consider that software composition is essential functional composition over the domain of software descriptions. By combining primitives for defining active objects with such a pattern mechanism, we effectively integrate the functional and object-oriented paradigms so that objects constitute the base paradigm for modelling concurrent, reactive applications, and functions (patterns) provide the compositional paradigm at the meta-level. The difference between patterns in a pattern language and functions in a pure functional language is that patterns may evaluate to configurations of agents rather than just to pure values or other functions.

But what exactly are patterns? The parameterized agents of Abacus are an example. Their parameters may stand for either communicable values or behaviour expressions. In a higher-level language, pattern parameters can stand for values (to initialize newly instantiated objects), methods and instance variables (to extend the behaviour of a class), classes (as in generics), or statements and expressions (as with inner, and before and after methods).

At first glance it would seem that generalized parameterization of objects is all that is needed, but the problem is actually more subtle than that. As we have hinted earlier, software composition is essentially concerned with binding of names, so it should be no surprise that we must be especially careful as to how names are treated. In general, we can distinguish between "pure values" (i.e., literals), agent variables bound by communication, pattern variables bound by application, and agent and pattern names bound by the environment. In the presence of nested environments we must be careful to avoid a confusion of internal and external names. An object, for example, must be able to distinguish between an internal object called x and an external one also called x. If it learns the name of the public x after the fact, one approach is to rename the private one to some new, unique x' [47]. But if we are to extend a class with new methods, we should be able to refer to the existing instance variables without renaming. That is, sometimes we will want patterns to be like functions, and sometimes like macros.

To further complicate (or simplify) things, there is a very close connection between communication and application as seen by this simple example:

```
a(X) := X ! a(X).
b := X ? X ! b : X.
```

Here a(m) and b::m denote the same visible behaviour (namely m!m!m!...), except that in the first case we bind X by functional application, and in the second case by communication. This suggests that patterns and agents are birds of a feather. (And hence the search for a unification of functions and processes [10][47][50][62].) What is interesting about this is that patterns need not be immutable. If we view patterns as resources, that is as objects, then they may have an associated state. This would allow us to model so-called reflective computations [63] in which, for example, not only objects but classes may change their behaviour with time. Patterns should therefore not just be considered as an “add-on” to active objects, but as a unifying concept for objects and concurrency.
5. Concluding Remarks

Reusability today is generally oversold and mythologized as a kind of Software Eldorado where old COBOL programs are continually being reborn as Ada packages and object classes. The hard fact is that it is very difficult to integrate object-oriented technology supporting reuse with other essential features concerned with concurrency, persistence and strong typing. From both our own experience implementing Hybrid and from that of other researchers we have identified a number of semantic interference issues between various object-oriented dimensions, and we have postulated a set of requirements for future language to meet that address both issues of expressive and of software composition.

We propose the development of a pattern language for active objects that would support the definition of software components over the domain of active objects. Such a language would integrate the functional and object-oriented paradigms by providing objects as the paradigm for decomposition and organization, and higher-order patterns as the paradigm for software composition and reuse. Furthermore, by varying the basic patterns that define the behaviour of active objects and by varying the set of standard software components available, one effectively has a “family” of interoperable languages to address various problem domains.

Although we have not presented a true pattern language, we have shown how active objects of a hypothetical concurrent object-oriented programming language can be viewed as patterns of communicating agents, and how software components can be viewed as higher-level patterns. We have further conjectured several features of a pattern language for active objects. Our list is far from exhaustive however, and we have not addressed here a host of other important issues such as: fairness, exceptions, interrupts (i.e., “priority” messages), garbage collection, default values for pattern parameters (to allow, for example, wrappers to be used as if they were classes), eager and lazy evaluation of composed patterns (i.e., pattern compilation), or type systems for pattern composition.

In conclusion, we would emphasize that reusability does not come for free. Programming in an object-oriented language not necessarily produce reusable code. Even if we succeed in developing a pattern language for active objects that meets the requirements we have discussed, there is no guarantee that software components written in such a language will be reusable. One must also invest in good tools for organizing and maintaining software libraries, in advanced programming and debugging environments, and in the re-education of programmers and project managers to encourage and promote both software reuse and its development.

References


