Viewing Objects as Patterns of Communicating Agents

Oscar Nierstrasz
Michael Papathomas

Université de Genève
Centre Universitaire d'Informatique
12 rue du Lac, CH-1207 Geneva, Switzerland
E-mail: {oscar,michael}@cui.unige.ch, oscar@cgeuge51.bitnet
Tel: +41 (22) 787.65.80, Fax: +41 (22) 735.39.05

Abstract

Following our own experience developing a concurrent object-oriented language as well of that of other researchers, we have identified several key problems in the design of a concurrency model compatible with the mechanisms of object-oriented programming. We propose an approach to language design in which an executable notation describing the behaviour of communicating agents is extended by syntactic patterns that encapsulate language constructs. We indicate how various language models can be accommodated, and how mechanisms such as inheritance can be modeled. Finally, we introduce a new notion of types that characterizes concurrent objects in terms of their externally visible behaviour.

1. Introduction

The message-passing model of communication in object-oriented languages appears to naturally support concurrent execution of autonomous objects. This fact has led many researchers to try to exploit this autonomy in developing concurrent object-oriented languages [2], [15], [17], [22], [23], [30], [32]. Various forms of active, message-passing objects, and passive, lockable objects have been proposed and implemented. Unfortunately none of these approaches has yet succeeded in resolving basic conflicts between concurrency mechanisms and the encapsulation that is needed for the safe use and reuse of object-oriented code [6], [14], [26], [27], [28].

We claim that the difficulty of the language design problem is due largely to the lack of:

- widely accepted and well-understood models of concurrency
- good tools for prototyping languages
- a good understanding of reuse criteria for encapsulated software

We propose a practical approach to the design of concurrent object-oriented languages using a well-defined computational model of communicating agents based on Milner's CCS [20] and Hoare's CSP [13]. A compact executable notation called *Abacus* [24] characterizes the behaviour of agents in terms of possible communications with other agents. Various object models can be easily captured by varying the mapping between objects and agents, and by varying the communication patterns that may take place. Programming language constructs are designed by mapping syntactic patterns to behavioural patterns. Since these mappings are straightforward translations to an executable notation, this can lead to a rapid prototype of a language interpreter.

In the following sections we will introduce our notation and outline how objects may be modeled. We shall then describe the design space for approaches to object-oriented concurrency, detailing some specific requirements for a consistent model of concurrent objects. We then show how classes, inheritance and genericity can be modelled as patterns of behaviour, and we argue the need for explicit reuse criteria to be associated with concurrent objects. Finally, we demonstrate that signatures are inadequate to express the valid use and reuse of concurrent objects, and we propose a new approach that views types and subtypes as partial specifications of the externally visible behaviour of objects.

2. Viewing Objects as Communicating Agents

We take the position that objects, whether they be "active" or "passive," and regardless of the particular object model of a language, are naturally modeled as communicating *agents*. An agent is an entity that may change state when it communicates with another agent. Complex agents that encapsulate a collection of cooperating agents may also change state due to hidden internal communications. Communication is synchronous, occurring only if there exist matching input and output *offers* to communicate between two agents. Concurrency and reactive behaviour are easily captured in such a model. The model is fully abstract in the sense that agents with the same external behaviour can be viewed as equivalent [20].

We have designed and implemented (in Prolog) an executable notation called Abacus [24], modelled closely after CCS [20] and CSP [13]. A *behaviour expression* specifies the behaviour of an agent or of a system of agents by indicating the current input and output offers of each agent, and the replacement behaviour of the agent should the offer be accepted. A behaviour expression consists of *event names* and *agent names* composed as follows:

e ! p	offer output e with replacement behaviour p
e ? p	offer input e with replacement p
p + q	behave like either p or q (exclusive choice)
p & q	p and q may communicate (concurrent composition)
nil	make no offers

Additionally there are several operators that help to encapsulate systems of cooperating agents. These include *restriction* and *relabelling* to hide or relabel selected offers, and *label prefixing* and *filter-*

ing to define scopes beyond which only "prefixed" offers are visible (for details see [24]). Event names only appear in the context of input and output offers. Wherever an agent name appears in the above, a behaviour expression may be used instead. Finally:

p := *behaviour-expression* .

binds the name p to the expression that follows. The operators ! and ? have the highest precedence, and are followed, in order, by +, & and :=. A formal semantics of Abacus is easily specified by a set of transition rules [24]. These rules are implemented in a straightforward way in Prolog, specifying for any given behaviour expression what events may take place, and what the replacement expression will be. We further exploit Prolog in the examples that follow by using functors as agent names and lists and tuples as event names.

A trivial example of a behaviour expression is:

e?nil & e!nil

This permits the communication event e to take place between the agents specified by e?nil and e!nil, with the net replacement:

nil & nil

Note that nil & nil is equivalent to nil, as nothing further can happen.

In order to define a programming language, we map language constructs to behavioural patterns in a denotational fashion [12]. We use the term *pattern* to refer to any function that evaluates to Abacus agents. The arguments may be arbitrary values, agents or syntactic patterns of the language being defined. To give a flavour of the approach, we provide a few short examples of defining the semantics of statements for a programming language. The stmt pattern takes two arguments: a statement, and the *continuation*, that is the agent that will perform the rest of the computation. For example, a skip statement does nothing, so the semantics of skip is simply that of the continuation:

```
stmt(skip,Cont) := Cont.
```

(We follow the Prolog convention of indicating variables by identifiers with a leading upper-case character.) The semi-colon is a statement separator. The interpretation of S1;S2 is simply an agent that interprets S1 with the continuation being the agent that interprets S2:

```
stmt((S1;S2),Cont) := stmt(S1,stmt(S2,Cont)).
```

Suppose that Boolean expressions are computed by agents that terminate with an output offer of either true or false. We could then specify an if command as follows:

```
stmt(if Bool then S1 else S2,Cont) := expr(Bool) &
                    true?stmt(S1,Cont) + false?stmt(S2,Cont).
```

In this example, the first agent expr(Bool) evaluates the expression Bool and outputs either the value true or false. The second agent consumes the value and becomes the agent that selectively evaluates the appropriate clause of the if command.

For a more complete illustration of the approach, we refer the reader to the Abacus specification of SAL [24], Agha's Simple Actor Language developed to explain the actor model [1].

Within this framework, it is relatively straightforward to express a variety of object models. To this end, it is convenient to model messages as *compound events*, expressed as lists or tuples. As a reference model, let us consider the following restrictions:

- An object is an agent with a unique *identity*. It communicates with other objects by sending *call* or *reply* messages, which are compound events with the receiver explicitly identified.
- *call* messages from a client object with identity Client to a server object with identity Server are of the form [Server,Msg,Client].
- *reply* messages to a client with identity Client are of the form [Client,Reply].

An object with identity Id is characterized by the behavioural pattern accept(Id):

```
accept(Id) := [Id,Msg,Client] ? ( call(Id,Client) + reply(Id,Client) ) .
call(Id,Client) := [Server,Msg,Id] ! wait(Id,Client) .
wait(Id,Client) := [Id,Reply] ? ( call(Id,Client) + reply(Id,Client) ) .
reply(Id,Client) := [Client,Reply] ! accept(Id) .
```

That is, object Id can accept a request from a Client, issue requests to other Server objects, and eventually reply to Client. The pattern accept(Id) is a partial specification of behaviour, since it says nothing about the contents of messages or the other objects used as servers. It only guarantees that no new requests will be accepted while there is a pending request.

A thread of control can be seen as a trace of *call* and *reply* events, with the control at any point being with the object in one of the abstract states call, or reply. Note that the idea of defining objects as agents that conform to certain patterns of behaviour is similar to Minsky's approach of characterizing behaviour by a set of *laws* [21]. By introducing variations in accept(Id) (i.e., by considering different sets of laws), we can express the behaviour of both active and passive objects, multi-threaded objects, asynchronously communicating objects, and objects that make use of a variety of concurrency control mechanisms.

3. A Design Space for Concurrent Object Models

Although encapsulation of single-threaded, passive objects is reasonably well-understood, the same is not true of concurrent objects. Depending on the way that concurrency is handled in a language, encapsulation may be violated in a number of ways. In the simplest case, if we take an object designed for use in a single-threaded application and expose it to multiple concurrent clients, the concurrent execution of methods can compromise the object's internal consistency.

On the other hand, even when an object is able to protect itself from concurrent requests, it may be necessary to expose implementation details in order to protect the integrity of its clients (for example, to avoid deadlock). For a language design to offer a reasonable encapsulation model for concurrent objects, we suggest that at least the following minimal set of criteria should be met:

- ***Protection:*** all objects should be guaranteed of their internal consistency independently of their environment or the presence of multiple threads.
- ***Scheduling:*** an object must be able to selectively refuse or delay certain requests not only on the basis of its internal state, but also depending on the contents of the request message [18].
- ***Interleaving:*** the desired external behaviour of an object should not over-constrain its internal behaviour, for example, internal concurrency should be permitted, as should multiple "readers" for methods that do not cause state changes. Mechanisms for suspending and interleaving threads must not compromise the consistency of "nearby" objects (e.g., enclosing objects, subclass instances).

Protection and scheduling are naturally modeled by agents, as they exercise complete control at all times over the messages they accept. For example, a single-slot buffer with identity b can be trivially specified as:

```
buf := [b,(put,Value),Prod] ? [Prod,ok] !
       [b,get,Cons] ? [Cons,Value] ! buf .
```

A producer p would send [b,(put,Value),p] messages, waiting for the [p,ok] response, and a consumer c would send [b,get,c] messages, picking up the [c,Value] response. More elaborate scheduling of requests can be effected by the use of internal message queues.

Interleaving of threads can be modeled by relaxing the restriction that objects respond before accepting new requests. Internal concurrency is straightforward to model, as we can view complex agents as being composed of more primitive agents.

With these criteria in mind, we may now consider our design space according to following language classes [26], [27]:

1. ***The Orthogonal approach:*** objects and concurrency constructs are independent, as in Smalltalk-80 [11], Emerald [4] or Trellis/Owl [22]. Semaphores, locks or monitors must be judiciously utilized to achieve object protection.
2. ***The Heterogeneous approach:*** both data objects similar to those found in sequential languages and protected "concurrent" objects are supported. The protection of concurrent objects may be accomplished by transactions, as in PAL [3], or by considering such objects as being active, as in the following approach.
3. ***The Homogeneous approach:*** threads are explicitly associated with objects, instead of being an independent programming construct. Hybrid [23], POOL-T [2] and ABCL/1 [32] fall into this category.

Objects conforming to any one of these language classes can be easily modeled by communicating agents simply by varying the synchronization policies observed. For example, objects of the orthogonal class would accept any request at any time, creating an internal agent to perform the method associated with the request. The method agents synchronize by consulting semaphore or lock agents, such as:

```
lock(Name) := [acquire,Name] ? [release,Name] ? lock(Name) .
```

Scheduling and interleaving of threads can be facilitated by introducing asynchrony while either sending or accepting either calls or replies, resulting in a variety of communication styles:

- ***Asynchronous call:*** the client object creates a *messenger* agent that delivers the message; the client is free to continue some other activity.
- ***Message queues:*** the server object contains an autonomous *queue* agent that filters and queues requests; the server can schedule multiple requests.
- ***Asynchronous reply:*** the server object creates a messenger to deliver the reply; the server can immediately switch to another request.
- ***Futures:*** instead of the actual client accepting the reply, it is picked up by a *future* agent [1], which saves the reply until it is needed; if the client asks for the reply before the future has received it, it will block.

These language classes and concurrency mechanisms may be technically equivalent in terms of expressive power, but there are profound differences in terms of convenience when packaging objects for re-use. As a concrete example, the first version of ConcurrentSmalltalk [30] did not support satisfactory mechanisms for scheduling and interleaving concurrent threads. As a result, the implementation given of a bounded buffer has a more complex interface than the usual put and get operations: the producer and consumer are required to check the return value of put and get in order to find out whether the buffer is empty or full, and suspend their own activity, if necessary. A wake-up method to be invoked asynchronously by the buffer must also be supplied. This problem was corrected in a later version of ConcurrentSmalltalk [31] by providing a monitor-like synchronization mechanism which enabled the buffer itself to suspend client threads when the buffer is empty or full. In this way the integrity of the buffer does not depend on whether its clients are well-behaved.

4. Software Composition with Reusable Patterns

The three mechanisms most notably responsible for the reusability of object-oriented software are object classes, class inheritance and genericity:

- ***Classes:*** all objects are instances of an object class, a template for objects that share the same internal structure and the same methods for responding to clients' requests.
- ***Class inheritance:*** *subclasses* can be defined as incremental modifications [29] of superclasses, with which they share some structure and some methods.
- ***Genericity:*** a generic class is a template for an object class, parameterized by the names of other object classes used within its specification.

Each of these mechanisms can be viewed as a means to reusing a behavioural pattern encapsulated as a parameterized syntactic pattern. Object classes are templates for objects parameterized by initialization values: an object instance is generated by calling a *constructor* for the class, optionally supplying values used to initialize the new object. Generic classes are a straightforward extension of this idea, with the main difference being that the parameters may be object classes rather than just values. A generic "container" class, such as a list, can be used to generate, for example, a list of integers as well as a list of windows.

We can view inheritance in the same way, by distinguishing between the two different ways in which a class may be used, namely to generate objects or to generate subclasses. Let us consider, for example, an object model in which each object consists of a control agent implementing the methods, and a hidden set of concurrent agents implementing the instance variables. A class might be defined by a pattern, as follows:

```
classA(Id) := aMethods(Id) & aVars .
```

We shall ignore, for the sake of brevity, initialization of variables, and how the object can protect its instance variables from being accessed by other objects. (This can be done either by using the *restriction* operator mentioned earlier to hide communication offers involving instance variables [20], or one may use *filtering* to hide all but selected offers to communicate with the outside world [24].)

The pattern classA(Id) can only be used to generate objects. In order to define a subclass, we need the concept of a *generator*, which is a template for a class, parameterized by additional behaviour (i.e., methods and instance variables):

```
genA(Id,MRest,VRest) := aMethods(Id) + MRest & aVars & VRest .
```

A class is instantiated from its generator by binding the "additional" behaviour to nil:

```
classA(Id) := genA(Id,nil,nil) .
```

A subclass, on the other hand, could be created by supplying as parameters the behaviour of the additional instance variables and methods. To permit further subclassing, however, we should first

create a subclass generator, adding the new behaviour and possibly introducing new parameters:

```
genB(Id,MRest,VRest) := genA(Id, bMethods(Id)+MRest,
                                 bVars & VRest) .
classB(Id) := genB(Id,nil,nil) .
```

The subclass pattern classB(Id) then results as if it had been directly defined by:

```
classB(Id) := aMethods(Id) + bMethods(Id) & aVars & bVars .
```

Overriding of inherited methods and instance variables could be handled in the same way that constructors permit default initialization of instance variables to be overridden: if the default behaviour of a method or instance variable is not what is desired, it can be simply re-assigned. (At present we support no means to do this in Abacus; "overriding" and name-conflict resolution for multiple inheritance is only possible by explicitly stating what to inherit.)

Note that our approach resembles somewhat that of Cook [9] who uses both "generators" and "wrappers" to develop a denotational semantics for functional objects. Functional objects, however, are pure values, and thus cannot be used to directly model side effects or reactive behaviour, in contrast to the case where communicating agents are used as a semantic target.

Although we do not argue that inheritance in object-oriented languages should be subsumed by parameterized software templates, we feel that modeling inheritance in terms of incremental modifications in the behaviour of communicating agents helps to expose semantic confusion in the design of an inheritance mechanism for a language, and thus leads to more robust language design.

5. Viewing Types as Partial Specifications of Behaviour

The key problem in designing a concurrency model consistent with the principles of object-oriented programming is how to package concurrent objects so that instantiation and inheritance can be safely applied without violating encapsulation [6], [14], [26], [28]. Signatures provide an abstract view of objects hiding implementation details, and thus furnish useful notions of substitutability, subtyping and type-checking. Unfortunately signatures fail to provide enough information about the externally visible behaviour of objects to guarantee valid use. We propose a new notion of types as partial specifications of external behaviour that extends substitutability and subtyping to concurrent objects.

A signature is a list of the operations (messages) understood by the object, together with their argument and return types; a subtype may add operations, permit existing operations to accept a wider range of argument types, or restrict the range of values returned by operations [8]. Signatures are inadequate to describe the possible interactions between concurrent objects and their clients, primarily because they do not take into account variations in behaviour over time:

- ***Mutability:*** an operation that allows one to set the state of an object, taking as an argument the "value" to be set, cannot be included in a subtype signature if the subtype refines the value space, since the more specific arguments required can put a client in error (see the discussion on "aging functions" in [10]).
- ***Changing roles:*** an object that presently conforms to a type specification may no longer conform in the future: teenager can therefore not be viewed as a subtype of person, even though the former may be signature compatible to the latter.
- ***Scheduling:*** concurrent objects exhibit non-uniform service availability as they attempt to schedule requests. Although a bounded buffer may support a read operation, there is no guarantee the request will return if no matching write has been issued.
- ***Interleaving:*** signatures in no way capture the interactions between an object and multiple concurrent clients.

From the client's point of view, a type should specify just enough information about an object to express the valid patterns of communication. We can interpret this in our context as meaning that (1) neither the client nor the server sends any inappropriate messages (i.e., safety), and (2) requests will be serviced (i.e., liveness).

It is convenient to think of types and subtypes in terms of partial specifications of behaviour and substitutability. In this view, to say that object x is of type t is the same as saying that t(x) is true. Furthermore, if s is also a type, and we know that all objects that satisfy s also satisfy t, then we say that s is a *subtype* of t. In effect, a type describes a "software contract" [19] between an object and its clients: a subtype is simply a stronger contract. For a client that expects an object of type t, we may substitute any object of subtype s.

In the domain of communicating agents, a concurrent type partially specifies the possible interactions between an agent and its peers. Consider, for example, the following partial specification of single-slot buffers:

```
buf0(Id) := [Id,(put,X),Prod] ? [Prod,ok] ! buf1(Id) .
buf1(Id) := [Id,get,Cons] ? [Cons,Y] ! buf0(Id) .
```

This says that buf0 alternately accepts requests from producers and consumers, but says nothing about the values that will be returned to the consumer (since Y is unbound). The implementation of the single-slot buffer buf given earlier satisfies this specification in the sense that producers and consumers that expect an object of type buf0 will be satisfied with buf. Note that buf0 can equivalently be viewed as a specification of a non-deterministic agent that provides random values to consumers. In this sense, buf is simply more deterministic than buf0.

Interestingly, signatures are completely subsumed by this view of types:

```
bufsig(Id) := [Id,(put,X),Prod] ? [Prod,ok] ! bufsig(Id)
            + [Id,get,Cons] ? [Cons,Y] ! bufsig(Id) .
```

bufsig is a partial specification of an unbounded buffer, since there is no limit to the number of put requests that can be made. It also permits the buffer to return arbitrary values, even when the buffer is empty. Note that bufsig can be viewed as a subtype of buf0, since any object that satisfies bufsig can be safely used where an object of type buf0 is expected.

We would like to define a subtype relation, s:<t, where s is a *subtype* of t, with the following properties:

1. s accepts *at least* the input offers of t
2. s presents *at most* the output offers of t
3. if s makes some input or output offer e with replacement sr, where e is an offer also made by t (thus expected by clients), then there is a replacement tr of t upon event e such that sr:<tr
4. if t makes some offer, then s must make at least one offer that t does

These criteria apply only to computation paths reachable by communications with the client. For example, the second condition doesn't apply to responses that s would make to a request that the client will not make (i.e., because t does not permit it).

According to these criteria, we can conclude that bufsig(Id):<-buf0(Id). Note that we treat input and output offers asymmetrically, in contrast to e.g., observation equivalence [20]. We justify this view by noting that input offers correspond to safety conditions (i.e., what messages the client can safely send), whereas output offers correspond to liveness conditions (i.e., the range of possible values the client can expect as a reply).

Unfortunately, our conditions appear to be necessary, but not sufficient. For example, consider an agent funnybuf that behaves just like buf, except that it blocks if a consumer tries to get a value before the producer puts anything, i.e.,

funnybuf := buf + [b,get,Cons]?nil .

This agent would conform to buf0 according to our requirements, but would non-deterministically deadlock in the presence of concurrent producers and consumers. Further constraints on the "services" specified by types and subtypes appear to be necessary to resolve this problem.

Within this framework for understanding concurrent types, we plan to investigate precisely which kinds of specifications will be useful for characterizing reuse criteria, and under which the circumstances type-checking will be feasible and practical. (If types are allowed to specify too much, "type-checking" becomes equivalent to program verification!) We have not attempted to unify object types and message types, since objects are agents, but messages (events) are pure values. In particular, an object cannot be sent as part of the contents of a message, since objects are not values: one may send an object id, or a value representing the state of the object, or even a value representing the behaviour of the object, but not the object itself. Since message contents are values, type-checking of communications can be handled in a more traditional way [8].

Two promising directions for further work are (1) to reconsider path expressions [7] as a means to describe abstract behaviour, perhaps along the lines of Procol [5], and (2) to use a restricted form of temporal logic [16] using abstract states to express the external behaviour of an object in terms of liveness and safety conditions. We are presently investigating the properties of *interaction conformance*, which characterizes agents in terms of their possible interactions with a set of observers [25].

6. Concluding Remarks

The clean integration of concurrency features into object-oriented languages is still an open problem. We have proposed a reference model for the design of concurrent object-oriented languages based on communicating agents, and we have presented a compact executable notation which can be used as a semantic target for language specification.

Although a large variety of powerful and expressive mechanisms have been proposed and included in various languages, it has proved difficult to devise an approach that is at once sufficiently powerful to easily express solutions to standard concurrency problems, and also minimizes the difficulties of reusing concurrent objects, whether by inheritance, or by other mechanisms for software composition. We claim that the majority of these problems result not so much from a particular choice of concurrency mechanisms as from a lack of good methods for encapsulating objects and specifying reuse criteria. To rectify this situation, we propose a new notion of object type that characterizes concurrent objects in terms of their externally visible behaviour.

We are working towards the design of a new generation of concurrent object-oriented language by:

- Identifying and attempting to resolve the key conflicts between concurrency and object-oriented software composition [26], [27].
- Continuing to use Abacus as a platform for exploring various models of concurrent objects [24].
- Developing a *pattern language* that will permit syntactic patterns to be bound to behavioural patterns in Abacus.
- Developing a polymorphic type model for concurrent objects that partially specifies the behaviour of objects in terms of safety and liveness conditions over interactions with clients [25].

References

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

[2] 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.

[3] A. Björnerstedt and S. Britts, "AVANCE: An Object Management System," ACM SIGPLAN Notices, Proceedings OOPSLA '88, vol. 23, no. 11, pp. 206-221, Nov 1988.

[4] A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, "Distribution and Abstract Data Types in Emerald," IEEE Transactions on Software Engineering, vol. SE-13, no. 1, pp. 65-76, Jan 1987.

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

[6] J-P. Briot and A. Yonezawa, "Inheritance and Synchronization in Concurrent OOP," Proceedings of the European Conference on Object-oriented Programming, pp. 35-43, Paris, France, June 15-17, 1987.

[7] R.H. Campbell and A.N. Habermann, "The Specification of Process Synchronization by Path Expressions," in *Operating Systems, International Symposium*, ed. E. Gelenbe, C. Kaiser, Lecture Notes in Computer Science 16, pp. 89-102, Springer-Verlag, 1974.

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

[9] Wm. Cook, "A Denotational Semantics of Inheritance," ACM SIGPLAN Notices, Proceedings OOPSLA '89, vol. 24, no. 10, pp. 433-443, Oct 1989.

[10] S. Danforth and C. Tomlinson, "Type Theories and Object-Oriented Programming," ACM Computing Surveys, vol. 20, no. 1, pp. 29-72, March 1988.

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

[12] M.J.C. Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

[13] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

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

[15] 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.

[16] L. Lamport, "Specifying Concurrent Program Modules," ACM TOPLAS, vol. 5, no. 2, pp. 190-222, April 1983.

[17] H. Lieberman, "Concurrent Object-Oriented Programming in Act 1," in *Object-Oriented Concurrent Programming*, ed. A. Yonezawa, M. Tokoro, pp. 9-36, The MIT Press, Cambridge, Massachusetts, 1987.

[18] B. Liskov, M. Herlihy and L. Gilbert, "Limitations of Synchronous Communication with Static process Structure in Languages for Distributed Computing," 13th Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, Jan 13-15, 1986.

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

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

[21] N.H. Minsky and D. Rozenshtein, "A Law-Based Approach to Object-Oriented Programming," ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 482-493, Dec 1987.

[22] J.E.B. Moss and W.H. Kohler, "Concurrency Features for the Trellis/Owl Language," Proceedings of the European Conference on Object-oriented Programming, pp. 223-232, Paris, France, June 15-17, 1987.

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

[24] O.M. Nierstrasz, "A Guide to Specifying Concurrent Behaviour with Abacus," in *Object Management*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, July 1990, (To be submitted for publication).

[25] O.M. Nierstrasz and M. Papathomas, "Towards a Type Theory for Active Objects," in *Object Management*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, July 1990, (Working Paper).

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

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

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

[29] P. Wegner and S. B. Zdonik, "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like," in *Proceedings of the European Conference on Object-oriented Programming*, ed. S. Gjessing and K. Nygaard, Lecture Notes in Computer Science 322, pp. 55-77, Springer Verlag, Oslo, August 15-17, 1988.

[30] 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.

[31] Y. Yokote and M. Tokoro, "Experience and Evolution of ConcurrentSmalltalk," ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 406-415, Dec 1987.

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