Using Metaobjects to Model Concurrent Objects with PICT[1]

Markus Lumpe, Jean-Guy Schneider, Oscar Nierstrasz
Software Composition Group, University of Berne[2]

Abstract. We seek to support the development of open, distributed applications from plug-compatible software abstractions. In order to rigorously specify these abstractions, we are elaborating a formal object model for software composition in which objects and related software abstractions are viewed as patterns of communicating processes. The semantic foundation is Milner's π calculus, and the starting point for our object model is Pierce and Turner's encoding of objects as processes in the experimental PICT programming language. Our experience shows that common object-oriented programming abstractions such as dynamic binding, inheritance, genericity, and class variables are most easily modelled when *metaobjects* are explicitly reified as first class entities (i.e., processes). Furthermore, various roles that are typically merged (or confused) in object-oriented languages such as classes, implementations, and metaobjects, each show up as strongly-typed, first class processes.

Keywords. Components, Software Composition, Object modeling, π calculus, PICT.

1 Introduction

One of the key challenges for programming language designers today is to provide the tools that will allow software engineers to develop robust, flexible, distributed applications from plug-compatible software components [19]. Current object-oriented programming languages typically provide an ad hoc collection of mechanisms for constructing and composing objects, and they are based on ad hoc semantic foundations (if any at all) [18]. A language for composing open systems, we argue, should be based on a rigorous semantic foundation in which concurrency, communication, abstraction, and composition are primitives.

The ad hoc nature of object-oriented languages can be manifested in three ways:

1. The granularity and nature of software abstractions may be restricted: the designer of a software component may be forced (unnaturally) to define it as an object. Useful abstractions may be finer (e.g., mixins) or coarser (e.g., modules) or even higher-order (e.g., a synchronization policy).

1. In *Proceedings of Languages et Modèles à Objets,* Leysin, Switzerland, October 1996, pp. 1-12.

2. *Authors' address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.4618. *Fax:* +41 (31) 631.3965.
E-mail: {lumpe, schneidr, oscar}@iam.unibe.ch. *WWW:* http://iamwww.unibe.ch/~scg.

2. The abstraction mechanisms themselves may be ad hoc and inflexible: programmers typically have only limited facilities for defining which features are visible to which clients, how binding of features (static or dynamic) should be resolved, or what kinds of entities may be composed.
3. Language features are informally specified or even implementation dependent. Combinations of features may exhibit unpredictable behaviour in different implementations.

Given the ad hoc way in which software composition is supported in existing languages, we identify the need for a rigorous semantic foundation for modelling the composition of concurrent object systems from software components. We also seek simplicity and unification of concepts: if we can understand all aspects of our object model in terms of a small set of primitives, then we have a better hope of being able to cleanly integrate these features and avoid semantic interference [18].

As a first step towards the definition of a compositional object model, we have used PICT [25], an experimental programming language based on the π calculus, as an executable specification language for modelling abstractions common to many object-oriented programming languages. Our experience shows that *metaobjects* — objects responsible for the managing the creation, initialization and behaviour of instances of a class — arise naturally when modelling advanced features in terms of more primitive mechanisms. Metaobjects provide a general way to model various aspects of object creation and composition, in contrast to ad hoc solutions that result in new language features for each new aspect.

In section 2 we motivate our choice of PICT as a modelling tool, and we present examples how object-oriented features can be modelled in this framework. In section 3 we summarize our experience using metaobjects to model various object-oriented features. We conclude with some remarks concerning future work and directions.

2 Objects as Processes

There are several plausible candidates as computational models for objects. The λ calculus has the advantage of having a well-developed theoretical foundation and being well-suited for modelling encapsulation, composition and type issues [6], but has the disadvantage of saying nothing about concurrency or communication. Process calculi such as CCS [15] have been developed to address just these shortcomings. Early work in modelling concurrent objects [22][23] has proven CCS to be an expressive modelling tool, except that dynamic creation and communication of new communication channels cannot be directly expressed and that abstractions over the process space cannot be expressed within CCS itself, but only at a higher level.

The π calculus [17] addresses these shortcomings by allowing new names to be introduced and communicated much in the same way that the λ calculus introduces new bound names. This is needed for modelling creation of new objects with their own unique object identifiers. The basic (monadic) calculus allows only communication of channel names. The polyadic π calculus [16] supports communication of tuples, needed to model passing of complex messages. The higher-order π calculus [27] supports the communication of process abstractions, which is needed for modelling software composition within the calculus itself. Interestingly, the polyadic and higher-order variants of the π calculus can be faithfully translated (or “compiled”) down to the basic calculus, so one may confidently use the features of richer variants of the calculus knowing that their meaning can always be understood in terms of the core calculus. The π calculus has previously been used by Walker [34], Jones [11] and Barrio [3] to model various aspects of object-oriented languages .

A further simplification has been studied by Honda [10], who proposed that asynchronous communication provides a better foundation for distributed systems, without any loss of expressive power. This variant (also known as the “mini π calculus”) essentially forms the core language for PICT and the basis for our study.

PICT is an experimental programming language [25] whose language features are all defined by syntactic transformation to a core language that implements the mini π calculus. PICT is as much an attempt to turn the π calculus into a full-blown programming language as it is a platform for experimenting with modelling of language features [26] and a platform for experimenting with type disciplines and type inference schemes for the π calculus [24]. We have been using PICT for modelling traditional object-oriented features, such as inheritance and dynamic binding [28] as well as more esoteric abstractions needed for composing concurrent systems, such as generic synchronization policies [13][31].

2.1 The Pierce/Turner Basic Object Model

Pierce and Turner [26] have outlined a basic model for objects in PICT, in which objects are modelled as a set of persistent processes representing instance variables and methods. The interface of an object is a record[1] containing the channels of all exported features. In figure 1 we show the specification of a concurrent queue conforming to this model.

The concurrent queue consists of (i) two exported request channels (`put` to add a new item to the queue and `get` to get a stored item) and (ii) a set of internal channels and processes representing the state of a queue object. Since communication is asynchronous, writing a value to a channel (e.g., `head!init`) is non-blocking, whereas reading a value from a channel (e.g., `tail?last > ...`) blocks the reader. A value associated to a channel (such as `head!init`) can be viewed as a message, which is consumed when it is read.

Each exported request channel is bound to a process abstraction. Processes defined with `abs` are “anonymous processes” analogous to lambda abstractions. So the `put` channel is bound to a process abstraction that reads a tuple `[value,r]` and then performs some actions. These exported abstractions are the only processes able to query and manipulate the state of an object (since the names of the channels used to realize the state are never exported). In order to simplify their use, the request channels are packaged together as a record.

The behaviour of a queue is correct in presence of concurrent clients: both methods obtain and release the necessary local resources in a manner that avoids both interference and deadlock. If a `get` request blocks because the queue is empty, a `put` request will nevertheless be possible. Interleaving `put` and `get` requests cannot interfere or result in deadlock.

1. Records, like tuples, can be easily encoded as processes in the π calculus, but are provided as primitives in PICT. It is not possible in the space available to give a complete introduction to PICT. For details, please consult the PICT tutorial [25].

```
def queue [:T:][] =             {- generic type parameter T -}
  let
    new head, tail, init        {- new, private channels -}
    run head!init               {- store name of head cell -}
    run tail!init               {- next available tail -}
  in
    record
      put = abs [value, r] >    {- put new value at tail of queue -}
        let
          new link              {- make a new tail channel -}
        in
          tail?last >           {- retrieve last available tail -}
          ( tail!link           {- store new link and value -}
          | last![value, (fold (Cell T) link)]
          | r![] )              {- and reply to client -}
        end
      end,
      get = abs [r] >           {- get value from head of queue -}
            head?item >         {- retrieve name of head cell -}
            item?[value, link] >{- retrieve value & next link -}
          ( head!(unfold link)  {- remember the new head -}
          | r!value )           {- return value to client -}
      end
    end
  end
```

Figure 1 A Concurrent Queue in PICT

The reader may have noticed (i) the generic type parameter `T` (one of the major advantages of the PICT type system is that it is quite easy to define processes with generic type parameters; the concrete type of an instantiated generic process will be inferred by the type system), and (ii) the explicitly *folding* and *unfolding* of recursive types (the type inference algorithm used by the current PICT implementation does not support recursive type resolution). The datatype `Cell`, which is used to fold a link channel, is a type alias for a generic and recursive tuple channel type: `Cell T = Rec(C) ^[T,C]`.

The essentials of concurrent objects are captured by this basic object model: encapsulation, identity, persistence, instantiation, and synchronization. It is less clear whether the model can be extended to capture other common features of object-oriented programming languages. Basic features found in most of the better known languages include self-references of objects, dynamic binding, inheritance, overriding, genericity, and class variables.

2.2 Extensions to the Basic Object Model

In this section we outline some extensions to the basic object model resulting from our experiences modelling object-oriented abstractions in PICT. For details, please refer to the corresponding technical reports [28][31].

The basic model does not encapsulate traditional class features like class variables and self-references of objects, which are needed to support dynamic binding in local method calls, and has no notion of inheritance.

```
val QueueClass =                        {- global metaobject channel -}
  let
    new total
    run total!0                         {- private class variable -}
  in
    record
      gettotal = abs [r] >              {- public class method -}
                   total?value >
                     (total!value | r!value)
                 end,
      create = abs [:T:][r] >           {- creation interface -}
                 r!(let
                       new head, tail, init
                       run head!init
                       run tail!init
                     in
                       record
                         put = abs [value, r] >
                           let
                             new link
                           in
                             tail?last > total?queued >
                               ( tail!link
                               | last![value,(fold (Cell T) link)]
                               | total!(queued+1)
                               | r![] )
                           end
                         end,
                         get = abs [r] >
                           head?item > item?[value, link] > total?queued >
                             ( head!(unfold link)
                             | total!(queued-1)
                             | r!value )
                         end
                       end
                     end)
               end
    end
  end
```

Figure 2 A Metaobject for a Concurrent Queue in PICT

Modelling Class Variables

As a first extension we add class variables and class methods to the basic model. In order to illustrate this we add a counter to the queue which counts all currently queued items in all active queues and a class method which gives us access to the value of the counter.

A straightforward mapping of these features is to define them in global scope as two processes, but this violates data encapsulation and allows every client to access these features. We have found that the most natural solution is to introduce explicit *metaobjects* to encapsulate the logic for creating and initializing instances of a class. Metaobjects [12] are a commonly used mechanism in various object-oriented programming languages to encapsulate the interpret of lan-

```
create = abs [:T:][r] >
    r!( let
        val Self = emptyRef[:AObjectType:][]
        new temporary               {- make a new channel for Self -}
        run temporary!( ...
                object creation
                ... )
      in
        Self.set[temporary];        {- bind Self -}
        Self.get[]                  {- return current value of Self -}
      end)
```

Figure 3 Initialization of Self

guage features behind the interface of an object. In this case we use metaobjects to encapsulate and restrict access to class variables and methods. Class variables and class methods are modelled as instance variables and exported methods of the metaobject, respectively.

We use metaobjects not only to model shared class features, but more generally to create, initialize, and control the behaviour of objects. In the basic model of Pierce and Turner, object creation is modelled by a generator process in global scope. Moving this generator process inside the metaobject is a first step towards modelling inheritance and self-references. A metaobject for the concurrent queue class is shown in figure 2.

`QueueClass` is declared as a unique global channel representing the metaobject for a generic concurrent queue class. The methods `put` and `get` have been extended in order to count the total number of currently queued items. The class method `gettotal` returns the total count and the class method `create` returns a new object of the queue class. Note that the generic type parameter `T` appears only in the `create` methods. The metaobject itself does not have to be generic.

Modelling Inheritance by Dynamic Binding of Self

The pseudo-variable *Self* is needed to model dynamic binding. To model this feature, we make use of a PICT library process that implements so-called *reference cells*. A reference cell is an object that provides `set` and `get` methods to set and retrieve stored values, respectively. Self is modelled as a reference cell that is set just once in the `create` method of the metaobject. In order to initialize *Self*, we first have to assign the new fresh object to a temporary channel (this is the generator process), and second to define a *fixed point operator* which delivers the minimal fixed point — *Self* (figure 3).

In a first approach we model inheritance by delegation (as in Self [30] and Sina [1]): each object owns an instance of its direct superclass. This means that only the exported methods of the superclass can be accessed by the subclass instance. Modelling dynamic binding requires special care. In the absence of dynamic binding of *Self,* if an inherited ancestor method calls a method redefined by the subclass, the original and not the redefined method will be called since *Self* within the superclass instance refers to the superclass object, but not to the subclass object. To achieve dynamic binding, we need a superclass instance in which *Self* refers to the subclass instance [7] (figure 4).

We now introduce “intermediate objects” in which all methods and instance variables of a class are defined, but *Self* is unbound: all methods have an additional first parameter `Self`. The

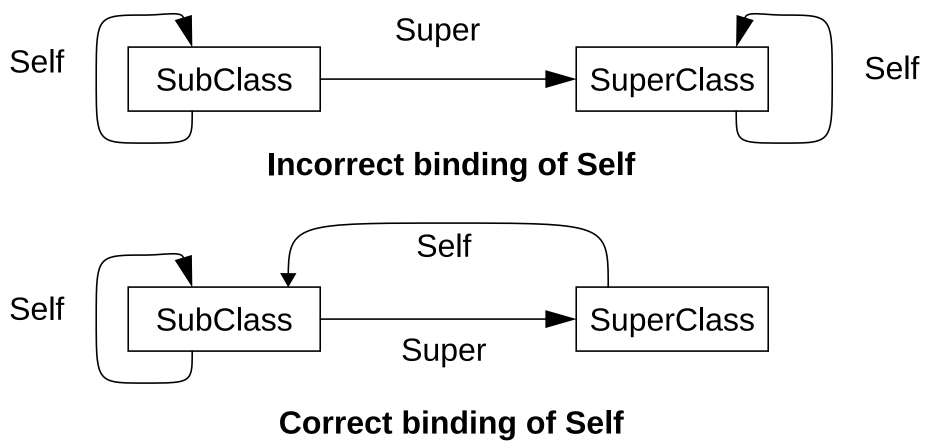

Figure 4 Binding of Self with inheritance

```
def CreateIntermediate [] =
  let
    new head, tail, init
    run (head!init | tail!init)
  in
    record
      put = abs [Self, v, r] > ...  {- Self is an explicit parameter -}
      get = abs [Self, r] > ...
    end
  end
```

Figure 5 Intermediate objects delay binding of Self

```
def Create [] =
  let
    val QueueIntermediate = CreateIntermediate []
    val Self = emptyRef []
    val NewInstance =
      record
        put = abs [v,r] = QueueIntermediate.put [Self.get[],v,r] end,
        get = abs [r] = QueueIntermediate.get [Self.get[],r] end
      end
  in
    Self.set[NewInstance];
    Self.get[]
  end
```

Figure 6 Binding Self in the metaobject

metaobject of each class defines a process `CreateIntermediate` (comparable with a generator in [7]) where the intermediate object of the class is defined (figure 5).

In the `Create` method of the metaobject, an intermediate object is created, each exported method is bound to a method defined in the intermediate object, and the correct binding of `Self` is established. As in the previous model, an empty reference cell is used to model self-reference (figure 6).

Now, in addition to exporting the `Create` method and all other public class methods, the metaobject exports the method `CreateIntermediate`, which returns a fresh copy of an intermediate object of the class.

Inheritance is now straightforward to model. In order to reuse the methods defined in an ancestor class, the metaobject of a class gets a fresh copy of the intermediate object of its direct superclass. This intermediate object is then used to define the intermediate object of the class itself. It is possible to (i) override methods, (ii) define new methods, and (iii) call inherited methods.

3 Observations

The basic object model of Pierce and Turner is a robust basis for modelling many aspects of objects. We have been able to adapt this model quite easily to support all the object-oriented features which we set as a challenge. While we added many features to objects and modified their internal representation and implementation, the interface of objects did not change.

An object is a server process containing a set of local processes and channels representing methods and instance variables. The interface to an object is a record containing the channels of all exported features. By modifying the interface record, the visibility of features can be selectively controlled.

Two mechanisms are used to control feature visibility: scope rules and type system. When finer grained control over a feature is needed, it is moved to an inner scope; for coarse-grained control, it is moved to an outer scope. The type system offers a more sophisticated way for controlling visibility: type restriction can be used to hide features whereas type extension allows features to be added or redefined. The use of type restriction may cause problems when type-safe downcasting is possible, because downcasting might be used to obtain uncontrolled access to private features.

To model class variables, class methods, and self-references, we have introduced *metaobjects* to represent classes as run-time entities. The need to use metaobjects arises naturally when we want to model correct initialization and controlled access to these features. Class variables and methods are modelled as features of the metaobject, whereas self-references are achieved by a combination of a generator and a fixed point process in the metaobject (i.e., mimicking the way self-reference can be modelled using functions and records [7]).

Although metaobjects are usually associated with Metaobject Protocols [12] (MOPs), we did not find a need to introduce a full MOP for the purpose of modelling objects in PICT. Metaobjects were useful even without any application of runtime reflection. Metaobjects provide a general way to model object creation and composition, in contrast to ad hoc solutions that introduce new language features — for example, to model the `super` feature of Smalltalk, we do not need to change the language, but simply alter the metaobject.

We also found that modelling objects and classes as processes clarifies the separate roles of mechanisms that are merged or confused in most object-oriented programming languages. For example, object-oriented languages overload classes to represent four or even five distinct notions: (i) classes as "cookie-cutters" (i.e., intensions) for objects, (ii) classes as extensible (i.e., inheritable) software components, (iii) classes as types, (iv) classes as metaobjects, and sometimes even (v) classes as sets of instances (i.e., extensions). The PICT object model clearly separates these distinct roles.

Since PICT is statically typed, every abstraction or process is statically typed. Therefore, unlike those of CLOS [12] or Smalltalk [8], our metaobjects are also statically typed. Typed metaobjects have several advantages: (i) metaobjects are typed first class objects representing plain classes, (ii) no runtime method lookup is needed, (iii) visibility of features of metaobjects can be controlled by the type system, and (iv) genericity is well-typed; it is just a parameterization of metaobject features.

Modelling inheritance and dynamic binding requires a more sophisticated solution. We found that we needed to define so-called *intermediate objects* that define all the methods and instance variables of a class, while leaving self-reference unbound. Binding of self-reference is established by the metaobject when an object is actually created. Inheritance can be modelled by copying and modifying intermediate objects of superclasses. This approach follows closely that used by Cook and Palsberg to propagate self-reference to a modified client [7].

4 Conclusions and Future Work

Our experiences show that the π calculus is expressive enough for modelling standard object-oriented programming language features in a convenient way. Walker [34] has shown that POOL [2] can be modelled in the π calculus, but in his approach, no subtyping or inheritance is supported. Subtyping and a notion of *Self* can be modelled with the “Calculus of Objects” of Vasconcelos [32]. Barrio [3] has given a nearly complete representation of active objects in the π calculus, but dynamic binding and a notion of *Self* are still missing. With this work we have shown that inheritance, dynamic binding, and self-reference can also be conveniently modelled with the π calculus with the aid of processes representing metaobjects.

Modelling object-oriented features in the π calculus is tedious work, akin to programming in a “concurrent assembler.” PICT simplifies this work somewhat by providing syntax for a large number of common, basic programming abstractions, like Booleans and integers, control structures, functions, expressions, and statements. Still, to model objects as processes, one is often obliged to forsake natural abstractions and explicitly describe behavioural in low-level, operational terms. For example, to specify the concurrent queue in figure 1, we had to explicitly create and manipulate the reply channel used to deliver `put` and `get` results to clients.

It is possible to specify the concurrent queue in PICT without explicitly mentioning reply channels, but the abstractions needed to do so are not immediately obvious [28]. Therefore we are looking for a less primitive, intermediate calculus that is more convenient for modelling concurrent objects. We are beginning to explore a so-called “guarded object calculus” (GOC) [20] in which an objects is modelled as a set of functions that read and write a local tuple space of messages representing the object’s state. Whenever an operation is called on an object, an *input guard* grabs the needed resources from the tuple space. After the calculation, an *output trigger* restores resources.

The use of guards and triggers for modelling objects has the advantage that (i) it is possible to specify any kind of operation in GOC style and (ii) objects behave correctly in the presence of multiple clients. On the other hand it is still an open question how to model other abstractions, such as local method calls, self-references and inheritance.

As an extension to our object model, we have modelled McHale's "generic synchronization policies" (GSP) [13] as composable concurrent abstraction in PICT. GSPs are reusable specifications of synchronisation policies, such as "mutual exclusion", "readers/writers" and so on, that may be bound to the implementation of different object classes. In our first approach, we used a preprocessor to translate GSP abstractions into PICT code. After a few iterations, we found we were able to omit the preprocessing phase and implement GSPs directly in PICT. There are numerous other interesting approaches in modelling concurrent objects worth investigating, such as the "composition filters" approach of Sina [4], the state variable unification approach to synchronization of Oz [29], or the *separate* extension to Eiffel [14].

Although it is our long-term goal to define an object model suitable for specifying the composition of open, concurrent systems, so far we have mainly concentrated on modelling common features of object-oriented languages that do not necessarily address concurrency. There are still a few abstractions we did not incorporate into our first object models, such as multiple inheritance, binary methods, type-safe downcasting, and constrained genericity. Modelling binary methods is a challenging task, especially in the context of subclassing and polymorphic data structures, since the definition of binary methods naturally leads to recursive type definitions. Bruce *et al.* [5] have surveyed the sources of problems with binary methods, and have presented a comparison of various solutions to these problems. We hope that an adaptation of some of these solutions to our object models will not only give us further insight into the precise requirements of a concurrent object model, but also help us to define an appropriate type system for software composition.

Although metaobjects are usually associated with MOPs, we only defined a basic MOP for our PICT object models. Two major questions arise: what kind of MOPs do we need in a composition language, and what are the consequences for the underlying type system? To our knowledge, most of the languages supporting run-time MOPs are not statically typed. It is therefore a challenging task to see what kind of MOP can be defined with the current type system of PICT, or how the type system should be extended in order to support run-time reflection using metaobjects.

Our overall goal in this work is to develop a formal model of software composition and an executable *composition language* [19] for specifying components, composition abstractions, and applications as compositions of software components. Ultimately we are targeting the development of open, hence distributed systems. A composition language for open systems should not only have its formal semantics specified in terms of communicating processes, but should really support concurrent and distributed behaviour. The run-time system of current implementation of PICT only runs on a single processor; it is not possible to specify real distribution of processes. As a first step towards real distribution, we have implemented a simple prototype for a subset of the PICT programming language supporting communication between distributed nodes [33]. What we need, however, is a distributed abstract machine as run-time system for the composition language, comparable to that used for Java [9]. A distributed abstract machine for software composition could be built on top of an existing intercomponent communication system (e.g., COM or CORBA)..

Acknowledgements

We would like to thank Benjamin Pierce for his helpful and detailed comments.

References

[1] Mehmet Aksit, “On the Design of the Object-Oriented Language Sina,” Ph.D. thesis, University of Twente, 1989.

[2] Pierre America, Jaco de Bakker, Joost N. Kok and Jan Rutten, “Operational Semantics of a Parallel Object-Oriented Language,” *Proceedings POPL ’86*, St. Petersburg Beach, Florida, Jan 13-15, 1986, pp. 194-208.

[3] Manuel Barrio Solorzano, “Estudio de Aspectos Dinamicos en Sistemas Orientados al Objeto,” PhD thesis, Universidad de Valladolid, September 1995.

[4] Lodewijk Bergmans, “Composing Concurrent Objects,” PhD thesis, University of Twente, 1994.

[5] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens and Benjamin Pierce, *On Binary Methods*, 1996, To appear in Theory and Practice of Object Systems.

[6] Luca Cardelli and Peter Wegner, “On Understanding Types, Data Abstraction, and Polymorphism,” *ACM Computing Surveys*, vol. 17, no. 4, Dec. 1985, pp. 471-522.

[7] William Cook and Jens Palsberg, “A Denotational Semantics of Inheritance and its Correctness,” *Proceedings OOPSLA ’89, ACM SIGPLAN Notices*, vol. 24, no. 10, October 1989, pp. 433-443.

[8] Adele Goldberg and David Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, Reading, Mass., May 1983.

[9] James Gosling and H. McGilton, *The Java Language Environment*, Sun Microsystems Computer Company, May 1995.

[10] Kohei Honda and Mario Tokoro, “An Object Calculus for Asynchronous Communication,” *Proceedings ECOOP ’91*, Pierre America (Ed.), LNCS 512, Springer-Verlag, Geneva, Switzerland, July 15-19, 1991, pp. 133-147.

[11] Cliff B. Jones, “A pi-calculus Semantics for an Object-Based Design Notation,” *Proceedings of CONCUR'93*, E. Best (Ed.), LNCS 715, Springer-Verlag, 1993, pp. 158-172.

[12] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[13] Ciaran McHale, “Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance,” Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, 1994.

[14] Bertrand Meyer, “Systematic Concurrent Object-Oriented Programming,” *Communications of the ACM*, vol. 36, no. 9, September 1993, pp. 56-80.

[15] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[16] Robin Milner, “The Polyadic pi Calculus: a tutorial,” ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, Oct. 1991.

[17] Robin Milner, Joachim Parrow and David Walker, “A Calculus of Mobile Processes, Part I/II,” *Information and Computation*, vol. 100, 1992, pp. 1-77.

[18] Oscar Nierstrasz, “Composing Active Objects,” *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner and A. Yonezawa (Ed.), MIT Press, 1993, pp 151-171.

[19] Oscar Nierstrasz and Theo Dirk Meijler, "Requirements for a Composition Language," *Proceedings of the ECOOP '94 Workshop on Coordination Languages*, ed. P. Ciancarini, O. Nierstrasz, A. Yonezawa, Springer-Verlag, LNCS 924, 1995, pp. 147-161.

[20] Oscar Nierstrasz, Jean-Guy Schneider and Markus Lumpe, "Formalizing Composable Software Systems — A Research Agenda," *Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems FMOODS'96*, Paris, France, March 1996, to appear.

[21] Else K. Nordhagen, "Omicron, An Object-Oriented Calculus," *Proceedings FMOODS'96*, IFIP WG 6.1 (Ed.), Paris, France, March 1996.

[22] Michael Papathomas, "Behaviour Compatibility and Specification for Active Objects," Object Frameworks, D. Tsichritzis (Ed.), Centre Universitaire d'Informatique, University of Geneva, July 1992, pp. 31-40.

[23] Michael Papathomas, "A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages," *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, M. Tokoro, O. Nierstrasz and P. Wegner (Ed.), LNCS 612, Springer-Verlag, 1992, pp. 53-79.

[24] Benjamin C. Pierce and David N. Turner, "Simple Type-Theoretic Foundations for Object-Oriented Programming," *Journal of Functional Programming*, vol. 4, no. 2, April 1994, pp. 207-247.

[25] Benjamin C. Pierce, "Programming in the Pi-Calculus: An Experiment in Concurrent Language Design," Technical Report, Computer Laboratory, University of Cambridge, UK, May 1995, Tutorial Notes for PICT Version 3.6a.

[26] Benjamin C. Pierce and David N. Turner, "Concurrent Objects in a Process Calculus," *Proceedings Theory and Practice of Parallel Programming* (TPPP 94), Takayasu Ito and Akinori Yonezawa (Ed.), Springer LNCS 907, Sendai, Japan, 1995, pp. 187-215.

[27] Davide Sangiorgi, "Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms," Ph.D. thesis, CST-99-93 (also: ECS-LFCS-93-266), Computer Science Dept., University of Edinburgh, May 1993.

[28] Jean-Guy Schneider and Markus Lumpe, "Modelling Objects in PICT," Technical Report, no. IAM-96-004, University of Bern, Institute of Computer Science and Applied Mathematics, January 1996.

[29] Gert Smolka, "A Survey of Oz," Draft, German Research Center for Artificial Intelligence (DFKI), January 24, 1995.

[30] David Ungar and Randall B. Smith, "Self: The Power of Simplicity," *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, no. 12, December 1987, pp. 227-242.

[31] Patrick Varone, "Implementation of 'Generic Synchronization Policies' in PICT," Technical Report, no. IAM-96-005, University of Bern, Institute of Computer Science and Applied Mathematics, March 1996.

[32] Vasco T. Vasconcelos, "Typed Concurrent Objects," *Proceedings ECOOP'94*, M. Tokoro, R. Pareschi (Ed.), LNCS 821, Springer-Verlag, Bologna, Italy, July 1994, pp. 100-117.

[33] Pierre Viret, "Viewing C++ Objects as Communicating Processes," Master's thesis, Laboratoire de Téléinformatique, Ecole Polytechnique Fédérale de Lausanne (EPFL), CH, March 1996.

[34] David Walker, "Objects in the π calculus," Information and Computing, vol. 116, no. 2, 1995, pp. 253-271.