Formalizing Composable Software Systems — A Research Agenda[1]

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

Abstract. Flexibility is achieved in open systems by adopting software architectures that allow software components to be easily plugged in, adapted and exchanged. But open systems are generally concurrent, distributed and heterogeneous in addition to being adaptable. Ad hoc approaches to specifying component frameworks can lead to unexpected semantic conflicts. We propose, instead, to develop a rigorous foundation for composable software systems by a series of experiments in modelling concurrent and object-based software abstractions as composable, communicating processes. Eventually we hope to identify and realize the most useful compositional idioms as a *composition language* for open systems specification.

Keywords. Components, Object-Oriented Programming, Software Composition, π calculus, PICT.

1 Introduction

Complex software systems are increasingly required to be open, flexible conglomerations of heterogeneous and distributed software components rather than monolithic heaps of code. This places a strain on old-fashioned software technology and methods that are based on the maxim:

Programs = Algorithms + Data

This equation perhaps still has some relevance for well-defined and delimited problems, but it tells us nothing about how to coordinate complex systems. We now need an equation of the form[3]:

Open Systems = Components + Coordination

But what do we mean by "components"? How should we understand "coordination"? Are components and coordination just glorified data and algorithms, or is something else going on? How can we tell? A rigorous semantic foundation for defining, studying and comparing approaches would clearly help, but where should we start, and what can we hope to achieve?

1. In *Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems FMOODS'96*, Paris, France, Chapmann and Hall, March 1996, pp. 271-282.

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:* {oscar, schneidr, lumpe}@iam.unibe.ch. *WWW:* http://www.iam.unibe.ch/~scg.

3. Or perhaps, as one wag put it: Objects = Objects + Objects

We present, in section 2, a list of requirements and a research agenda for formalizing software composition. In section 3, we outline our observations from modelling object-oriented features in PICT, an experimental programming language based on the π calculus. In section 4 we discuss various ways in which we may bootstrap from the low-level computational model of the π calculus to higher-level composition abstractions and idioms useful for developing open systems. We conclude with some remarks about future work and directions.

2 Requirements

Open systems pose an interesting mix of computational and compositional requirements.The basic requirements for open systems are that they be distributed, (and hence concurrent), heterogeneous and evolving. Clearly some form of computation is the end goal of any system, but an open system has the additional constraint that the computations it performs be flexibly composed from interchangeable components.

The *computational* viewpoint of open systems is that of collections of communicating, potentially active *objects*. The *compositional* viewpoint is that of collections of coordinated *components*. These two viewpoints are complementary rather than opposing. They are not, however, equally well-supported by current software technology and methods. In fact, the object viewpoint is often presented and interpreted as if it were a component viewpoint, with consequent disappointments and failures in software projects.

The object viewpoint is essentially computational. Objects can be seen as (either active or passive) server processes that encapsulate and manage computational resources and data. Composition is provided mainly through programming language features: dynamic binding, inheritance and genericity support, respectively, plug-compatibility, incremental modification, and parameterization. But only the last of these three allows one to compose specifications at a higher level of abstraction. In general it is not possible to specify new kinds of objects by composing library abstractions, just as it is not possible to specify systems of objects by composing library abstractions alone.

The component view is compositional. Components are *designed* to be plugged together. Components may be implemented as objects, but they need not necessarily be. The granularity of a component is typically coarser than that of objects, but may also be finer. (Mixins and synchronization policies are good candidates for fine-granularity components.) The main difference between the two viewpoints is (or should be) that the component viewpoint makes system architecture explicit [15]. The kinds of components that may exist in the system, what their interfaces are, how they can be plugged together, and how they are currently configured, must be explicitly represented if the system is to evolve in a disciplined way [25].

2.1 A Research Agenda

If we are to have any hope in developing a better software technology for open systems we must be precise about our requirements, and we must resolve the computational and compositional (or object and component) viewpoints. A kind of “lambda calculus for open systems” would help us to formally specify different notions of objects and components, compare object models, investigate the integration of computational and compositional language features like synchronization and inheritance, explore richer notions of contracts and type compatibility for concurrent systems, and reason about properties of systems built from components.

Here are some of the questions we would like to answer:

- *What is an appropriate Object Calculus?* The π calculus [20] seems to support the basic features we wish to model: concurrency, communication, abstraction, mobility, creation of processes and names. But the π calculus is very low level, and we must work hard to model objects [29][31]. Is there a more suitable formalism with equivalent expressive power but more convenient abstractions for modelling and reasoning about objects and components?
- *What is a good model of objects and components?* One appealing view is that of objects as processes and components as (higher-order) abstractions over the object space [22]. Should objects and components be unified? Do functions play a special role? Should active and passive objects be distinguished?
- *What forms of composition are fundamental?* Most forms of composition, including inheritance [5], can be reduced to more basic forms. Functional composition, communication and genericity seem to be primitive, but genericity is only an issue in typed systems, and functional composition can be effectively modelled by communication [30], so perhaps communication is the root of all forms of composition. What is a good basis for modelling other kinds of composition (such as inheritance, pipes, dataflow, triggering, etc.)?
- *Is there a uniform type system that accommodates objects and components?* Most object-oriented languages do not treat all software entities uniformly. The inheritance interface, for example, is typically not typed [14], but is indirectly described by ad hoc language constructs to control the visibility of features to clients.
- *Can dynamic aspects of contracts be expressed in the type system?* Services provided by objects and components may not be uniformly available. Correct compositions may depend on clients and servers conforming to a common protocol [23].
- *Can we reason about correctness in a compositional way?* Traditional approaches to software specification and verification require global knowledge to prove programs correct. In an open system, global knowledge is by definition not available. We would like to reason about correctness of parts of a system based on known properties of constituent components and their compositions [22].
- *How can we explicitly represent software architecture?* A "composition language" would serve the component viewpoint much as object-oriented languages serve the object viewpoint. Explicit representations of components, compositions and software architectures should facilitate the evolution of open systems [15][25].

2.2 Experiments in Formalizing Software Composition

We have been using PICT [29], an experimental programming language based on the π calculus, as an executable specification language for modelling compositional abstractions. We have used it to model both traditional object-oriented features, such as inheritance and dynamic binding [31], as well as more esoteric abstractions needed for composing concurrent systems, such as generic synchronization policies [16][34]. We expect these modelling exercises to lead us to (i) an expressive formal model of objects and components, and (ii) a formal language for specifying open systems abstractions at a higher level than the π calculus.

Our goal is to define a formal foundation that we can use to design and implement a *composition language* suitable for specifying component frameworks for open systems development [24]. Such a language would support both *Component Engineers* who need a means to specify compositional interfaces, rules and components, i.e., *component frameworks*, and *Application Developers*, who will use component frameworks to develop specific applications, i.e., *compositions*.

A *system* for software composition should also support the integration of components written in other systems and languages and it should offer component engineers and application developers an interactive environment that supports design, composition and re-engineering. At present, we are (i) identifying and developing software abstractions for open, distributed applications, and (ii) developing experimental tools to support visual composition of graphically presented software components. These additional efforts provide us with concrete application requirements for the design of the composition language.

3 Modelling Objects as Processes

PICT is an experimental programming language [28] whose features are defined by syntactic transformation to a core language that implements the mini π calculus (a reduction of the π calculus [20] originally proposed by Honda and Tokoro [12]). 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 [29] and a platform for experimenting with type disciplines and type inference schemes for the π calculus [27]. As such, it appears to be an ideal tool for modelling objects and more advanced object features.

A detailed description of PICT is beyond the scope of this paper; for further information about its usage, its implementation, and its type system refer either to the PICT tutorial [28], or to Turner's thesis on the implementation of PICT [33].

3.1 The Pierce/Turner Basic Object Model

Pierce and Turner [29] have outlined a basic model for object as processes in PICT, in which an object is 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. A concurrent queue could thus be modelled as shown in figure 1.

A concurrent queue consists of (1) two exported request channels (`put` to add a new item to the queue and `get` to get a stored item) and (2) a set of internal channels and processes representing the state of a queue object. Each request channel is the interface to a process abstraction. These are defined using the keyword `abs` and 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

1. Records, like tuples, can be encoded as processes in the π calculus, but are provided as primitives in PICT.

```
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 > item?[value, link] >
                ( head!(unfold link){- remember the new head -}
                | r!value )         {- return value to client -}
            end
        end
    end
```

Figure 1 A Concurrent Queue in PICT

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 sources in a consistent sequence, thus avoiding both interference and deadlock.

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 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.

3.2 Modelling Object-Oriented Abstractions in PICT

Let us outline some observations resulting from our experiences modelling objects in PICT. For details, please refer to the corresponding technical reports [31][34].

The basic object model of Pierce and Turner is a robust basis for modelling many aspects of objects. We have been able to extend this model to support all the basic features mentioned above. 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 protected 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 [5]).

Metaobjects and Metaobject Protocols [13] (MOPs) are a key feature of several object-oriented languages and systems, including CLOS [8], Smalltalk [10], Beta [2] and now even C++ [4]. Although metaobjects are usually associated with 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 mechanism for modelling various aspects of object creation and composition, in contrast to ad hoc solutions that result in new language features for each new aspect — for example, to model the `super` feature of Smalltalk, we do not need to introduce a new language feature, 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 or Smalltalk, 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 [5].

As an extension to our object model, we have modelled McHale's "generic synchronization policies" (GSP) [16] 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.

4 Compositional Idioms

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. This is not inherently a problem of either the π calculus or PICT, but is rather symptomatic of the fact that we have not yet been able to identify the right compositional idioms for specifying concurrent objects.

In fact, 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. It is necessary to model a range of different kinds of objects before such compositional idioms become apparent and can be factored out as useful software abstractions.

A number of questions then suggest themselves: Can we identify a less primitive, intermediate calculus that is more convenient for modelling objects and components? Can a type system be developed, perhaps based on that currently used for PICT, that is more convenient for characterizing the kinds of abstractions we need? Can we identify a set of "kernel abstractions" that simplify the task of modelling higher-level components? Can we use our modelling tools to formal characterize reusable design abstractions, i.e., design patterns? Can we adequately characterize the compositional rules of a component framework? Let us briefly consider each of these questions in turn.

Several authors have already proposed various "object calculi" for modelling object-oriented concepts [6][12][18][21][26]. So far none of these calculi provides *both* a good basis for modelling concurrent object abstractions *and* a formal foundation as mature as that of the π calculus (while acknowledging that the π calculus is still far from being well-understood!). Rather than trying to define yet another original object calculus, it would seem to be a better strategy to look for an intermediate calculus that (i) provides the "right" abstractions for modelling component frameworks, and (ii) can be easily specified by a mapping to the π calculus, just like PICT constructs are defined by a mapping to the mini π core language. The difference is that we would

```
concQueue () =                   -- concurrent queue abstraction
    let
        new head, tail, item     -- local tuple space
        head!item                -- name of head item in queue
        tail!item                -- next available tail slot
    in {                         -- record with two fields
        put val =                -- put new value at tail
            let
                tail?item        -- get next available tail slot
                new link         -- make a new slot name
                item!(val, link) -- link value into queue
                tail!link        -- remember new tail slot
            in
                ()               -- confirm completion
            end
        get () =                 -- get value from head of queue
            let
                head?item        -- get name of head item
                item?(val, link) -- retrieve value and next item link
                head!link        -- remember the next head
            in
                val              -- return the value
            end
    }
```

Figure 2 The Concurrent Queue in the GOC Idiom

like to be able to work exclusively at the higher level of the intermediate calculus, and hide the π core.

As a hypothetical example, consider the respecification of our concurrent queue in a so-called “guarded object calculus” (figure 2). With only some minor syntactic variation, we can write this specification directly in PICT. The key difference is that we restrict ourselves to the guarded object calculus (GOC) idiom, which can be summarized as “Linda meets Lambda”: terms are lambda expressions (abstractions or applications), possibly decorated with input guards or output triggers relative to a local tuple space. Instead of being able to specify arbitrary π calculus processes, as is possible in PICT, we are now forced to specify components exclusively in the GOC style. On the one hand, this liberates us from having to explicitly represent primitive notions such as reply channels, on the other hand it takes away from us the freedom to represent these notions, should we need them.

By analogy, consider the difference between programming in a pure object-oriented language like Eiffel, in which we *must* program with objects, or programming in C++, which enables, but does not enforce the object paradigm. The relative advantages and disadvantages of the two approaches are clear in both cases, and are not the subject of our debate. Instead, our question is, given that we want to enforce a *component*-oriented paradigm, what should be the core abstractions that we provide? Is the GOC idiom a good basis, or do we need to look further?

This example presents preliminary ideas using the GOC idiom. The constructs illustrated are very closely related to those currently available in PICT. But the process of developing a suitable intermediate calculus is an incremental one. We are continuing with our PICT object modelling experiments, and hope to discover useful compositional idioms in this fashion. Therefore, we hope that by evaluating our results and refining the intermediate calculus we will be able to establish the foundations for a compositional programming environment.

The current type system of PICT is very rich, but it also has some drawbacks. First, the programmer has to explicitly *fold* and *unfold* values of recursive types (`unfold` transforms a value with a recursive type into one with a non-recursive type, `fold` is used for the inverse transformation). In the queue implementation, the types of all internal channels (`head`, `tail`, `init`, `link`) are recursive. Second, in the presence of recursive types no subtype relation can be established. In the current version of PICT, two recursive types are either equal or incomparable[1], which is problematic when one wants to use polymorphic data structures. At this point it is not clear how the PICT type system can be consistently extended to support subtypes for recursive types. (Subtyping of recursive types is still an active research area.)

An important task will be to discover what kind of abstractions are necessary for software composition. Do we really need all the abstractions common to most object-oriented languages, or are there abstractions which should be avoided? Is it, on the other hand, possible (or even desirable) to incorporate abstractions from other programming paradigms (such as functional or logic programming)? One possible approach is to define a small number of well-understood and orthogonal kernel abstractions and to provide mechanisms for defining higher level abstractions in terms of the kernel abstractions. These higher level abstractions can be seen as syntactic sugar on top of the kernel abstractions and be used to define domain-specific framework abstractions. (PICT already goes a long way in this direction.)

Design patterns [9] are specifications of compositional idioms at the level of design rather than as concrete software abstractions. Design patterns provide specific *design* guidelines for building flexible object-oriented systems given certain requirements. They provide, amongst others, guidelines for setting-up and adapting class structures. So far design patterns are not available as reusable generic abstractions that are realized in software. No approaches have been presented that transform these guidelines into rules that can be enforced in software in order to make adaptation and extension of class structures easier. One may apply design patterns in the implementation of a system, and one may recognize where they have been used, but the possibility to reuse these patterns in software is rather restricted. Turning design patterns into reusable software abstractions would be one step towards explicitly representing software architecture in the implementation of open systems [15][17]. A good test of a formal object calculus is how well it can be used to express design patterns as components.

In [7], a knowledge-based parallel programming environment is presented. The environment assists an application programmer in finding an *algorithmic skeleton* well-suited for solving a particular problem, which can then be completed by the programmer. A similar approach could be used for design patterns: the composition language can be used to specify *design-pattern component templates*, which only need to be bound to application-specific classes and components.

Before we define a component model for *software*, it is natural to have a closer look how *hardware* is built. A stereo system, for example, may consist of an amplifier, CD-player, tuner,

1. According to Pierce, this is likely to change in a future version of PICT.

tape deck, and other components. Each of these stereo components is built up from smaller components (e.g., circuits), which again use even smaller components (e.g., transistors). All stereo components have a well-defined basic behaviour and support standard interfaces. Before they can be used, they have to be connected to a power supply. Although in principle each of these components would function by itself, their real value lies in the way they are designed to be plugged together.

A customer composing a stereo system is usually not interested in how the components are built, but is interested in the services they deliver (a tape deck should support a specific noise reduction system) and their composability (it should be possible to connect components from different vendors). The producers on the other hand have a different view of their hardware components: they know the internal architecture (design and implementation) of their components, and often reuse existing layouts and pieces of hardware to develop new components.

We would like to adapt the concepts and standard mechanisms of hardware composition and use them in software development. If we hope to compose software in the same way hardware is composed, a (software) *component framework* must support the specification of (i) exact behaviour of components, (ii) standard interfaces, (iii) protocols for intercomponent communication, and (iv) rules for component substitutability.

There are other properties which a component framework must guarantee. As an example, let us consider dynamically bound local method calls in object-oriented programming: a method `foo` calls another method `bar`. If the method `bar` is redefined in a subclass, its behaviour is changed. As a side effect, the behaviour of `foo` will also change, although the implementation of `foo` has not changed. We argue that any kind of implicit dependencies have to be avoided in a component framework: all allowable dependencies should be explicitly represented and documented.

5 Future Work

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.* [3] have surveyed the sources of problems with binary methods, and have presented a comparison of various solutions to these problems. We plan to adapt some of these solutions to our π calculus object models.

One of the next steps will be to model abstractions for concurrent and distributed programming and to define a concurrent object model. As mentioned above, we already have some preliminary results in modelling McHale's "generic synchronization policies." There are numerous other interesting approaches concurrent objects worth investigating, such as the "composition filters" approach of Sina [1], the state variable unification approach to synchronization of Oz [32], or the *separate* extension to Eiffel [19].

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.

A general-purpose software composition system must support the integration of components developed using other systems or languages. As a first step towards such an integration, we have implemented a simple interpreter for a subset of the PICT programming language with an additional possibility to integrate C++ objects [35]. Although this first prototype has only limited applicability, it has helped us to obtain further insight in how to define precisely the requirements for such integrations. An important step will be to study already existing standards for intercomponent communication (e.g., COM and CORBA).

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 prototype mentioned in the paragraph above is first step in this direction, since one can add components that support communication between distributed nodes. What we need, however, is a distributed abstract machine as run-time system for the composition language, comparable to that used for Java [11]. A distributed abstract machine for software composition could be built on top of an existing intercomponent communication system.

The development of an environment for software composition will be an iterative task. In each iteration step, the usability of the composition environment for real applications has to be validated by (i) programming components within the environment, (ii) integrating components written in other systems and languages, (iii) building up component libraries, and (iv) using components to build larger applications. After each step, the benefits and drawbacks have to be carefully evaluated in order to improve the environment.

References

Further references may be found at: http://iamwww.unibe.ch/~scg.

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

[2] Søren Brandt and René W. Schmidt, “The Design of a Meta-Level Architecture for the BETA Language,” *Proceedings of META ’95: Workshop on Advances in Metaobject Protocols and Reflection at ECOOP ’95*, August 1995.

[3] 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.

[4] Shigru Chiba, “A Metaobject Protocol for C++,” *Proceedings of OOPSLA ’95*, ACM SIGPLAN Notices, vol. 30, no. 10, October 1995, pp. 285—299.

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

[6] Laurent Dami, “Functions, Records and Compatibility in the Lambda N Calculus,” *Object-Oriented Software Composition,* O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 153-174.

[7] Karsten M. Decker, Jiri J. Dvorak and René M. Rehmann, “A tool environment for parallel programming — User-driven development of a novel programming environment for distributed memory parallel processor systems,” *Priority Programme Informatics Research, Information Conference Module 3 on Massively parallel systems,* November 1994, pp. 40—47.

[8] Linda G. DeMichiel and Richard P. Gabriel, “The Common Lisp Object System: An Overview,” *Proceedings ECOOP ’87,* J. Bézivin, J-M. Hullot, P. Cointe and H. Lieberman (Ed.), LNCS 276, Springer-Verlag, Paris, France, June 15-17, 1987, pp. 151-170.

[9] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns,* Addison Wesley, Reading, MA, 1995.

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

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

[12] 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.

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

[14] John Lamping, “Typing the Specialization Interface,” *Proceedings OOPSLA ’93, ACM SIGPLAN Notices,* vol. 28, no. 10, Oct. 1993, pp. 201-214.

[15] Jeff Magee, Naranker Dulay and Jeffrey Kramer, “Specifying Distributed Software Architectures,” Proceedings European Software Engineering Conference, Springer Verlag, Lecture Notes in Computer Science, 1995.

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

[17] Theo Dirk Meijler and Robert Engel, “Making Design Patterns Explicit in FACE, a Framework Adaptive Composition Environment,” draft manuscript, IAM, U. Berne, April 1996, Submitted for publication.

[18] Tom Mens, Kim Mens and Patrick Steyaert, “OPUS: a Calculus for Modelling Object-Oriented Concepts,” Technical Report, No. VUB-TINF-TR-94-04, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1994.

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

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

[21] Oscar Nierstrasz, “Towards an Object Calculus,” *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. 1-20.

[22] Oscar Nierstrasz and Laurent Dami, “Component-Oriented Software Technology,” in *Object-Oriented Software Composition,* ed. O. Nierstrasz and D. Tsichritzis, Prentice Hall, 1995, pp. 3-28.

[23] Oscar Nierstrasz, “Regular Types for Active Objects,” *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 99-121.

[24] 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.

[25] Oscar Nierstrasz and Theo Dirk Meijler, “Research Directions in Software Composition,” ACM Computing Surveys, vol. 27, no. 2, June 1995, pp. 262-264.

[26] Else K. Nordhagen, “Omicron, An Object-Oriented Calculus,” *Proceedings FMOODS’96*, IFIP WG 6.1 (Ed.), Paris, France, March 1996.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] David N. Turner, “The Polymorphic Pi-Calculus: Theory and Implementation,” Ph.D. thesis, Department of Computer Science, University of Edinburgh, UK, 1996.

[34] 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.

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