Towards an Object Calculus*

Oscar Nierstrasz
University of Geneva†

Abstract

The development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundations for defining their semantics. Furthermore, the delicate relationship between object-oriented features supporting reuse and operational features concerning interaction and state change is poorly understood in a concurrent setting. To address this problem, we propose the development of an object calculus, borrowing heavily from relevant work in the area of process calculi. To this end, we briefly review some of this work, we pose some informal requirements for an object calculus, and we present the syntax, operational semantics and use through examples of a proposed object calculus, called OC.

1 Introduction

In order for object-oriented languages to be an effective medium for implementing reusable software components for reactive applications, they must be able to cope with concurrency, distribution and persistence. Although distribution and persistence can arguable be considered as being purely run-time concerns, concurrency cannot, for it directly concerns the semantics of software composition. There have been numerous attempts in recent years to integrate concurrency features into object-oriented languages (see [33] for a survey). As a result of these experiences, a number of difficulties have become apparent:

1. Most concurrent object-oriented languages lack a well-defined semantic foundation. There is no generally accepted semantic domain or computational model for specifying such languages or for comparing their features. This naturally makes it quite difficult to reason about the abstract properties of any software component.

2. The clean integration of concurrency features with object-oriented features supporting encapsulation and reuse is difficult to achieve. In the particular case of inheritance, difficulties that arise in sequential languages due to confusion between encapsulation of instances relative to their clients and encapsulation of classes relative to subclasses are aggravated in the concurrent case [22].

3. Compositionality of concurrent objects is poorly understood. Standard notions of polymorphism do not carry over very well to the world of concurrent objects that may exhibit non-uniform service availability [29, 31].

To address these issues, we propose the development of an *object calculus* that integrates the concept of *agents* present in process calculi with that of *functions* present in λ calculi. An (active) object can then be viewed as a function (agent) with state. Mechanisms for software composition can be viewed as functional composition of agents. The semantics of concurrent object-oriented languages can then be understood within a uniform framework that addresses both computational and compositional issues.

*In *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, ed. M. Tokoro, O. Nierstrasz, P. Wegner, A. Yonezawa, LNCS 612, Springer-Verlag, 1992, pp. 1-20.

†*Mailing address:* Centre Universitaire d'Informatique, 12 Rue du Lac, CH-1207 Geneva, Switzerland. *E-mail:* oscar@cui.unige.ch

We shall first briefly review the status of current work in semantics of concurrent object-oriented languages and trends in process calculi, and summarize our requirements for an object calculus. We shall then proceed by presenting and exploring an attempt at the definition of such a calculus. We conclude with some remarks on various theoretical and practical considerations for future exploration.

2 The Search for an Object Calculus

2.1 Semantics of Concurrent Object-Oriented Languages

Until recently, much of the work on models of concurrency has proceeded independently of the development of concurrent object-oriented languages. Perhaps the earliest attempts to provide semantic foundations for these languages has been by means of the actor model [1, 12]. (Which helps to explain why many concurrent object-oriented languages either are, or claim to be, actor languages.) More recently, both operational and denotational semantics have been developed for POOL, based respectively on transition systems [3] and on complete metric spaces [4]. Rewriting logics have been used to provide an operational semantics for Maude [23] (with a corresponding denotational semantics based on category theory).

Another promising direction is to use process calculi as a semantic foundation for concurrent object-oriented languages. In this case, the approach is to view objects as "patterns" of agents that obey the higher-level protocols established by the programming language. One may view the specification of a programming language as a mapping from syntactic patterns representing language constructs to the behavioural patterns that they stand for [29, 30]. The choice of the underlying computational model is critical if the semantic mapping is to be as simple as possible. As such, one would like the primitives of the process calculus to be as natural as possible for modelling the concepts of the programming language.

Difficulties with the semantics of inheritance in object-oriented languages, and in particular, difficulties with the conflict between inheritance of code versus inheritance of specification [14], have led to interest in formal semantics for inheritance [10, 13]. Although there have been many attempts to unify inheritance of code and specification, there is some consensus that the two concepts should be kept separate [5]. On the one hand, there is some feeling that inheritance is not the right way to approach software composition as the complex mechanics of inheritance are not made explicit [16, 35, 37], and on the other hand there are some ongoing attempts to "unbundle" inheritance to make its mechanics more explicit [9, 17]. In either case, object-oriented software composition is essentially being viewed as *functional composition* of the software components that make up an object.

We feel that these trends lend weight to our conviction that an integration of functions and agents will lead to calculus suitable for modelling both the computational aspects of active objects as well as the compositional aspects of mechanisms for software reuse.

2.2 Trends in Process Calculi

During the 1980s there has been a great deal of relevant work in the development of models of concurrency based on synchronously communicating concurrent *agents* (also called "processes"). The most familiar and accessible work has been on Milner's Calculus of Communicating Systems (CCS) [24] and Hoare's Communicating Sequential Processes (CSP) [19].

The Actor model of computation [1] bears comparison to the agent-based models, but is based on asynchronous communication, and its theoretical foundations are less well-developed (there is, for example, no commonly accepted notion of actor equivalence). More recently there has been work by Honda and Tokoro on the development of a process calculus for actor-like objects [20] and, within this framework, a notion of actor equivalence that is closely related to, but distinguishable from, usual notions of process equivalence [21].

There has been renewed interest in extending process calculi to cope with the communication of

"labels" (i.e., the name, or *port* by which one may address a communication to a particular agent) and the communication of *agents* themselves. CCS did not originally permit label passing because of technical difficulties in controlling the scope of names, but this meant that only static interconnections between agents could be modelled. Label passing is important for modelling the semantics of active objects because (1) object identifiers can in general be communicated between objects, and (2) reflective capabilities, which are especially important in persistent object systems, are more easily modelled if we can manipulate and communicate behaviours (just as we can manipulate and communicate functions in the λ calculus). Since the label of an agent provides access to the agent itself, both research directions can be seen as attempts to integrate process calculi and λ calculi (i.e., where function application is analogous to communication and functions are first class communicable values).

The original work on extending CCS to accommodate label passing is by Engberg [15]. More recently, there has been the development of the π calculus [25], Thomsen's Calculus of Higher Order Communicating Systems (CHOCS) [38], Nielsen's λ calculus with first-class processes [28], Boudol's proposal for a concurrent λ calculus [8], and Berry and Boudol's Chemical Abstract Machine (CHAM) [7]. The most important contributions in these developments appear to be:

1. The notion of *migration* in the π calculus which facilitates the creation and visibility of names analogously to the substitution and conversion rules of the λ calculus [6].
2. The *structural congruence* of CHAM that simplifies the expression of the semantic reduction rules.

Although various authors have demonstrated how λ calculi can be accurately encoded or embedded in these process calculi [8, 26, 38], a single calculus that truly unifies the notions of functions and agents in a convincing way remains a topic for active research. In particular, the theoretical foundations of HOπ, a higher-order variant of the π calculus in which not only labels but also processes may be communicated, are being explored by means of translation to the (first-order) π calculus [36].

2.3 Some Requirements for an Object Calculus

There are three fundamental aspects of concurrent object-oriented languages that we would like to capture through the formalism of an object calculus:

Encapsulation:

- *Objects* are processes that encapsulate services. Each communication is typically either a *request* or a *reply*, and every communication of a request eventually results in a reply.
- Objects have an internal state which may (or may not) change. This state is accessible only indirectly through the services provided.
- Objects (usually) have a unique *identity* or name which is needed to gain access to its services.

Active objects:

- Objects are autonomous entities that have full control over which communications they will send or accept at any time.
- Objects may be internally concurrent.
- An object may simultaneously service multiple pending requests.

Composition:

- Objects may be composed of systems of more primitive objects.
- Objects may be specified as a functional composition of (higher-order) abstractions over objects, services and other object parts. In composing objects, it is possible to override "inherited" services with new ones.

If we take process calculi as a natural starting point for modelling active objects as processes, it quickly becomes clear that a unification of functions and processes is needed in order to express object composition. An *agent* can be either viewed as a process (when communicating by message passing) or as a "function with state" (when accepting input by local application). Objects can then be seen either as primitive agents or as compositions of lower-level agents.

We can now translate our requirements to specific features that would be desirable in an object calculus:

1. *Concurrency:* as in process calculi.
2. *Tuple-based communication:* to model complex messages.
3. *Local and remote communication:* unifying the functional and process paradigms.
4. *Recursion:* to express non-terminating agents and state change.
5. *Higher-order agents:* to express agent composition.
6. *Name creation:* as in the π calculus, to create unique identifiers for objects.
7. *Left-preferential choice:* to express overriding of services.

In the following we will present an attempt to define such an object calculus and explore some of its properties.

3 OC — An Object Calculus

The object calculus (OC) we present here is an evolution of Abacus [29, 30], an executable notation based on CCS and intended for specifying and prototyping object-based concurrent languages. OC bears the same relationship to Abacus that the π calculus does to CCS — we have tried to take advantage of recent developments in process calculi to simplify and generalize the notation, and we have tried to recover the higher-order expressiveness of the λ calculus. In fact, as we shall see, OC is effectively a unification of the π and λ calculi, with communication generalized to tuples. OC is an experimental calculus — by applying OC to concrete examples in object-based concurrency, we hope to arrive at a practical calculus and also to motivate strongly further theoretical explorations.

The terms of OC consist of a set of expressions representing *agents*, $\mathcal{A}$. Agents are composed of a set of *names*, $\mathcal{N}$, a set of *output patterns* (or *values*), $\mathcal{V}$, and a set of *input patterns*, $\mathcal{X}$. We let a, b, c range over $\mathcal{A}$, n, m over $\mathcal{N}$, v over $\mathcal{V}$ and x over $\mathcal{X}$.

The syntax of agents is as follows:

$$a ::= a\&a \mid n := a \mid a|a \mid x \to a \mid v \wedge a \mid a@v \mid n\backslash a \mid n \mid \mathsf{nil}$$

The operators are given from loosest to tightest binding. Concurrent composition is &, recursion is :=, (left-preferential) choice is |, abstraction (input) is $\to$, output is $\wedge$, (functional) application is @, restriction (of local names) is $\backslash$ and the inactive agent is given by nil. A name n stands for an agent only if it has been bound to an agent expression by a communication or by a recursive definition. Our syntax differs slightly from that of the π calculus, partly because OC is executable (so we prefer to use typable characters), but mainly because we adopt a tuple-based rather than a channel-based approach to

communication. As we shall see, we have chosen $\rightarrow$ to denote an input guard as it will serve for accepting both remote and local (functional) communications.

The operators &, |, $\rightarrow$, $\wedge$ and $\backslash$ associate to the right, and @ associates to the left. So $x \rightarrow y \rightarrow a$ is parsed as $x \rightarrow (y \rightarrow a)$ and $a@b@v$ as $(a@b)@c$.

Agents communicate by sending messages, which are tuples containing names, agents or other tuples:

$$v ::= n \mid a \mid (v, ..., v)$$

In general, communications may only take place when an output matches an input pattern of another agent. Input patterns are tuples of names and local variables:

$$x ::= n \mid n? \mid (x, ..., x)$$

The construct $n?$ occurring in x binds the name n locally within a in the expression $x \rightarrow a$. These locally bound names act as variables in communications, whereas the free names (i.e., not annotated by ?) serve to match input and output patterns.

Definition 1 *Matching of outputs and inputs is denoted by $\sim$, defined as follows:*

$$\begin{array}{lcll} n & \sim & n & \\ v & \sim & n? & \\ \vec{v} & \sim & \vec{x} & \Leftrightarrow \forall i, v_i \sim x_i \end{array} \quad \square$$

We will make use of a slightly special substitution function, $a\{v/x\}$ to substitute free occurrences in a of variables introduced in x by their matching values in v. Free names in x are ignored as they must match exactly the corresponding names in v, so no substitution is required. So $a\{(b, m)/(n?, m)\}$ causes (free) instances of n in a to be substituted by b. On the other hand, a substitution such as $a\{(b, m)/(n, m)\}$ is invalid, since $(b, m) \not\sim (n, m)$.

As in CHAM and the π calculus, we start by defining the structural congruence, $\equiv$:

Definition 2 *Structural congruence* for agents is the smallest congruence $\equiv$ over $\mathcal{A}$ satisfying:

1. $a\&b \equiv b\&a$, $a\&(b\&c) \equiv (a\&b)\&c$, $a\&nil \equiv a$
2. $n := a \equiv a\{(n := a)/n?\}$
3. $n\backslash a \equiv a$, $n \notin fn(a)$
4. $n\backslash m\backslash a \equiv m\backslash n\backslash a$
5. $n\backslash a \star b \equiv n\backslash(a \star b)$, $n \notin fn(b)$, $\star$ is any of &, | or @
   $a|n\backslash b \equiv n\backslash(a|b)$, $n \notin fn(a)$
6. $n := a \equiv n' := a\{n'/n?\}$, $n' \notin fn(a)$
7. $n\backslash a \equiv n'\backslash a\{n'/n?\}$, $n' \notin fn(a)$
8. $x \rightarrow a \equiv x\{n'/n?\} \rightarrow a\{n'/n?\}$, $n' \notin fn(x, a)$

$\square$

A few words of explanation are in order. The first set of equations simply tells us that concurrent composition is commutative and associative, and that nil contributes nothing. The second equation tells us how to expand recursive agents (i.e., substitute all free occurrences of n in a by $n := a$). The third allows us to discard a restriction of an unused name ($fn(a)$ is the set of *free names* in a). The fourth equation tells us that the order of restriction is unimportant. The fifth equation allows us to expand the scope of a restriction to nearby agents (called *scope extrusion* [25]), if the restricted name is new. The last three equations define α-convertibility for agents. They are needed for substituting local names by globally unique names prior to scope extrusion.

Definition 3 *Communication offers,* denoted by $\xrightarrow{c}$, where c is either v (for input) or $\bar{v}$ (for output), and *Reduction,* written $\longrightarrow$, are induced by the following rules:

$$\textbf{Out}: \frac{}{v \wedge a \xrightarrow{\bar{v}} a} \qquad \textbf{In}: \frac{v \sim x}{x \to a \xrightarrow{v} a\{v/x\}} \qquad \textbf{Conc}: \frac{a \xrightarrow{c} a'}{a\&b \xrightarrow{c} a'\&b}$$

$$\textbf{If}: \frac{a \xrightarrow{c} a'}{a|b \xrightarrow{c} a'} \qquad \textbf{Else}: \frac{a \not\longrightarrow \ , \ a \not\xrightarrow{c} \ , \ b \xrightarrow{c} b'}{a|b \xrightarrow{c} b'}$$

$$\textbf{Apply}: \frac{a \xrightarrow{v} a'}{a@v \longrightarrow a'} \qquad \textbf{Comm}: \frac{a \xrightarrow{v} a', \ b \xrightarrow{\bar{v}} b'}{a\&b \longrightarrow a'\&b'}$$

$$\textbf{Left}: \frac{a \longrightarrow a'}{a \star b \longrightarrow a' \star b} \ [\star \text{ is } \&, \ | \text{ or } @] \qquad \textbf{Right}: \frac{b \longrightarrow b'}{a \star b \longrightarrow a \star b'} \ [\star \text{ is } \&, \ | \text{ or } \backslash]$$

$$\textbf{Struct}: \frac{a \equiv b, \ b \longrightarrow b', \ b' \equiv a'}{a \longrightarrow a'} \quad \square$$

Note that the left argument of the choice operator has priority over the right argument, so we must first be sure (in **Else**) that a cannot be further reduced.

The rules **Apply** and **Comm** define local and remote communication. In an expression such as $a@v$, the agent a is *required* to (eventually) accept the input v, if it can. By rule **Left**, a may first reduce to some form a' before accepting v, but it may not communicate with any external agent until it has done so (since $\xrightarrow{c}$ is not defined for the form $a@v$). If a is incapable of accepting v at any time, then $a@v$ is effectively dead. With remote communication, on the other hand, in $a\&b\&c$, a is free to communicate with either b or c, should their offers match.

As a matter of convenience, we will also overload the operator := to stand for $\stackrel{def}{\equiv}$.

4 Using OC

4.1 Concurrency, Communication and Synchronization

Let us consider first the very simple example of a binary semaphore:

bsem := p→v→bsem | v→bsem

bsem would like to accept as input a p (from an agent claiming a resource) and then accept a v (when the agent releases it). Note that any attempts to release the resource when it has not yet been claimed are discarded.

We may similarly define a printer that accepts print requests:

printer := print→printer

and a couple of clients:

c1 := p∧print∧print∧v∧nil

c2 := p∧print∧print∧v∧nil

Each client attempts to grab the semaphore, communicates twice with the printer, and then releases it.

Now, if we start with the system bsem & printer & c1 & c2, we may reach after the communication of a p the following configuration:

⟶ v→bsem & printer & print∧print∧v∧nil & p∧print∧print∧v∧nil

Note that the second client is unable to claim the resource since it is waiting to send a p that the semaphore is not yet prepared to accept. Let us trace through the rest of the computation.

⟶ v→bsem & printer & print∧v∧nil & p∧print∧print∧v∧nil

⟶ v→bsem & printer & v∧nil & p∧print∧print∧v∧nil

⟶ bsem & printer & p∧print∧print∧v∧nil

⟶ v→bsem & printer & print∧print∧v∧nil

⟶ v→bsem & printer & print∧v∧nil

⟶ v→bsem & printer & v∧nil

⟶ bsem & printer

At this point the system is *stable*, as it can be reduced no further. Note that exactly two possible computation paths were possible, depending on which client grabbed the resource first.

4.2 Composition

So far we have seen only pure synchronization and remote communication. We shall now show how an agent can also be treated as a function.

Let us consider an agent that models the behaviour of the Linda *tuple space* [11]. Linda provides a small set of primitives to allow concurrent processes to communicate and synchronize by writing and reading tuples to a so-called tuple space. A process may write a tuple using the non-blocking *out* primitive, and may read a tuple either destructively with the *in* primitive, or non-destructively with the *rd* primitive. Both read primitives block if no matching tuple exists. The following agent, linda, supports these three primitives:

linda := (out,t?) → (linda & tuple@t)

tuple := t? → ((in,t)∧nil | (rd,t)∧tuple@t)

When linda receives a request to create a new tuple, it replaces itself by a system including a copy of itself and an agent that implements the behaviour of a tuple. tuple is in fact an abstraction over the possible set of tuple values. To instantiate it, the value t must be applied to tuple.

With this agent, we may re-specify our clients of the previous example as follows:

c1 := (in,sem) → print∧print∧(out,sem)∧ nil

c2 := (in,sem) → print∧print∧(out,sem)∧ nil

and the system to evaluate is now:

linda & (out,sem)∧nil & printer & c1 & c2

Note that tuple is not just a function but is in fact an agent. It is a little unusual in that it contains no free names in its input pattern. In a sense, it is an "anonymous" agent in that t? will match any output whatsoever. We can force it to accept a particular communication as input only through the use of @.

In the Linda example, only names were communicated. The following example, of a stack, makes use of agent communication to define an abstraction of a stack:

```
empty := (push,x?) → stack@(x,empty)
stack  := (top?,rest?) → ( (pop,top) ∧ rest | (push,x?) → stack@(x,stack@(top,rest)))
```

The agent stack accepts two values as input: top, which is the value to be popped off, and rest, which is the agent (i.e., stack) to be revealed when the top is popped off. Note that it is imperative that rest be bound to an agent, whereas top may be any value. If we incorrectly try to evaluate stack@(n,m) & (pop,x?)→nil then we will eventually reduce to a term (namely m) which is not an agent. This suggests that communications have sorts, and agents have types associated with them, as is the case in the π calculus [27].

4.3 Encapsulation

Up to now our agents have communicated only through a fixed set of names. Let us now consider the standard example of a sequence of linked agents that implement a queue:

```
queue := (put,x?) → done\(head@(x,done) & tail@done)
head  := (x?,done?) → (get,x)∧done∧nil
tail  := ready? → ( (put,x?) → done\(ready→head@(x,done) & tail@done)
                  | ready → queue )
```

The empty queue can only accept requests to put a new value. When it receives the first value, it turns into a head agent containing this value, linked to a tail that accepts further put requests. head is an abstraction over x, the value to remember, and done, a private name to communicate to the next agent in the queue when it has yielded its value. tail takes as an argument the name of the link to the last head cell in the queue. When the tail receives a new put request, it creates a new head cell that waits to be receive this name as input before being ready to output its value. A new name done is introduced to link the tail to this new head agent. When the tail itself comes to the real head of the queue, it simply becomes an empty queue, since this means there are no more values to get.

Let us see just a few intermediate states resulting from the following system:

queue & (put,a)∧(put,b)∧nil

with (put,a) we reduce to:

```
done\( head@(a,done)
     & tail@done)
& (put,b)∧nil
```

which further reduces to:

```
done\(( (get,a)∧done∧nil
      & (put,x?) → done′\ ( done→head@(x,done′) & tail@done′)
        | done→queue )
      & (put,b)∧nil)
```

With (put,b) we get:

```
done\ ( (get,a)∧done∧nil
      & done′\ ( done→head@(b,done′)
               & tail@done′))
```

which finally reduces to:

```
done\done′\( (get,a)∧done∧nil
           & done→head@(b,done′)
           & (put,x?) → done\ (done′ →head@(x,done) & tail@done)
             | done′ →queue )
```

Note that the local name done in tail is α-converted to done′ to avoid the conflict with the free name done in the expression tail@done. Also, the scope of done has expanded as well to permit the communication (put,b). If a further put were required, the innermost done would have to be α-converted to done″ and migrated outward to permit the communication.

4.4 Numbers

Our examples so far have avoided arithmetic. We could have provided numbers as primitives in our calculus, but it is more satisfying to provide an encoding that allows us to view them just as any other kind of agent. A natural place to look is at the standard encodings into the λ calculus [6].

First, we need Boolean values encoded as agents. Booleans are used in practice for making a choice between two alternatives, so:

true := (a?,b?) → a

false := (a?,b?) → b

With this interpretation, we can also define:

neg := a? → a@(false,true)

and := (a?,b?) → a@(b,a)

or := (a?,b?) → a@(a,b)

So, for example,

neg@true ⟶ false

Our encoding of natural (non-negative) numbers differs only slightly from the standard one. Instead of viewing an expression such as $1 + 2$ as a function + applied to the values 1 and 2, we interpret it as syntactic sugar for applying the tuple $(+, 2)$ to the agent 1, i.e., as $1@(+, 2)$. That is, + is not a function but merely a name serving as a *message selector*. In this way we are later free to define other kinds of agents that are not numbers, but that also understand messages of the form $(+, a)$, exactly as one would when defining new classes in an object-oriented programming language.

We now encode the natural numbers as an abstraction over two values: a Boolean value indicating if the number is 0, and the number's predecessor, if any:

```
nat := (z?,p?) → ( iszero → z
                 | pred → p
                 | succ → nat@(false,nat@(z,p))
                 | (+,n?) → z@(n, (p+(n@succ)))
                 | (×,n?) → z@(0, (n+(p×n)))
                 | ... )
```

where 0 := nat@(true,nil).

Now it is easy to see that 0@iszero ⟶ true and 0@succ@iszero ⟶ false. Addition and multiplication are defined in the usual way. If m is zero then m+n evaluates to n, otherwise it evaluates to p plus the successor of n, where p is m's predecessor. Note how the Boolean value z is used to choose between the two possible continuations.

4.5 Actors

As a final example, let us consider the problem of modelling actors. Actors are computational entities that communicate by asynchronous message-passing [1, 18]. An actor consists of a queue of pending messages and a "behaviour" that accepts and responds to messages. Every actor is associated with a

unique "mail address" used to receive messages. An actor may know the mail addresses of other actors which are its acquaintances. When an actor accepts a message, it can do three things:

1. Create new actors.
2. Send messages to its acquaintances.
3. Specify the replacement behaviour to handle the next message.

The replacement behaviour may be specified at any time, thus permitting an actor to begin processing the next message concurrently with the processing of the current one.

Let us consider Hewitt's standard example of a factorial actor written in a version of Agha's Simple Actor Language, SAL [1, 30].

```
def recFact accept fact:[n,client] ⇒
                   become self ;
                   if (n=0)
                   then send result:[1] to client
                   else  let c = new factCust with [n,client]
                         in { send fact:[n−1,c] to self }
def factCust with [n,c] accept result:[k] ⇒ send result:[n×k] to c
```

The behaviour recFact accepts requests of the form fact:[n,client] to compute the factorial of n and eventually causes the message: result:[*factorial of n*] to be sent back to the client. If the request is for the factorial of 0, the factorial actor responds immediately. Otherwise it dynamically creates a customer whose acquaintances are n and client, and it sends itself a request to compute the factorial of n−1 and send the result to the customer. The customer will eventually receive this result, compute the product of n and the factorial of n−1 and it will send the value to the client. For a request to compute n factorial, then, recFact will end up creating n customers, thus simulating an execution stack. Since recFact maintains no state information itself (it uses the customer to remember the original client) it immediately specifies its replacement as self to begin processing the next message. As a consequence, the factorial actor may service multiple requests concurrently.

A plausible and straightforward translation of recFact into OC is as follows:

```
recFact := id? → (id,fact,n?,client?) → ( recFact@id
                                        & n @ iszero @
                                            ( ((client,result,1)∧nil),
                                              (new\( factCust@(new,n,client)
                                                   & (id,fact,n-1,new)∧nil))))
factCust := (id?,n?,c?) → (id,result,k?) → (c,result,n×k)∧nil
```

A behaviour is simply an abstraction whose arguments are the actor's id and its acquaintances, if any. Actor messages are represented as tuples in which the first argument is the actor's id, the second the message selector, and the remaining arguments the contents of the message. An actor may become self by spawning a copy of its behaviour, as recFact does immediately upon receiving a request. A new actor is created by introducing a new name, new, and binding it to a new instance of factCust.

Note that we have opted for a lazy interpretation. The agents that will evaluate the resulting arithmetic expressions are passed around rather than the final results.

5 Our Requirements, Reconsidered

Although a full treatment of how to model features of various concurrent object-oriented languages is beyond the scope of this paper (see, however, [34] for a CCS framework for modelling such languages), let us briefly review our informal requirements to see how OC addresses them.

First, we wish to view objects as agents encapsulating services. Let us suppose that all remote communications take one of the following two forms:

1. (request, *oid*, *selector*, *contents*, *reply-address*)
2. (reply, *reply-address*, *reply*)

Then, to send a message m with contents a to object x and obtain a reply, one may use the following protocol:

rid\((request,x,m,a,rid)∧(reply,rid,val?)→...)

More precisely, since the value obtained should be passed on to some expression continuation, we may abstract the calling sequence as follows:

call := (x?,m?,a?,econt?) → rid\((request,x,m,a,rid) ∧ (reply,rid,val?) → econt@val))

Thus, to call x as before with the continuation c, we simply instantiate call@(x,m,a,c).

State change can clearly be modelled, as shown even in the semaphore examples. Unique object identifiers are provided by restriction and scope extrusion. As objects are agents, they have full control over their communications. Non-uniform service availability can be readily specified, as shown by the stack and queue examples. Internal concurrency and multiple pending requests are illustrated by the recursive factorial agent. The ability to generate unique reply addresses is essential for managing multiple pending requests. Finally, objects as systems of more primitive objects and as functional compositions of various abstractions has been shown in several of the examples.

We have not yet demonstrated how inheritance and overriding can be accurately modelled, but we conjecture that the approach of Cook [13] will work well here. To give a flavour of this approach, let us abstract from the stack example given earlier and introduce *generators* for stacks:

```
emptyGen := sub? → ( sub@nil
                   | (push,x?) → stackGen@sub@(x,emptyGen@sub))

stackGen := sub? → (top?,rest?) →
                        ( sub@(top,rest)
                        | (pop,top) ∧ rest
                        | (push,x?) → stackGen@sub@(x,stackGen@sub@(top,rest)))
```

The new variable sub may be bound to an abstraction of a new service in order to extend or override the services already provided. It is provided with the top of the stack and the rest of the stack as parameters. Now, to obtain the original stack from the generator, we may define:

empty := emptyGen@nil

Note that nil@(top,rest) behaves like nil, and so adds nothing to the behaviour of emptyGen or stackGen. Now suppose that we want to add a new service peek that allows a client to peek at the top of the stack without popping the value off. We may define the new service as:

peek := (top?,rest?) → (peek,top) ∧ stackGen@peek@(top,rest)

and the new empty stack as:

newempty := emptyGen@peek

Of course peek@nil is incapable of any action, which is correct for the case when the stack is empty.

The same approach could be use to define natGen, a generator for natural numbers. New message selectors to be understood by numbers could thus be defined, and existing ones could be overridden, as long as the parameter sub appears as the first choice.

6 Some Theoretical Considerations

Although we have provided operational semantics for OC, we have not shown that we can prove any interesting properties about OC agents, such as when two expressions denote the same behaviour, nor have we shown that any standard results from other process calculi carry over to OC. As we hope to recover as much as possible from previous work, let us briefly resume the differences between OC and other process calculi (mainly the π calculus), and summarize the problems to be resolved.

- *Tuple-based communication:* communications in OC are tuples of names and agent terms. Synchronization is by matching of free names to free names and names or agents to locally bound names (introduced by a ?). Free names in input patterns serve essentially the same function as ports in the π calculus, except that one may synchronize with respect to several free names instead of just a single port name. It is also possible to have input patterns containing no free names at all, permitting anonymous ("port-less") communications.
- *Functions are agents:* although several other higher-order process calculi have been proposed, and accurate encodings of the λ calculus into first-order process calculi have been demonstrated, to our knowledge only OC has proposed the unification of functions and processes through a single abstraction mechanism. (Though it should be noted that Boudol [8] has proposed a *cooperation* operator $\odot$ which can be used to similar effect: $p \odot q$ forces p and q to interact until one of them is exhausted (i.e., equal to nil). Then a@v can be expressed by a$\odot$(v$\wedge$nil).)
- *Left-preferential choice:* the choice operator of OC is purely deterministic, preferring left-hand interactions to right-hand ones in the case of conflicts. This suggests that we lose some expressive power with respect to the summation operator of CCS or the π calculus, but in practice we are interested only in non-determinism arising from concurrency. Left-preferential choice makes it possible to override interactions in composing new agents much in the same way that we can override default behaviour when inheriting from a superclass in an object-oriented language. (A similar effect can be obtained using the restriction operator of CCS [34]). Furthermore, choice in OC is insensitive to internal actions, which is essential when expressing summands as compositions of other agents. We also suspect that this will help in developing a behavioural equivalence which is also a congruence, since we wish to distinguish agents only on the basis of their visible interactions.

There appears to be a faithful translation of the π calculus into OC. We offer (without proof) the following mapping (taking the version of the π calculus presented in [26]):

$$\begin{array}{lcl}
\Pi(\bar{x}yP) & = & (x,y) \wedge \Pi(P) \\
\Pi(x(y).P) & = & (x,y?) \rightarrow \Pi(P) \\
\Pi(0) & = & \mathsf{nil} \\
\Pi(P|Q) & = & \Pi(P)\&\Pi(Q) \\
\Pi(!P) & = & n := n\&\Pi(P), \quad n \notin fn(\Pi(P)) \\
\Pi((y)P) & = & y\backslash\Pi(P)
\end{array}$$

The reverse problem, of demonstrating a translation of OC to the π calculus appears to be more difficult, especially as there is no simple way to simulate the tuple-based communication of OC using ports. For example, in the system

(a,b)$\wedge$nil & (a,x?)$\rightarrow$nil & (x?,b)$\rightarrow$nil

a and b are used simultaneously as ports and as values to be passed. For arbitrary communications, any subset of the free names of the message may be needed to match an input pattern. This suggests that there will be an explosion of port names to model the various possible matchings. We expect, however, that an encoding exists, and that it can be closely modelled after the translation of the higher-order π calculus, HOπ, into the first-order one.

The translation of the lazy λ calculus is straightforward:

$$\begin{array}{lcl} \Lambda(\lambda x.E) & = & x? \rightarrow \Lambda(E) \\ \Lambda(E_1 E_2) & = & \Lambda(E_1) @ \Lambda(E_2) \end{array}$$

Eager evaluation cannot be directly expressed in OC since there are no rules for reducing active subexpressions of v in $a@v$ (or, for that matter, in $v \wedge a$). We could alter **Left** and **Right** to permit eager evaluation, but this would introduce an unwanted aspect of non-determinism since the Church-Rosser property (i.e., that evaluation order does not matter) does not hold in general for OC. Intuitively it seems as if it should hold, since any reduction within an output pattern is purely local and independent of context, but if the reductions are due to **Comm**, then we may lose something. Consider, for example: (x?→x)@(n→nil & n∧nil). Lazy evaluation reduces this to (n→nil & n∧nil), which permits further interactions with the environment, whereas eager evaluation would reduce it in two steps to nil. As a consequence, we must either explicitly evaluate expressions *before* communicating them, if strictness is desired, or we must demonstrate for a particular case that evaluation order does not matter.

Note that these translations suggest that OC is actually a merge of the π and λ calculi, extended by | and $\sim$. Furthermore, since we can simulate the λ calculus, we may express Curry's fixed-point combinator as an agent:

fix $\stackrel{def}{\equiv}$ f? → (x? → f@(x@x)) @ (x? → f@(x@x))

which means that := is not strictly needed. For example, we can then express the ! operator of the π calculus as

fix@(bang? → p? → (p & bang@p))

The principle problems to be explored are:

- Is there a translation of OC to the π calculus (or to another established process calculus) that preserves its operational semantics? If not, what conclusions can be drawn?
- What notion of behavioural equivalence is appropriate for OC? Since we should also factor out equivalent agents appearing in communications, perhaps the higher-order bisimulation of CHOCS [38] is called for.
- Can we develop a type theory for agents that allows us to reason about compositionality? How is type conformance related to the (stronger) notion of behavioural equivalence?
- Under what circumstances are eager and lazy evaluation equivalent for reducible terms appearing in communications?

7 Concluding Remarks

We have put forward some informal requirements for a calculus suitable for specifying the behaviour of active objects, and we have presented the syntax and operational semantics of a proposed object calculus, OC. We have illustrated the use of OC through a series of examples that highlights the requirements we have posed. The interesting formal properties of OC are unknown as yet, but we have indicated some of the key differences with existing process calculi and outlined a program of topics for further study.

In a larger context, we wish to use OC to explore:

1. *Integration of language features* for concurrent object-oriented languages, particularly reuse features and concurrency mechanisms.
2. *A type theory for active objects* in which a type is a specification of a "software contract" between an object and its clients, and a subtype is just a stronger specification.
3. *Language design and prototyping* by translation to executable specifications in OC.

An interpreter for the version of OC presented here has been implemented in Prolog. The implementation is very similar to the earlier one of Abacus [30], but all dependency on Prolog variables has been eliminated since the semantics of unification in Prolog is incompatible with that of communication in OC. In particular, α-conversion and variable substitution are directly implemented. If and when OC stabilizes, a more efficient implementation is planned as the Prolog version is impractical for large examples.

In the long term, we hope to use OC as the foundation for a new programming language — a "pattern language" for active objects — in which applications are constructed by composing reusable software patterns, much in the way that architectural designs can be composed from established architectural patterns [2]. Such a language would be used in two complementary ways: first, to design and develop reusable patterns of objects, and second, to compose applications from pre-designed patterns [32]. An object calculus is the first step to defining such a pattern language.

Acknowledgements

Many thanks to Michael Papathomas and Laurent Dami who offered considerable improvements to the presentation of this paper.

References

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

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

[3] P. America, J. de Bakker, J.N. Kok and J. Rutten, "Operational Semantics of a Parallel Object-Oriented Language," in *Proceedings POPL '86*, pp. 194-208, St. Petersburg Beach, Florida, Jan 13-15, 1986.

[4] P. America, J. de Bakker, J. Kok and J. Rutten, "Denotational Semantics of a Parallel Object-Oriented Language," Information and Computation, vol. 83, no. 2, pp. 152-205, Nov 1989.

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

[6] H.P. Barendregt, *The Lambda Calculus – Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, 103, North-Holland, 1984, (Revised edition).

[7] G. Berry and G. Boudol, "The Chemical Abstract Machine," in *Proceedings POPL '90*, pp. 81-94, San Francisco, Jan 17-19, 1990.

[8] G. Boudol, "Towards a Lambda-Calculus for Concurrent and Communicating Systems," in *Proceedings TAPSOFT '89*, ed. Díaz and Orejas, LNCS 351, pp. 149-161, Springer-Verlag, 1989.

[9] G. Bracha and Wm. Cook, "Mixin-based Inheritance," ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP '90, vol. 25, no. 10, pp. 303-311, Oct 1990.

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

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

[12] W.D. Clinger, "Foundations of Actor Semantics," AI-TR-633, MIT Artificial Intelligence Laboratory, May 1981.

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

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

[15] U. Engberg and M. Nielsen, "A Calculus of Communicating Systems with Label Passing," DAIMI PB-208, University of Aarhus, 1986.

[16] R. Helm, I.M. Holland and D. Gangopadhyay, "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems," ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP '90, vol. 25, no. 10, pp. 169-180, Oct 1990.

[17] A.V. Hense, "Denotational Semantics of an Object Oriented Programming Language with Explicit Wrappers," Technical report A11/90, FB 14, Universität des Saarlandes, Nov. 5, 1990, submitted for publication.

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

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

[20] K. Honda and M. Tokoro, "An Object Calculus for Asynchronous Communication," in *Proceedings ECOOP '91*, ed. P. America, LNCS 512, pp. 133-147, Springer-Verlag, Geneva, Switzerland, July 15-19, 1991.

[21] K. Honda and M. Tokoro, "On Asynchronous Communication Semantics," in *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, ed. M. Tokoro, O. Nierstrasz, P. Wegner, LNCS, Springer-Verlag, Geneva, Switzerland, July 15-16, 1991, to appear.

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

[23] J. Meseguer, "A Logical Theory of Concurrent Objects," ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP '90, vol. 25, no. 10, pp. 101-115, Oct 1990.

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

[25] R. Milner, J. Parrow and D. Walker, "A Calculus of Mobile Processes, Parts I and II," Reports ECS-LFCS-89-85 and -86, Computer Science Dept., University of Edinburgh, March 1989.

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

[27] R. Milner, "Sorts and Types in the π Calculus," manuscript (RM15), Computer Science Dept., University of Edinburgh, December 1990.

[28] F. Nielson, "The Typed Lambda-Calculus with First-Class Processes," in *Proceedings PARLE '89, Vol II*, ed. E. Odijk, J-C. Syre, LNCS 366, pp. 357-373, Springer Verlag, Eindhoven, June 1989.

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

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

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

[32] O. Nierstrasz, "The Next 700 Concurrent Object-Oriented Languages – Reflections on the Future of Object-Based Concurrency," in *Object Composition*, ed. D.C. Tsichritzis, pp. 165-187, Centre Universitaire d'Informatique, University of Geneva, June 1991, Submitted for publication.

[33] 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.

[34] M. Papathomas, "A Unifying Framework for Process Calculus Semantics of Concurrent Object-Based Languages," in *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, ed. M. Tokoro, O. Nierstrasz, P. Wegner, LNCS, Springer-Verlag, Geneva, Switzerland, July 15-16, 1991, to appear.

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

[36] D. Sangiorgi, forthcoming Ph.D. thesis, Computer Science Dept., University of Edinburgh, 1992.

[37] D. Taenzer, M. Ganti and S. Podar, "Problems in Object-Oriented Software Reuse," in *Proceedings ECOOP '89*, pp. 25-38, Cambridge University Press, Nottingham, July 10-14, 1989.

[38] B. Thomsen, "A Calculus of Higher Order Communicating Systems," in *Proceedings POPL '89*, pp. 143-154, Austin, Texas, Jan 11-13, 1989.