Inheritance in the Join Calculus (extended abstract)

Cédric Fournet[1], Cosimo Laneve[2], Luc Maranget[3], and Didier Rémy[3]

[1] Microsoft Research, 1 Guildhall Street, Cambridge, U.K.
[2] Dipartimento di Scienze dell'Informazione, Università di Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy
[3] INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex France.

Abstract. We propose an object-oriented calculus with internal concurrency and class-based inheritance that is built upon the join calculus. Method calls, locks, and states are handled in a uniform manner, using labeled messages. Classes are partial message definitions that can be combined and transformed. We design operators for behavioral and synchronization inheritance. Our model is compatible with the JoCaml implementation of the join calculus.

1 Introduction

Object-oriented programming has long been praised as favoring abstraction, incremental development, and code reuse. Objects can be created by instantiating definition patterns called *classes*, and in turn complex classes can be built from simpler ones. To make this approach effective, the assembly of classes should rely on a small set of operators with a clear semantics and should support modular proof techniques. In a concurrency setting, such promises can be rather hard to achieve.

The design and the implementation of concurrent object-oriented languages, *e.g.* [2,20,1,4], has recently prompted the investigation of the theoretical foundations of concurrent objects. Several works provide encodings of objects in process calculi [19,18,12,5] or, alternatively, supplement objects with concurrent primitives [16,3,11]. Those works promote a unified framework for reasoning about objects and processes, but they do not address the composition of object definitions or its typechecking.

In this work, we model concurrent objects in a simple process calculus—a variant of the *join calculus* [7,6], we design operators for behavioral and synchronization inheritance, and we give a type system that statically enforces basic safety properties.

The join calculus is a simple name-passing calculus, related to the pi calculus but with a functional flavor. It is the core of a distributed programming language, currently implemented as an extension of ML [8,13]. In the join calculus, communication channels are statically scoped: when channels are created, their definition provides a set of *reaction rules* that specify, once for all, how messages sent on these names will be synchronized and processed. Although the join calculus does not have a primitive notion of object, definitions encapsulate the details of synchronization, much as concurrent objects.

Applying the well-known objects-as-records paradigm to the join calculus, we obtain a simple language of objects with asynchronous message passing. Method calls, locks, and states are handled in a uniform manner, using labeled messages. There is no primitive notion of functions, calling sequences, or threads (they can all be encoded using continuation messages). Our language—the *objective join calculus*—allows fine-grain internal concurrency, as each object may send and receive several messages in parallel.

For every object of our language, message synchronization is defined and compiled as a whole. This allows an efficient compilation of message delivery into automata [14] and simplifies reasoning on objects. However, the static definition of behavior can be overly restrictive for the programmer. This suggests some compile-time mechanism for assembling partial definitions. To this end, we promote partial definitions into classes. Classes can be combined and transformed to form new classes. They can also be closed to create objects.

The class language is layered on top of the core objective calculus, with a semantics that reduces classes into plain object definitions. We thus retain strong static properties for all objects at run-time. Some operators are imported from sequential languages and adapted to a concurrent setting. For instance, multiple inheritance is expressed as a disjunction of join definitions, but some disjunctions have no counterpart in a sequential language. In addition, we propose a new operator, called *selective refinement*. Selective refinement applies to a parent class, and rewrites the parent reaction rules according to their synchronization patterns. Selective refinement treats synchronization concretely, but it handles the parent processes abstractly. Our approach is compatible with the JoCaml implementation of the join calculus [13], which already singles out synchronization patterns using concrete compile-time representation, and, on the contrary, compiles behaviors into functional closures.

Our design of the class language follows from common programming patterns in the join calculus. We motivate it further by coding some standard problematic examples that mix synchronization and inheritance.

The language is equipped with a polymorphic type system, in the style of [9]; in addition to basic safety properties, the type system also enforces privacy. The formal presentation of both dynamic and static semantics, the soundness results, and their proofs are omitted from this extended abstract. They can be found in the full paper [10].

The paper is organized as follows. In section 2, we present the objective join calculus and develop a few examples. In section 3, we supplement the language with classes. In section 4, we provide more involved examples of inheritance and concurrency. In section 5, we discuss related works and possible extensions.

2 The objective join calculus

Getting started. The basic operation of our calculus is asynchronous message passing in object style. For instance, the process $\mathit{out}.\mathit{print\_int}(n)$ sends a message with label *print_int* and with content n to an object named *out*, meant to print integers on the terminal.

Accordingly, the definition of an object describes how messages received on some labels can trigger processes. For instance,

$$\mathsf{obj}\ \mathit{continuation} = \mathit{reply}(x) \triangleright \mathit{out}.\mathit{print\_int}(x)$$

defines an object that reacts to messages on *reply* by printing their content on the terminal. Another example is the rendez-vous, or synchronous buffer:

$$\mathsf{obj}\ \mathit{sbuffer} = \mathit{get}(r)\ \&\ \mathit{put}(a,s) \triangleright r.\mathit{reply}(a)\ \&\ s.\mathit{reply}()$$

The object *sbuffer* has two labels *get* and *put*; it reacts to the *simultaneous presence* of one message on each of these labels by passing a message to the continuation r, with label *reply* and contents a, and passing an empty message to s. (Object r may be the previously-defined *continuation*; object s is another continuation taking no argument on *reply*.) As regards the syntax, concurrent execution and message synchronization are expressed in a symmetric manner using the same infix operator &. Also, the calculus is polyadic, *i.e.*, messages carry tuples of values.

Some labels may convey messages representing the internal state of an object, rather than an external method call. This is the case of label *Some* in the following unbounded, unordered, asynchronous buffer:

$$\begin{array}{l}
\mathsf{obj}\ \mathit{abuffer} = \mathsf{self}(z) \\
\quad \mathit{put}(a,r) \triangleright r.\mathit{reply}()\ \&\ z.\mathit{Some}(a) \\
\mathsf{or}\ \mathit{get}(r)\ \&\ \mathit{Some}(a) \triangleright r.\mathit{reply}\ (a)
\end{array}$$

The object *abuffer* can react in two different ways: a message (a, r) on *put* may be consumed by storing the value a in a self-inflicted message on *Some*; alternatively, a message on *get* and a message on *Some* may be jointly consumed, and then the value stored on *Some* is sent to the continuation received on *get*. The indirection through *Some* makes *abuffer* behave asynchronously: messages on *put* are never blocked, even if no message is ever sent on *get*. As regards the syntax, the prefix $\mathsf{self}(z)$ explicitly binds the name z to the defined object.

In the example above, the messages on label *Some* encode the state of *abuffer*. The following definition illustrates a tighter management of state that implements a one-place buffer:

$$\begin{array}{l}
\mathsf{obj}\ \mathit{buffer} = \mathsf{self}(z) \\
\quad \mathit{put}(a,r)\ \&\ \mathit{Empty}() \triangleright r.\mathit{reply}()\ \&\ z.\mathit{Some}(a) \\
\mathsf{or}\ \mathit{get}(r)\ \ \&\ \mathit{Some}(a) \triangleright r.\mathit{reply}(a)\ \&\ z.\mathit{Empty}() \\
\mathsf{init}\ \ \mathit{buffer}.\mathit{Empty}()
\end{array}$$

Such a buffer can either be empty or contain one element. The state is encoded as a message pending on *Empty* or *Some*, respectively. Object *buffer* is created empty, by sending a first message on *Empty* in the (optional) init part of the

Fig. 1. Syntax for the core object calculus

$P ::=$	**Processes**
$\quad \mathsf{0}$	null process
$\quad x.M$	message sending
$\quad P_1 \mathbin{\&} P_2$	parallel composition
$\quad \mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$	object definition
$D ::=$	**Definitions**
$\quad M \triangleright P$	reaction rule
$\quad D_1\ \mathsf{or}\ D_2$	disjunction
$M ::=$	**Patterns**
$\quad \ell(\widetilde{u})$	message
$\quad M_1 \mathbin{\&} M_2$	synchronization

obj construct. As opposed to *abuffer* above, a *put* message is blocked when the buffer is not empty.

To keep the *buffer* object consistent, there should be a single message pending on either *Empty* or *Some*. This invariant holds as long as external users cannot send messages on these labels directly. In the full paper [10], we describe a refined semantics and a type system that distinguishes private labels such as *Empty* and *Some* from public labels, and restrict access to private labels. In the examples, private labels conventionally bear an initial capital letter.

Once private labels are hidden, each variant of our buffer provides the same interface to the outside world (two methods labeled *get* and *put*) but their concurrent behaviors are very different.

Syntax. We use two disjoint countable sets of identifiers for object names $x, z, u \in \mathcal{O}$ and for labels $\ell \in \mathcal{L}$. Tuples of names are written ${x_i}^{i \in I}$ or simply $\widetilde{x}$. The grammar of the *objective join calculus* (without classes) is given in Figure 1; it has syntactic categories for processes P, definitions D, and patterns M. We abbreviate $\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P\ \mathsf{in}\ Q$ by omitting $\mathsf{self}(z)$ when z does not occur free in D and omitting $\mathsf{init}\ P$ when P is $\mathsf{0}$.

A reaction rule $M \triangleright P$ associates a pattern M with a guarded process P. Every message pattern $\ell(\widetilde{u})$ in M binds the object names $\widetilde{u}$ with scope P. We require that every pattern M guarding a reaction rule be linear; that is, each label and each name appears at most once in M. This will be enforced by typing. In addition, the object definition $\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$ binds two names x and z to D. The scope of x is the processes P_1 and P_2; the scope of z is every guarded process in D. Free names in processes and definitions, written $\mathit{fn}(\cdot)$, are defined according to these binders. Terms are taken modulo renaming of bound names (or α-conversion).

The reduction relation on processes is defined using a reflexive chemical abstract machine; it appears in the full paper.

3 Inheritance and concurrency

We now extend the calculus of concurrent objects with classes and inheritance. The behavior of objects in the join calculus is statically defined: once an object is created, it cannot be extended with new labels or with new reaction rules synchronizing existing labels. Instead, we provide this flexibility at the level of classes. Our operators on classes can express various object paradigms, such as method overriding (with late binding) or method extension. As regards concurrency, these operators are also suitable to define synchronization policies in a modular manner.

Deriving a concurrent class. We introduce the syntax for classes in a series of simple examples. We begin with a class *buffer* behaving as the one-place buffer of Section 2:

```
class buffer = self(z)
    get(r)  & Some(a) ▷ r.reply(a) & z.Empty()
or  put(a,r) & Empty() ▷ r.reply() & z.Some(a)
```

The class *buffer* can be used to create objects:

```
obj b = buffer init b.Empty()
```

Assume that, for debugging purposes, we want to log the buffer content on the terminal. Our first solution uses an explicit *log* label.

```
class logged_buffer = self(z)
    buffer
or  log() & Some(a) ▷ out.print_int(a) & z.Some(a)
or  log() & Empty() ▷ out.print_string("Empty") & z.Empty()
```

The class above is a disjunction of an inherited class and of additional reaction rules. The intended meaning of disjunction is that reaction rules are cumulated, yielding competing behaviors for messages on labels that appear in several disjuncts. The order of the disjuncts does not matter. The programmer that writes *logged_buffer* must have some knowledge of the parent class *buffer*, namely the use of private labels *Some* and *Empty* for representing the state.

Other possible debugging information is the synchronous log of all messages that are consumed on *put*. This is done by selecting the patterns in which *put* occurs and adding a printing message to the corresponding guarded processes:

```
class logged_buffer_bis =
    match buffer with
      put(a,r) ⇒ put(a,r) ▷ out.print_int(a)
    end
```

The match construct can be understood by analogy with pattern matching *à la* ML, applied to the reaction rules of the parent class. In this example, every reaction rule from the parent *buffer* whose synchronization pattern contains the label *put* is replaced in the derived *logged_buffer_bis* by a rule with the same synchronization pattern (since *put* appears on both sides of ⇒) and with the original guarded process in parallel with the new printing message (the original guarded process and the & are left implicit in the match syntax). Every other parent rule is kept unchanged. Hence, the above definition behaves as the following definition:

$$\begin{array}{l}
\mathsf{class}\ \mathit{logged\_buffer\_bis}\ =\\
\quad \mathit{get}(r)\ \&\ \mathit{Some}(a) \triangleright\ r.\mathit{reply}(a)\ \&\ z.\mathit{Empty}()\\
\mathsf{or}\ \mathit{put}(a,r)\ \&\ \mathit{Empty}() \triangleright\ r.\mathit{reply}()\ \&\ z.\mathit{Some}(a)\ \&\ \mathit{out}.\mathit{print\_int}(a)
\end{array}$$

Yet another kind of debugging information is a log of *put* attempts:

$$\begin{array}{l}
\mathsf{class}\ \mathit{logged\_buffer\_ter}\ = \mathsf{self}(z)\\
\quad \mathsf{match}\ \mathit{buffer}\ \mathsf{with}\\
\qquad \mathit{put}(a,r) \Rightarrow \mathit{Parent\_put}(a,r) \triangleright O\\
\quad \mathsf{end}\\
\mathsf{or}\ \mathit{put}(a,r) \triangleright \mathit{out}.\mathit{print\_int}(a)\ \&\ z.\mathit{Parent\_put}(a,r)
\end{array}$$

In this case, the match construct performs a renaming of *put* into *Parent_put* in the patterns of selected rules, without affecting their guarded processes. The net effect is similar to parent method overriding, with the new *put* calling the parent one and a late-binding semantics for guarded processes.

The examples above illustrate that the very idea of class refinement is less abstract in a concurrent setting than in a sequential one. In the first *logged_buffer* example, logging the buffer state requires knowledge of how this state is encoded; otherwise, some states might be forgotten or logging might lead the buffer into deadlock. The other two examples expose another subtlety: in a sequential language, the distinction between logging *put* attempts and *put* successes is irrelevant. Thinking in terms of sequential object invocations, one may be unaware of the concurrent behavior of the object, and thus write *logged_buffer_ter* instead of *logged_buffer_bis*.

Syntax. The language with classes extends the core calculus of section 2; its grammar is given in Figure 2. Classes are taken up to the associative-commutative laws for disjunction or. We use two additional sets of identifiers for class names $c \in \mathcal{C}$ and for sets of labels $L \in 2^{\mathcal{L}}$. Such sets L are used to represent abstract classes that declare the labels in L but do not define them.

Join patterns J generalize the syntactic category of patterns M given in Figure 1 with an or operator that concisely represents alternative synchronization patterns. Join patterns are taken up to simple equivalence laws: & and or are associative-commutative, and & distributes over or. Hence, every join pattern J can be written as a non-empty alternative of patterns $\mathsf{or}_{i \in I}\, M_i$, and the reaction rule $(\mathsf{or}_{i \in I}\, M_i) \triangleright P$ behaves as $\mathsf{or}_{i \in I}(M_i \triangleright P)$.

Selection patterns K are either join patterns or the empty pattern $\mathsf{0}$. Their normal forms are of the form above, except that I can be empty. We always assume that patterns J and K meet the following well-formed conditions:

1. All conjuncts M_i in the normal form of K are linear (as defined in section 2) and bind the same names. By extension, we say that K binds the names $\mathit{fn}(M_i)$ bound in every M_i, and write $\mathit{fn}(K)$ for these names.
2. In a refinement clause $K_1 \Rightarrow K_2 \triangleright P$, the pattern K_1 is either M or $\mathsf{0}$, and the pattern K_2 binds at least the names of K_1 ($\mathit{fn}(K_1) \subseteq \mathit{fn}(K_2)$).

Fig. 2. Syntax for classes

$P ::=$	**Processes**
$\ldots$	(as in Figure 1)
$\mathsf{obj}\ x = \mathsf{self}(z)\ C\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$	object definition
$\mathsf{class}\ c = \mathsf{self}(z)\ C\ \mathsf{in}\ P$	class definition
$C ::=$	**Classes**
c	class variable
L	abstract class
$J \triangleright P$	reaction rule
$C_1\ \mathsf{or}\ C_2$	disjunction
$\mathsf{self}(x)\ C$	self binding
$\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$	selective refinement
$S ::=$	**Refinement clauses**
$(K_1 \Rightarrow K_2 \triangleright P) \mid S$	refinement sequence
$\emptyset$	empty refinement
$J ::=$	**Join patterns**
$\ell(\widetilde{u})$	message
$J_1\ \&\ J_2$	synchronization
$J_1\ \mathsf{or}\ J_2$	alternative
$K ::=$	**Selection patterns**
$\mathsf{0}$	empty pattern
J	join pattern

As described in section 2, binders for object names include object definitions (binding the defined object to name x and self name z) and patterns (binding the received names). In a reaction rule $J \triangleright P$, the join pattern J binds $\mathit{fn}(J)$ with scope P. In a refinement clause $K_1 \Rightarrow K_2 \triangleright P$, the selection pattern K_1 binds $\mathit{fn}(K_1)$ with scope K_2 and P; the modification pattern K_2 binds $\mathit{fn}(K_2) \setminus \mathit{fn}(K_1)$ with scope P. Finally, the self renaming $\mathsf{self}(x)\ C$ binds the object name x with scope C. Class definitions $\mathsf{class}\ c = C\ \mathsf{in}\ P$ are the only binders for class names c, with scope P. Processes, classes, and reaction rules are taken up to α-conversion.

Labels don't have scopes. Join patterns J declare the labels appearing in their message. Classes C declare the labels of their reaction rules. Abstract classes trivially declare their labels. Finally, selective refinements declare labels appearing either in the parent class or in a refinement clause.

Class expressions are simplified by means of a reduction semantics, that allows to obtain processes in the core calculus without classes. These reduction sementics (see the full paper [10]) has been designed to support separate compilation of classes.

4 Inheritance anomaly

As remarked by many authors, the classical point of view on class abstraction—method names and signatures are known, method bodies are abstract—does not mix well with concurrency. More specifically, the signature of a parent class does not usually convey any information on its synchronization behavior. As a result, it is often awkward, or even impossible, to refine a concurrent behavior using inheritance. (More conservatively, object-oriented languages with plain concurrent extensions usually require that the synchronization properties be invariant through inheritance, e.g., that all method calls be synchronized. This strongly constrains the use of concurrency.) This well-known problem is often referred to as the *inheritance anomaly.* Unfortunately, inheritance anomaly is not defined formally, but by means of problematic examples.

In [15] for instance, Matsuoka and Yonezawa identify three patterns of inheritance anomaly. For each pattern, they propose a refinement of the class language that suffices to express the particular synchronization property at hand: they identify the parts of the code that controls synchronization in the parent class (which is otherwise hidden in the body of the inherited methods); they express this "concurrency control" in the interface of the class; and they rely on the extended interface to refine synchronization in the definition of subclasses. In principle, it should be possible to fix any particular anomaly by enriching the class language in an ad hoc manner. However, the overall benefits of this approach are unclear.

Our approach is rather different: we started from a core calculus of concurrency, rather than programming examples, and we are primarily concerned with the semantics of our inheritance operators. Tackling the three patterns of inheritance anomaly of [15], as we do in this section, appears to be a valuable test of our design.

We consider the same running example as Matsuoka and Yonezawa: a FIFO buffer with two methods *put* and *get* to store and retrieve items. We also adopt their taxonomy of inheritance anomaly: inheritance induces desirable modifications of "acceptable states" [of objects], and a solution is a way to express these modifications.

In the following examples, we use a language extended with basic datatypes. Booleans and integers are equipped with their usual operations. Arrays are created by create(n), that gives an uninitialized array of size n. The size of an array A is given by $A.size$. Finally, the array $A[i]\leftarrow v$ is obtained from A by overwriting its i-th entry with value v.

The FIFO buffer of [15] can then be written as follows:

```
class buff = self (z)
    put(v,r) & (Empty(A, i, n) or Some(A, i, n)) ▷
        r.reply() & z.Check(A[(i+n) mod A.size] ← v, i, n+1)
or get(r) & (Full(A, i, n) or Some(A, i, n)) ▷
        r.reply(A[i]) & z.Check(A, (i+1) mod A.size, n−1)
or Check(A,i,n) ▷
        if  n = A.size then z.Full(A, i, A.size)
```

```
      else if n = 0 then z.Empty(A, 0, 0)
      else z.Some(A, i, n)
  or Init(size) ▷ z.Empty(create(size),0,0)
```

The state of the buffer is encoded as a message with label *Empty*, *Some*, or *Full*. The buffer may react to messages on *put* when non-full, and to messages on *get* when non-empty; this is expressed in a concise manner using the or operator in patterns. Once a request is accepted, the state of the buffer is recomputed by sending an internal message on *Check*. As *Check* appears alone in a join pattern, message sending on *Check* acts like a function call, which could be in-lined by an optimizing compiler.

Partitioning of acceptable states. The class *buff2* supplements *buff* with a new method *get2* that atomically retrieves two items from the buffer. For simplicity, we assume here $size > 2$.

Since *get2* succeeds when the buffer contains two elements or more, the buffer state needs to be refined. Furthermore, since for instance a successful *get2* may disable *get* or enable *put*, the addition of *get2* has an impact on the "acceptable states" of methods *get* and *put*, which are inherited from the parent *buff*. Therefore, label *Some* is not detailed enough and is replaced with two labels *One* and *Many*. *One* represents the state when the buffer holds exactly one item; *Many* represents a state with two items or more in the buffer.

```
  class buff2 = self(z)
     get2(r) & (Full(A,i,n) or Many(A, i, n)) ▷
           r.reply(A[i], A[(i+1) mod A.size])
      &  z.Check(A, (i+2) mod A.size, n−2)
  or match buff with
     Some(A, i, n) ⇒ (One(A, i, n) or Many(A, i, n)) ▷ O end
  or Some(A, i, n) ▷
        if n > 1 then z.Many(A, i, n) else z.One(A, i, n)
```

In the program above, a new method *get2* is defined, with its own synchronization condition. The new reaction rule is cumulated with those of *buff*, using a selective refinement that substitutes "*One*(...) or *Many*(...)" for every occurrence of "*Some*(...)" in a join pattern. The refinement eliminates *Some* from any inherited pattern, but it does not affect occurrences of *Some* in inherited guarded processes: the parent code is handled abstractly, so it cannot be modified. Instead, the new class provides an adapter rule that consumes any message on *Some* and issues a message on either *One* or *Many*, depending on the value of *n*.

History-dependent acceptable states. The class *gget_buff* alters *buff* as follows: the new method *gget* returns one item from the buffer (like *get*), except that a request on *gget* can be served only immediately after serving a request on *put*. More precisely, a *put* transition enables *gget*, while *get* and *gget* transitions disable it. This condition is reflected in the code by introducing two labels *AfterPut* and *NotAfterPut*. Then, messages on *gget* are synchronized with messages on *AfterPut*.

```
class  gget_buff = self (z)
    gget(r) & AfterPut() & (Full(A, i, n) or Some(A, i, n)) ▷
          r.reply(A[i]) & z.NotAfterPut()
     &   z.Check(A, (i+1) mod A.size, n−1)
or match buff with
    Init(size) ⇒ Init(size) ▷ z.NotAfterPut()
  | put(v, r) ⇒
      put(v, r) & (AfterPut() or NotAfterPut()) ▷ z.AfterPut()
  | get(r) ⇒
      get(r) & (AfterPut() or NotAfterPut()) ▷ z.NotAfterPut()
end
```

The first clause in the match construct refines initialization, which now also issues a message on *NotAfterPut*. The two other clauses refine the existing methods *put* and *get*, which now consume any message on *AfterPut* or *NotAfterPut* and produce a message on *AfterPut* or *NotAfterPut*, respectively.

Modification of acceptable states. We first define a general-purpose lock with the following *locker* class:

```
class  locker = self (z)
    suspend(r) & Free() ▷ r.reply() & z.Locked()
or resume(r) & Locked() ▷ r.reply() & z.Free()
```

This class can be used to create locks, but it can also be combined with some other class such as *buff* to temporarily prevent message processing in *buff*. To this end, a simple disjunction of *buff* and *locker* is not enough and some refinement of the parent class *buff* is required:

```
class  locked_buff = self (z)
    locker
or match buff with
      Init(size) ⇒ Init(size) ▷ z.Free()
    | O ⇒ Free() ▷ z.Free()
    end
```

The first clause in the match construct supplements the initialization of *buff* with an initial *Free* message for the lock. The second clause matches every other rule of *buff*, and requires that the refined clause consume and produce a message on *Free*. (The semantics of clause selection follows the textual priority scheme of ML pattern-matching, where a clause applies to all reaction rules that are not selected by previous clauses, and where the empty selection pattern acts as a default case.)

As a consequence of these changes, parent rules are blocked between a call to *suspend* and the next call to *resume*, and parent rules leave the state of the lock unchanged. In contrast with previous examples, the code above is quite general; it applies to any class following the same convention as *buff* for initialization.

5 Related and future works

There are many works on supplementing object calculi with concurrent primitives [16,3,11], or on supplementing process calculi with objects (usually by the mean of an encoding in the original calculus) [19,18,12,5]. Our work follows the latter tradition. However, to our knowledge, it is the only one to address safe object composition in a process calculus.

In [17], Odersky proposes an object-oriented extension of a language based on the join calculus. His proposal amounts to adding some record structure to join definitions, as we do in section 2. However, Odersky does not consider the problem of inheritance and refinement of synchronization. A small technical difference is that Odersky's calculus is designed with pattern matching on values: values in Odersky's calculus are not only object names but also constructed values (such as strings, integers, lists, etc.); then, the shape of values can be taken into account during synchronization. For instance, a rule of the form $\ell(h :: t) \triangleright P$ is allowed, and will only fire when a message on ℓ carries a list that contains at least one cell. The extension of our calculus to structured values is easy (see [9]). However, we believe that synchronization should only concern names, and not depend on the shape of values itself. In particular, this allows a simpler semantics and an efficient compilation of synchronization into automata [14].

Since our type system abstracts from the shape of synchronization patterns in classes, it is blind to a number of relevant properties of concurrency, such as the presence of race conditions or deadlock freedom. The design of a sophisticated analyzer that is sensitive to synchronizations is a promising research direction.

6 Conclusions

We have proposed a simple, object-based variant of the join calculus. Every object is defined as a fixed set of reaction rules that describe its synchronization behavior. The expressiveness of the language is significantly increased by adding classes—a form of open definitions that can be incrementally assembled before object instantiation. Thereby, we partially recover the ability of the pi calculus to dynamically define the receptive behavior for messages. Our layered design confines this capability to classes. From a programming-language viewpoint, this seems a good compromise between flexibility and simplicity of the model. Indeed, our proposal still enables efficient compilation of synchronization and type inference.

Acknowledgments. This work benefited from discussions with Luca Cardelli, Sylvain Conchon, Fabrice Le Fessant, and François Pottier. In particular, Sylvain Conchon and Fabrice Le Fessant contributed to discussions on the inheritance operators and the selective refinement. We also discussed the relation between generalization and synchonization with Sylvain Conchon and Franois Pottier.

References

1. G. Agha, P. Wegner, and A. Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
2. P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.
3. P. D. Blasio and K. Fisher. A calculus for concurrent objects. In U. Montanari and V. Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, LNCS 1119, pages 406–421, 1996.
4. L. Cardelli. Obliq A language with distributed scope. SRC Research Report 122, Digital Equipment, June 1994.
5. S. Dal-Zilio. Quiet and bouncing objects: Two migration abstractions in a simple distributed blue calculus. In H. Hüttel and U. Nestmann, editors, *Proceedings of the Worshop on Semantics of Objects as Proceedings (SOAP '98)*, pages 35–42, June 1998.
6. C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, Palaiseau, Nov. 1998.
7. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96*, pages 372–385, Jan. 1996.
8. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, LNCS 1119, pages 406–421, 1996.
9. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the 8th International Conference on Concurrency Theory*, LNCS 1243, pages 196–212, 1997.
10. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join-calculus. Full version. Available electronically at `http://cristal.inria.fr/~remy/work/ojoin/`, June 2000.
11. A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16(3) of *entcs*, Nice, France, Sept. 1998.
12. J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98)*. North-Holland, 1998.
13. F. Le Fessant. The JoCAML system prototype. Software and documentation available from `http://pauillac.inria.fr/jocaml`, 1998.
14. F. Le Fessant and L. Maranget. Compiling join-patterns. *Electronic Notes in Computer Science*, 16(2), 1998.
15. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 4, pages 107–150. The MIT Press, 1993.
16. O. Nierstrasz. Towards an object calculus. In O. N. M. Tokoro and P. Wegner, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 1–20, 1992.
17. M. Odersky. Functional nets. In *European Symposium on Programming 2000*, LNCS. Springer Verlag, 2000.

18. D. Sangiorgi. An interpretation of typed objects into typed π-calculus. *Information and Computation*, 143(1):34–73, 1998.
19. D. J. Walker. Objects in the pi-calculus. *Information and Computation*, 116(2):253–271, 1995.
20. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. *ACM SIGPLAN Notices*, 21(11):258–268, Nov. 1986. Proceedings of OOPSLA '86.