1

Inheritance in Concurrent Objects

Cosimo Laneve

Dipartimento di Scienze dell'Informazione, Bologna

Abstract

Inheritance is a crucial concept in object-oriented specification, design and programming, potentially enabling code reuse. This concept has been recently integrated with concurrency, due to the development of concurrent object-oriented programming.

This chapter reviews the inheritance principle initially in the sequential setting and then in the context of concurrent object calculi. We discuss, subsume into a unifying model, and systemize the proposals in the literature.

Keywords: Inheritance, concurrent object calculi, synchronization mechanisms.

1.1 Introduction

Inheritance is the basic mechanism of object-oriented programming to favour abstraction, incremental development and code reuse. In the object-oriented programming style, the code is usually structured in object patterns, called *classes*, that are defined out of basic definitions or pre-existing classes.

When object-oriented languages are integrated with concurrent primitives, the emphasis is usually laid on the modelling of the parallelism and of mutual exclusion (see, for instance, `POOL` [Ame89], `ABCL` [YBS86], `Obliq` [Car94] or `Java` [GJS96]). No special care is given to inheritance, that is mostly handled as in the sequential setting. As a result, one often stumbles into subtle typing flaws [Sar97] or into anomalies that compel to redefine the inherited code [MY93].

Some progress in devising a rigorous model bringing objects and concurrency together has been recently made. Several researchers have in fact defined basic languages for the concurrent object-oriented paradigm, equipping them with precise and clear semantics, simple type systems and equational theories. To date two approaches have emerged. One consists on recasting some well-established object-calculus and enriching it with concurrent primitives (parallel composition and the operation of scope restriction) [BF96, GH98]. The second attempts the other way around, trying to extend process calculi with objects by introducing record types [Vas94, DZ98].

However, there is an apparent distance among these calculi and concurrent object-oriented languages that jeopardizes their utility. The focus in the latters is on classes and inheritance of class definitions while the formers, being object-based, largely overlook these issues. Regrettable, forgetting about classes eludes any treatment of inheritance, which is even more crucial when concurrency is added.

In this chapter we define a concurrent object-based calculus, in the same style of [BF96, GH98], but simpler because it uses a name passing discipline. Then, on top of it, we introduce classes and inheritance. The full calculus is used to review and systemize the main developments in the design of concurrent object-oriented languages as concurrent primitives, inheritance and synchronization operations for inheritance.

Synchronization operations for inheritance allow to assemble concurrent objects and, at the same time, modify the internal synchronization between the methods. For instance, consider a two-cell buffer defined by inheriting the code of a one-cell one. In this case, reads and writes must be desynchronized to admit a further method emptying the cell and storing the value in another field. This amounts to redefine the whole code, if standard inheritance is used.

Two basic inheritance mechanisms to refine object's internal synchronizations are *enabled sets* (developed for `Rosette` [TS89]) and *method guards* (proposed for `Guide` [DKM+89]). The formers specify the set of methods that can be invoked along the life of an object: objects always accept invocations of methods in the active enabled set and refuse the others. Enabled sets may be suitablely refined along the inheritance, thus implicitly changing the object behaviour. Method guards are boolean expressions attached to each method that must be satisfied in order the method can execute. Also method guards may be refined during the inheritance, therefore altering the overall behaviour. We prove that two class-based languages, respectively with enabled sets and with method guards, can be encoded in a compositional way into our basic language when extended with an operation testing for object name equality. In particular we demonstrate that name disequality is not necessary to encode enabled sets.

Therefore, we may conclude that enabled sets and method guards do not seem to solve any anomaly about inheritance (of the synchronization code) that cannot be solved into a simple calculus without any synchronization mechanism. Perhaps, a rigorous analysis of the inheritance of the synchronization code could shed some light and suggest some new model. In this respect, the work [FLMR99] should contribute to this aim.

The rest of the chapter is organized as follows. In Section 1.2 we present the sequential object calculus, including the underlying model, the observational semantics and the class extension. In Section 1.3 we extend the basic calculus with primitives for concurrency and discuss other proposals in the literature. In Section 1.4 we introduce the problem of inheritance anomaly and discuss an example. In Section 1.5 we describe two languages, one with enabled sets and the other with method guards, and determine their expressive power. We conclude in Section 1.6.

1.2 The object calculus

In this section we introduce the core, sequential object calculus, detailing the underlying model and the observational semantics.

1.2.1 Syntax

We consider three countable disjoint sets of names: *object names* – ranged over by $o, o', x, \cdots$ – ; *method names* – ranged over by $m, n, \cdots$ – ; *field names* – ranged over by $a, b, \cdots$. Tuples of names are written ${u_i}^{i \in 1..n}$, or abbreviated into $\widetilde{u}$.

The syntax of the *object calculus* consists of *terms* p, *definitions* d and *expressions* e that are defined by:

$$
\begin{array}{rcl}
p & ::= & \mathsf{0} \\
  & \mid & o.m(\widetilde{e}) \\
  & \mid & o.a := e \,;\, p \\
  & \mid & \mathsf{obj}\ o = \varsigma(x)\,[a_i = {o_i}^{i \in 1..h}, m_i(\widetilde{x_i}) = {p_i}^{i \in 1..k}]\ \mathsf{in}\ p
\end{array}
\qquad
\begin{array}{rcl}
e & ::= & o \\
  & \mid & o.a
\end{array}
$$

In a term $\mathsf{obj}\ o = \varsigma(x)\, d\ \mathsf{in}\ p$, the *object names* o and x are bound and their scope are p and d, respectively. In a definition $[a_i = {o_i}^{i \in 1..h}, m_i(\widetilde{x_i}) = {p_i}^{i \in 1..k}]$, the object names $\widetilde{x_i}$ are bound and their scope is p_i. As usual, a name that is not bound is called *free* and we write $p\{^e/_x\}$ for the substitution of e for each free occurrence of x in p. Terms will be identified when equal up-to renaming of bound names. Finally, tailing $\mathsf{0}$ in a term p will be often omitted.

A term may be the inert program $\mathsf{0}$; or $o.m(\widetilde{e})$, the asynchronous invocation of the method m in the object o with arguments $\widetilde{e}$; or $o.a := e\,;\,p$, the updating of the field a in the object o with the (value of the) expression e and with continuation p; or the object definition $\mathsf{obj}\ o = \varsigma(x)\,[\sigma]\ \mathsf{in}\ p$, where x is the *internal name* of the object o, and $[\sigma]$ is a record of field and method definitions. Expressions have two shapes: they may be object names o or field selectors $o.a$.

As a simple example of object, consider the cell with the field *val* and methods *read* and *write*.

$$\begin{array}{ll} \mathsf{obj}\ cell = \varsigma(x) & [val = x, \\ & \ read(o) = o.back(x.val), \\ & \ write(o, v) = x.val := v\,;\,o.ack() \\ &]\ \mathsf{in}\ \cdots \end{array}$$

In this chapter we shall also use an extension of the object calculus with a *name comparison operator* $[x = y]\,p,\, p'$, whose intended meaning is that p or p' is performed whether $x = y$ or not. A weaker operator is *name-matching* that has no continuation in case of disequalities. We write the name-matching as a derived operator: $[x = y]\,p,\, \mathsf{0}$. Name comparisons will make easy the encoding of inheritance mechanisms in our basic calculus. See [NP96] for an analysis of the expressiveness of name comparison in process algebras.

1.2.2 Semantics

The operational semantics is defined as a *reflexive chemical abstract machine* – RCHAM, for short – in the style of [FG96]. The chemical rules are given in Table 1.1. They define disjoint sets of transitions $\rightharpoonup$ (*heatings*), $\leftharpoondown$ (*coolings*), $\longrightarrow$ (*reductions*) between solutions $\mathcal{O} \Vdash p$, where $\mathcal{O}$ is a set of named definitions and p is the term to evaluate. As is usual in chemical semantics, each rule mentions only the components that participate to the rewriting, and the rewriting applies to every chemical solution that contains them. Sequences of fields and methods $a_i = o_i{}^{i\in 1..h}, m_i(\widetilde{o_i}) = p_i{}^{i \in 1..k}$ are shortened into σ. Solutions will be ranged over by $\mathcal{S}, \mathcal{T}, \cdots$. In Table 1.1, the rules for name comparison have been given, too.

The meaning of the rules is as follows. Rules (NIL) and (OBJ) describe structural, reversible rearrangements in a solution. They serve to prepare terms for (proper) reductions. The rule (OBJ) describes the introduction of objects in a solution; the side-condition of the rule enforces a static scoping discipline by substituting a fresh name for the local variable when the object

(NIL)	$\Vdash 0$	$\rightleftharpoons$	$\Vdash$
(OBJ)	$\Vdash \mathsf{obj}\ o = \varsigma(x)\,[\sigma]\ \mathsf{in}\ p$	$\rightleftharpoons$	$o' = [\sigma]\{o'/x\} \Vdash p\{o'/o\}$
(MET)	$o = [m(\widetilde{x}) = p, \sigma] \Vdash\ o.m(\widetilde{o'})$	$\longrightarrow$	$o = [m(\widetilde{x}) = p, \sigma] \Vdash\ p\{\widetilde{o'}/\widetilde{x}\}$
(ASG)	$o = [a = o', \sigma] \Vdash\ o.a := o''\,;\, p$	$\longrightarrow$	$o = [a = o'', \sigma] \Vdash\ p$
(ARG)	$o = [a = o', \sigma] \Vdash\ o''.m(\widetilde{e_1}, o.a, \widetilde{e_2})$	$\longrightarrow$	$o = [a = o', \sigma] \Vdash\ o''.m(\widetilde{e_1}, o', \widetilde{e_2})$
(EXP)	$o' = [b = o'', \sigma'] \Vdash\ o.a := o'.b\,;\, p$	$\longrightarrow$	$o' = [b = o'', \sigma'] \Vdash\ o.a := o''\,;\, p$
(NMY)	$\Vdash\ [o = o]\, p,\, p'$	$\longrightarrow$	$\Vdash p$
(NMN)	$\Vdash\ [o = o']\, p,\, p'$	$\longrightarrow$	$\Vdash p'$

With side condition:

(OBJ) o' is a fresh name.

(NMN) o and o' are two different names.

Table 1.1. *The reflexive cham of the object calculus*

is introduced, and checking that the object name does not appear elsewhere in the solution when folding it back as a local object definition. Moves $\rightleftharpoons$ are called *structural moves*.

Rule (MET) models method invocation: it happens when all the arguments have been evaluated into object names and the effect is to trigger the method body where the formal arguments have been replaced by the actual ones. The rule (ASG) defines field updating. Observe that the continuation p is evaluated with the updated object. The rules (ARG) and (EXP) model expression evaluation; according to (ARG), arguments of method invocations may be evaluated in any order.

The last two rules describe name-comparison: (NMY) and (NMN) define the behaviour of the process $[o = o']\, p,\, p'$ when the boolean guard is true and false, respectively.

Our object calculus is different from most of the calculi in the literature. Both the calculus in [FHM93] and the ς-calculus in Chapter 6 of [AC96] are functional; the former is also higher-order (expressions may be complex objects) and has "extensible" objects, meaning that attributes may be lenghtened at wish. The closest relative of the object calculus in this section is the untyped imperative calculus **imp**ς in Chapter 10 of [AC96]. However **imp**ς is somehow more involved: the updating operation may not terminate and the (big-step) semantics is also not so simple. Still we have defined an encoding of **imp**ς into our object calculus, though we omit it here. It relies on defining suitable objects for every method body, and storing the names of these objects in suitable fields of the original object. Then method calls reduce to dereferentiations and method update just relies on updating the field related to the method. We conjecture that this encoding is adequate with respect to the barbed congruence defined below.

1.2.3 Barbed congruence

The extensional semantics of the object calculus is *barbed congruence*. We recall its definition below, and refer to [MS92, GR96] for discussion.

Some notation is needed to formally describe barbed congruence. A *context* $C[\,\cdot\,]$ is a term with a placeholder $[\,\cdot\,]$; $C[p]$ means the term obtained by replacing the placeholder with p. Sometimes we use simpler contexts, called *object contexts*, where the hole may be replaced by object names. Object contexts are defined as follows:

$$C[\,\cdot\,] \;\;::=\;\; [\,\cdot\,].a := o\,;\, p \;\mid\; o.a := [\,\cdot\,].b\,;\, p \;\mid\; [\,\cdot\,].m(\widetilde{e}) \;\mid\; o.m(\widetilde{e}, [\,\cdot\,].a, \widetilde{e'})$$

The *output barb* $\Downarrow_o$ is a predicate over RCHAMs that tests for the potential access to a free object name o:

$$\mathcal{S} \Downarrow_o \;\stackrel{\triangle}{=}\; \mathcal{S} \rightleftharpoons^* \mathcal{O} \Vdash C[o] \text{ where } C[\,\cdot\,] \text{ is an object context and } o \text{ is free.}$$

Definition 1.2.1 *A relation $\mathcal{R}$ on open solutions is a* barbed simulation *if, whenever $\mathcal{S}\mathcal{R}\mathcal{T}$, we have*

- *if $\mathcal{S} \Downarrow_o$ then $\mathcal{T} \Downarrow_o$;*
- *if $\mathcal{S} \rightleftharpoons^* \longrightarrow \rightleftharpoons^* \mathcal{S}'$ then $\mathcal{T} \rightleftharpoons^* \mathcal{T}'$ and $\mathcal{S}'\mathcal{R}\mathcal{T}'$.*

A relation $\mathcal{R}$ is a barbed bisimulation *if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are barbed simulations.*

A barbed congruence *is a barbed bisimulation that is closed by application of contexts. We let $\approx$ be the largest barbed congruence.*

Barbed bisimulation is not a congruence. For instance, $o.m()$ and $o.n()$ are barbed bisimilar, but $o.m() \not\approx o.n()$ because the context $\mathsf{obj}\ o = [m() = o'.a(),\ n() = o''.b()]\ \mathsf{in}\ [\,\cdot\,]$ separates them. Also barbed congruence is unsensitive to divergence, i.e. $\mathsf{0} \approx \mathsf{obj}\ o = [m() = o.m()]\ \mathsf{in}\ o.m()$.

Barbed congruence coincides with (may) testing congruence because the RCHAM of the object calculus is determinate (there is always at most one successor solution). These two semantics differ when the model is nondeterministic, that will be the case when concurrency comes into the scene.

We are committing to barbed congruence because it is easier to check than testing semantics.

1.2.4 Classes and Inheritance

The object calculus misses a crucial feature of object-oriented programming: the *possibility of re-using the code*. For instance, to specify two objects with the same definitions, the entire code has to be copied. This problem is usually solved by promoting object templates, called *classes*, and adding class instantiations to object definitions.

It is often the case that, in re-using a definition, one needs to perform some smooth adaptations. To deal with these situations, languages also offer mechanisms for expanding, modifing or combining class definitions. In this context, a class that is defined out of another one is called a *subclass* of the latter, or, alternatively, it *inherits* the code of the latter. For instance, in `Smalltalk` a subclass can add additional methods to a class or replace existing methods [GR83]; and in general there is no relation between methods in a class and those in a subclass (attributes of the parent class may be accessed from the subclass with the directive `super`). To avoid this anomaly, `Beta` forces inherited definitions to have precedence over extensions [KMMPN87]. See [BC90] for a detailed comparison between the two forms of single inheritance.

We illustrate classes and inheritance by extending our basic calculus to cover the new features. Two new countable sets of names are introduced: *class names*, ranged over by $c, c', \cdots$, and *super names*, ranged over by $s, s', \cdots$. The grammar for terms and definitions is extended as follows.

$$
\begin{array}{rcl}
p & ::= & \ldots \\
  & | & \mathsf{obj}\ o = c\ \mathsf{in}\ p \\
  & | & \mathsf{class}\ c = \varsigma(x)\, d\ \mathsf{in}\ p
\end{array}
\qquad
\begin{array}{rcl}
d & ::= & [a_i = o_i{}^{i \in 1..h}, m_i(\widetilde{o_i}) = p_i{}^{i \in 1..k}] \\
  & | & c \\
  & | & \varsigma(x)\ d \\
  & | & [a_i = o_i{}^{i \in 1..h}, m_i(\widetilde{o_i}) = p_i{}^{i \in 1..k}]\ \mathsf{extends}\ d \\
  & | & d\ \mathsf{hide}\ m\ \mathsf{as}\ s
\end{array}
$$

The class constructor $\mathsf{class}\ c = \varsigma(x)\, d\ \mathsf{in}\ p$ defines a class name c whose scope is the term p; d is out of the scope of c – recursive class definitions are banned out. The scope of the self object name x is the definition d. Definitions now have a broad range of alternatives. They may be simple definitions, as before, or may inherit definitions of another class c. The operation $\varsigma(x)\ d$ abstracts the self name of the class. The extension and the possible replacement of the attributes of a definition d is managed by $[a_i = o_i{}^{i \in 1..h}, m_i(\widetilde{o_i}) = p_i{}^{i \in 1..k}]$ $\mathsf{extends}$ d; the intended meaning of this

For definitions: $\overset{x}{\leadsto}$

$$\varsigma(z)\ d \overset{x}{\leadsto} d\{x/z\}$$

$$[\sigma]\ \mathsf{extends}\ [\sigma'] \overset{x}{\leadsto} [\sigma/\sigma']$$

$$[a_i = o_i{}^{i\in 1..h}, m_i(\widetilde{o_i}) = p_i{}^{i\in 1..k}]\ \mathsf{hide}\ m\ \mathsf{as}\ s \overset{x}{\leadsto} [a_i = o_i{}^{i\in 1..h}, m_i\{s/m\}(\widetilde{o_i}) = p_i{}^{i\in 1..k}]$$

$$\frac{d \overset{x}{\leadsto} d'}{[\sigma]\ \mathsf{extends}\ d \overset{x}{\leadsto} [\sigma]\ \mathsf{extends}\ d'} \qquad \frac{d \overset{x}{\leadsto} d'}{d\ \mathsf{hide}\ m\ \mathsf{as}\ s \overset{x}{\leadsto} d'\ \mathsf{hide}\ m\ \mathsf{as}\ s}$$

For terms: $\leadsto$

$$\mathsf{class}\ c = \varsigma(x)\,[\sigma]\ \mathsf{in}\ p \leadsto p\{\varsigma(x)\,[\sigma]/c\} \qquad \frac{d \overset{x}{\leadsto} d'}{\mathsf{class}\ c = \varsigma(x)\,d\ \mathsf{in}\ p \leadsto \mathsf{class}\ c = \varsigma(x)\,d'\ \mathsf{in}\ p}$$

+ context closure

Table 1.2. *The compilation of class expressions into object terms*

operation is the one of `Smalltalk`; however `Beta` enthusiasts may recover the other form of inheritance by changing the replacement operation σ'/σ to be discussed sooner. The last operation d hide m as r is not usual in object-oriented languages and comes from the objective join calculus [FLMR99]. This operation replaces the method definition $m(\widetilde{x}) = p$ occurring in d by $s(\widetilde{x}) = p$ and it gives to subclasses the way for invoking methods of the superclass.

The semantics of classes and inheritance is given by compiling class expressions into terms of the basic object calculus. This definitely strenghtens the fact that these features are just syntactic sugar for organizing and combining code; no expressivity is added to the core calculus. To formalize this compilation, we introduce the notation σ/σ'. Let σ be $a_i = o_i{}^{i\in 1..h}, m_i(\widetilde{o_i}) = p_i{}^{i\in 1..k}$ and σ' be $a_i = o_i'{}^{i\in h'..h}, a_i = o_i{}^{i\in h+1..h''}, m_i(\widetilde{o_i}) = q_i{}^{i\in k'..k}, m_i(\widetilde{o_i}) = p_i{}^{i\in k+1..k''}$; then σ/σ' abbreviates the tuple $a_i = o_i{}^{i\in 1..h''}, m_i(\widetilde{o_i}) = p_i{}^{i\in 1..k''}$.

Table 1.2 defines the compilation step. There are two reductions: $\overset{x}{\leadsto}$ for definitions and $\leadsto$ for terms; and context rules for the latter one are omitted. We discuss the rule $[a_i = o_i{}^{i\in 1..h}, m_i(\widetilde{o_i}) = p_i{}^{i\in 1..k}]$ hide m as $s \overset{x}{\leadsto} [a_i = o_i{}^{i\in 1..h}, m_i\{s/m\}(\widetilde{o_i}) = p_i{}^{i\in 1..k}]$ since the others are fairly intelligible. This rewriting rule implements the so-called *late binding* mechanism because renaming applies to the method heading only: every invocation to m in the body will refer to the last definition of m in the inheritance hierarchy.

Therefore it implicitly assumes that the renamed method is also redefined in the subclass. The untyped calculus has no mechanism to compel this redefinition; rather a typing system, in the style of [FLMR97] could solve this flaw.

Example 1.2.2 Let us put at work our compiler with a standard example of class and subclass: cells and logged cells.

$$
\begin{array}{ll}
\mathsf{class}\ \mathit{cell} = \varsigma(x) & [\mathit{val} = x,\\
& \ \mathit{read}(o) = o.\mathit{back}(x.\mathit{val}),\\
& \ \mathit{write}(o,v) = x.\mathit{val} := v\,;\, o.\mathit{ack}(),\\
&]\\
\ \mathsf{in\ class}\ \mathit{logcell} = & \varsigma(x)\,[\mathit{write}(o,v) =\\
& \qquad\qquad \mathsf{obj}\ o' = [\mathit{ack}() = x.\mathit{log}(o,v)]\ \mathsf{in}\ x.\mathit{Write}(o',v),\\
& \qquad \mathit{log}(o,v) = \mathtt{print}(o, \text{“}\mathit{writes}\text{”}, v)\,;\, o.\mathit{ack}()\\
& \qquad]\ \mathsf{extends}\ (\mathit{cell}\ \mathsf{hide}\ \mathit{write}\ \mathsf{as}\ \mathit{Write})
\end{array}
$$

where `print` is a directive (system call) added to the basic calculus. A logged cell improves (i.e. overrides) the *write* operation in order to keep track of updates. In particular, for each update, the new *write* also calls the method *log* that prints who is updating and the new value. According to Table 1.2, the above definition of *logcell* expands as follows:

$$
\begin{array}{lll}
\mathsf{class}\ \mathit{logcell} = & & \\
& \varsigma(x) & [\mathit{val} = x,\\
& & \ \mathit{read}(o) = o.\mathit{back}(x.\mathit{val}),\\
& & \ \mathit{Write}(o,v) = x.\mathit{val} := v\,;\, o.\mathit{ack}()\\
& & \ \mathit{write}(o,v) = \mathsf{obj}\ o' = [\mathit{ack}() = x.\mathit{log}(o,v)]\ \mathsf{in}\ x.\mathit{Write}(o',v),\\
& & \ \mathit{log}(o,v) = \mathtt{print}(o, \text{“}\mathit{writes}\text{”}, v)\,;\, o.\mathit{ack}()\\
& &]
\end{array}
$$

The compilation steps in Table 1.2 do terminate and give a unique result (are determinate). The crucial statements entailing these properties are collected in the following lemma.

Lemma 1.2.3

(i) *Reductions* $\overset{x}{\leadsto}$ *and* $\leadsto$ *are confluent.*

(ii) *Both* $\overset{x}{\leadsto}$ *and* $\leadsto$ *do terminate.*

Multiple inheritance. Several programming languages as `C++`, `Eiffel`, `CLOS`, `Self` allow the definition of classes with more ancestors, called *multiple inheritance*. In these cases, the problematic issue becomes the resolution of ambiguities of attributes conjointly defined by several classes. In C++ these ambiguities are ruled out by overriding in the subclass the methods with multiple definitions. The mechanism of multiple inheritance is still richer in CLOS. In this language ambiguities are solved by *linearizing* the ancestor graph of a class to produce an inheritance list (see [Sny87] for a criticism of this process). In Self linearization is furtherly weakened yielding *partial orders* of ancestors and solving ambiguities with error messages [CUCH91].

To introduce multiple inheritance in our framework we extend definitions with

$$s \ ::= \ \ldots \\ \quad\quad\ \mid \ \widetilde{d}$$

meaning that s inherits all the attributes in every class in $\widetilde{c}$. Ambiguities may be solved by suitable rewriting rules for definitions. For instance a rule solving multiple inheritance as in CLOS could be the following:

$$[\sigma_1], [\sigma_2], \cdots, [\sigma_n] \overset{x}{\leadsto} [\sigma_1] \ \mathsf{extends} \ ([\sigma_2] \ \mathsf{extends} \ \cdots [\sigma_n]) \cdots)$$

where classes in the left override those in the right. It is not the case to detail further this topic in the present paper; the reader is referred to the literature. Multiple inheritance plays a prominent rule in concurrency; a thourough discussion can be found in [FLMR99].

1.2.5 Typing issues

There is a broad literature about types of object-oriented languages, mainly covering the relationships between subtyping polymorphism and other features such as recursive definitions, width and depth subtyping, binary methods, existential types, etc. (see [AC96] and the references therein).

The extended object calculus of Section 1.2.4 may be equipped with a polymorphic type system *à la* ML that can ban *message-not-understood errors* as well as some encapsulation violations (fields and super methods can be accessed by methods in the same object). We overlook this issue in the present paper; the reader can find in [FLMR97] an analogous typing system.

1.3 Adding concurrency: threads and locks

In this section, our basic object calculus is extended with concurrent primitives. To this end, we commit to choices that have a simple formal foundation; the other proposals are briefly discussed later on.

The presence of concurrent computations may compromise the encapsulation property of objects because they may invalidate the consistency of object's internal state. To avoid this flaw, most of the proposals in the literature rely on locks and primitives for acquiring and releasing locks (*mutexes*). We give a slight general solution than [GH98], by introducing *lock fields* and primitives for acquiring and releasing locks (in [GH98], every object has exactly one lock; here this constraint is relaxed). Actually, since the concurrent object calculus can express atomic reads and writes on a shared memory, the mutexes are not strictly needed because we could use a standard mutual exclusion algorithm for synchronizations. However this should lead to complex programs that will be difficult to read. It is better to have mutexes as primitive operations and study the internal encoding of the calculus with mutexes into the one without them.

Let *locks* be a new countable set of names, ranged over by ℓ, ℓ'. Locks get two primitive object values: tt and ff. We use the notation "-" to mean one of the two values tt or ff. The syntax of terms p, from now on called *processes*, is extended as follows:

$$
\begin{array}{rcl}
p & ::= & \cdots \\
  &   & \mid \; p \,|\, p \\
  &   & \mid \; \mathit{acquire}(o.\ell) \,;\, p \\
  &   & \mid \; \mathit{release}(o.\ell)
\end{array}
$$

This extension encompasses parallel composition of processes and two new directives $\mathit{acquire}(o.\ell)$ and $\mathit{release}(o.\ell)$, respectively for setting and releasing the locks. The intended meaning of $p \,|\, q$ is that the operations of p and q may be performed in every order; $\mathit{acquire}(o.\ell) \,;\, p$ may be performed provided the lock ℓ of the object o is tt (so it is blocking), then it sets this lock to ff and triggers the continuation p; $\mathit{release}(o.\ell)$ sets the lock ℓ of the object o to tt, regardless of its value, it is not blocking.

Table 1.3 defines the RCHAM for the object calculus with threads and locks. Remark that now solutions are $\mathcal{O} \Vdash \mathcal{P}$, where $\mathcal{P}$ is a *multiset* of processes. Notice also that the test and set operations in (ACQ) are atomic.

Example 1.3.1 A standard example of a concurrent class is the *asynchronous channel*. It consists of a value field and two methods for reading and writing. In order to sequentialize reads and writes, two locks are added, ℓ_w and ℓ_r, that respectively enable read and write when set to tt. At the

Rules of Table 1.1 +

$$
\begin{array}{lrcl}
(\textsc{Par}) & \Vdash p \,|\, q & \rightleftharpoons & \Vdash p,\, q \\[4pt]
(\textsc{Acq}) & o = [\ell = tt, \sigma] \Vdash \ acquire(o.\ell)\,;\, p & \longrightarrow & o = [\ell = ff, \sigma] \Vdash \ p \\
(\textsc{Rel}) & o = [\ell = -, \sigma] \Vdash \ release(o.\ell) & \longrightarrow & o = [\ell = tt, \sigma] \Vdash
\end{array}
$$

Table 1.3. *The reflexive cham of the object calculus with threads and locks*

beginning ℓ_w is set to tt and ℓ_r is set to ff.

$$
\begin{array}{l}
\mathsf{class}\ ch = \\
\quad \varsigma(x)\ \ [v = x,\ \ell_w = tt,\ \ell_r = ff, \\
\qquad\quad read(o) = acquire(x.\ell_r)\,;\, \mathsf{obj}\ o' = [v = o']\ \mathsf{in} \\
\qquad\qquad\qquad\qquad o'.v := x.v\,;\, (o.back(o'.v) \,|\, release(x.\ell_w))\,, \\
\qquad\quad write(o, o') = acquire(x.\ell_w)\,;\, x.v := o'\,;\, (o.ack() \,|\, release(x.\ell_r)) \\
\qquad\ ]\ \mathsf{in}\ \cdots
\end{array}
$$

Note that *read* sends back to the caller the field of a local object, rather than the value of the channel. Indeed a body as $acquire(x.\ell_r)\,;\,(o.back(x.v) \,|\, release(x.\ell_w))$ is wrong because the value returned by $o.back(x.v)$ is in general different from the value of the channel when $release(x.\ell_w)$ is performed.

Given the definition of asynchronous channels, it is straightforward to translate process calculi – like asynchronous π-calculus [ACS98] – in our concurrent object calculus. This highlights the expressiveness of the concurrent object calculus as far as concurrency is concerned. We shall return on this issue in Section 1.5.

1.3.1 Reentrant mutexes and self-serializations

Cardelli in [Car94] separates between *external* and *self-inflicted* operations. The formers are those operations a method performs on objects other than the method host object; the latters are operations a method performs on its own object (*self*). In our calculus self-inflicted calls may deadlock when the caller does not reset the state before the call.

Several object-oriented programming languages avoid these misbehaviours giving rights to threads instead of methods (see Java [GJS96], for example). In this refined variant of mutexes, called *reentrant mutexes*, a thread does not deadlock if it acquires a lock already locked by itself. To model correctly reentrant mutexes, one should keep track of the acquired locks and, when a thread forks, of the lock ownerships of the parent thread. This clearly makes the semantics more difficult. Still the solution is also questionable because it is somehow too liberal and too restrictive at the same time. It is liberal because external calls calling back methods of the original object do deadlock. It is restrictive because the notion of thread is local to an address space, whilst weaker notions are needed when features as process migration are added to the language.

An intermediate proposal between mutexes and reentrant mutexes is the so-called *self-serialization* (see [Car94] and [BF96]). With self-serialization a method can invoke an external operation provided the target object is "idle" (no other thread is performing operations on that object). On the other hand, self-inflicted operations can be invoked without deadlocking. Therefore the property whether an operation is self-inflicted or not is crucial. This property cannot be set at compile time; hence it is necessary to keep track at runtime of the current host object. Besides, this proposal is also exposed to criticisms because deadloks can occur when an external method invokes back a method of the original object.

Instead of discussing some tricky solution, we have committed to simplicity in this paper (i.e. simple mutexes), leaving to programmers the complete responsability of avoiding deadlocks in the design of complex algorithms. A detailed analysis of self-serialization can be found in [BF96]. Up-to our knowledge there is no formal study on reentrant mutexes.

1.4 The inheritance anomaly

In the last two decades several people has stumbled over oddities when inheriting concurrent code. America was the first to observe that there is no practical advantage in inheriting concurrent code because large redefinitions were usually needed to avoid synchronization flaws [Ame89]. Since then, a number of proposals to reduce this problem have appeared in the literature; in [MY93] the reader can find a survey and a comparison of most of the techniques, together with a name for this phenomenon: *inheritance anomaly*.

We introduce the problem with an example; inheritance anomaly is discussed soon after.

Example 1.4.1 Let us define a two place buffer out of the asynchronous channel in Example 1.3.1. Therefore, we extend the asynchronous channel with a new field, called u, that will store the second message. Moreover, we add a new method, named *daemon*, that will fill the field u when v is full and, at the same time, empty v. The method *daemon* also allows to parallelize reads and writes. So, as a first approximation, the class c could be extended by the definition:

$$\varsigma(x)\,[u = x,\ daemon() = x.u := x.v]\ \textsf{extends}\ c$$

The body of *daemon* is cleary broken because it must synchronize with *read* and *write*. This amounts to introduce two new locks ℓ_d and ℓ'_d, and change *read* and *write*:

$$
\begin{array}{lll}
\textsf{class}\ two\text{-}ch & = & \\
\qquad \varsigma(x) & [\ell_d = \textit{ff},\ \ell'_d = \textit{ff} & \\
 & \quad read(o) = acquire(x.\ell_r)\,;\ \textsf{obj}\ o' = [v = o']\ \textsf{in} & \\
 & \qquad\qquad\qquad o'.v := x.u\,;\,(o.back(o'.v) \mid release(x.\ell_d))\,, & \\
 & \quad write(o,o') = acquire(x.\ell_w)\,;\ x.v := o'\,;\,(o.ack() \mid release(x.\ell'_d)) & \\
 & \quad daemon() = acquire(x.\ell_d)\,;\ acquire(x.\ell'_d)\,;\ x.v := x.u\,; & \\
 & \qquad\qquad\qquad (release(x.\ell_r) \mid release(x.\ell_w) \mid x.daemon()\,) & \\
 &]\ \textsf{extends}\ ch\ \textsf{in}\ \cdots &
\end{array}
$$

We assume that *daemon* is invoked along the class instantiation. (There is a bit more conservative solution where only the method *read* is redefined along the inheritance and the field ℓ_r plays the same role of ℓ'_d.)

The inheritance anomaly is now evident because only fields ℓ_w and ℓ_r are inherited from *ch*, thus making the benefits of inheritance vanished in this case. Actually this is a controversial issue: one can argue that perhaps there is some basic concept behind asynchronous channels and two place buffers such that inheriting from it may be more useful; or it is possible to envisage some better reuse of asynchronous channels in Example 1.4.1. It is difficult to reply to these remarks. Nevertheless notice that the naive solution seems unpractical in this case and many other ones. This has to be considered as a noisy allarm-bell warning for a serious analysis of the problem.

What hinders code reuse in Example 1.4.1 is that we must weaken the synchronization between methods *read* and *write*. This amounts to deeply understand the code of these two methods and to completely rewrite it in order to match with the new requirements. Actually, even a smooth change of a method may have effects on every method interacting with it, and so on with the methods interacting with these latter ones. It is not surprising that such a propagation of changes may usually concern all the methods of the class. For this reason *inheritance of the control*, or of the *synchronization code*, of a class is a critical issue in concurrent object-oriented languages. On the contrary, inheritance anomaly is harmless in sequential languages because methods have no competing behaviour there (i.e. no need for synchronizing code).

It is beyond the scope of this paper to do a detailed analysis of inheritance anomaly; rather we point to references [MY93, Ber94, McH94]. In the next section we focus on two basic mechanisms for inheriting the control and compare their expressive power.

1.5 Proposals for inheriting the control

Several proposals to alleviate the inheritance anomaly have been studied in the literature. A thorough description of every mechanism should require too much space and should even be boring because many of them are somehow naive, without a rigorous motivation for the design choices. In this section we focus on two mechanisms that, we retain, underlie most of the proposals:

(i) *enabled sets* by Tomlinson and Singh [TS89],
(ii) *method guards* by Decouchant *et al.* [DKM+89],

In particular, *path expressions* refine enabled sets by adding regular expressions [CH73]; the `Maude` system [Mes93] or the *negative method guards* [Frø92] propose more elaborate method guards; the *synchronization schemas* [MY93] combine different synchronization mechanisms based on enabled sets and method guards.

We formally define the two models supporting enabled sets and method guards, then we systemize the expressive power of these mechanisms by proving the existence of a translation into our concurrent object calculus when it is extended with *name comparison*. Remarkablely, we provide two compositional encodings of the class-based languages with the operations for enabled sets and method guards into our language with classes. As a consequence, every solution to the problem of inheritance anomaly in the former two models may be immediately rephrased in our simple class language at the cost of using name comparisons operators. This confirms, at least for enabled sets and method guards, what emerges from McHale's PhD Thesis (see Chapter 13 of [McH94]), namely that all the proposals for inheriting the control vary in the expressive power they posses.

1.5.1 Enabled sets

An *enabled set* $\rho : \widetilde{m}$ is a labelled set of method names of an object; it specifies the set of methods $\widetilde{m}$ that can be invoked along the life of an object when ρ is "active". Enabled sets are activated by the special operation $\mathsf{become}(o.\rho)$, where o is the object to whom the enabled set belongs. Locks and the corresponding directives for acquiring and releasing them are now removed: all the synchronizations are delegated to enabled sets. The methods of an object are performed in a mutual exclusive way.

Enabled sets have been developed in the concurrent object oriented language `Rosette` [TS89]. To illustrate the programming practice with them, we rewrite the asynchronous channel of Example 1.3.1 into a `Rosette`-like language. We are informal at this stage: any doubt will be solved afterwards.

$$
\begin{array}{lll}
\mathsf{class}\ ch = \varsigma(x) & [\,v = x, & \\
& \quad read(o) = \mathsf{obj}\ o' = \varsigma(y)\,[v = y]\ \mathsf{in} & \\
& \quad\qquad\qquad o'.v := x.v\,;\,(o.back(o'.v) \mid \mathsf{become}(x.\mathrm{empty}))\,, & \\
& \quad write(o, o') = x.v := o'\,;\,(o.ack() \mid \mathsf{become}(x.\mathrm{full})) & \\
&],\ ([\mathrm{empty} = \{write\},\ \mathrm{full} = \{read\}) &
\end{array}
$$

When an object of class ch begins in a state where "empty" is enabled then the method *write* can execute. The termination of *write* activates the enabled set "full", thus enabling the execution of a *read* operation. With respect to Example 1.3.1, there is no lock and no primitive operation *acquire* and *release*. On the other hand, now, when a class is instantiated, one has to specify the active enabled set.

We split the analysis of the expressivity of enabled sets in two parts: we first focus on a basic object calculus of enabled sets and we present the encoding into the calculus of Section 1.3. Then we consider classes and inheritance operations over enabled sets.

Table 1.4 collects the syntax and the semantics of the object calculus with enabled sets. Objects in the left hand side of solutions have two shapes $o =_{\widetilde{m}} [\sigma], r$ and $o = [\sigma], r$. The former relates an object name to a definition and a collection of enabled set, and also specifies the set of methods $\widetilde{m}$ that can be invoked. The latter describes the state of the object where no method can be invoked. When a method invocation does succeed – see rule (Met) – the state of the object is turned into $o = [\sigma], r$, thus guaranteeing mutual exclusion. A method may be called again when the directive become is performed.

The encoding of objects. To translate the object calculus with enabled set into the concurrent object calculus of Section 1.3, we systematically add a method *ctr* to every object. The method *ctr* forces method sequentialization and constrains method executions to fit with what defined by the directive become and by the enabled sets. In this setting, every method call is turned

The Syntax:

$$
\begin{array}{rcl}
p & ::= & \mathsf{0} \\
& \mid & o.m(\widetilde{e}) \\
& \mid & o.a := e\,;\,p \\
& \mid & \mathsf{obj}\ o = \varsigma(x)\,[a_i = {o_i}^{i\in 1..h}, m_i(\widetilde{x_i}) = {p_i}^{i\in 1..k}], r\ \mathsf{in}\ p \\
& \mid & p \,|\, p \\
& \mid & \mathsf{become}(o.\rho)
\end{array}
\qquad
\begin{array}{rcl}
e & ::= & o \\
& \mid & o.a \\
& & \\
r & ::= & \rho : \widetilde{m} \\
& \mid & r\,,\,r
\end{array}
$$

The Semantics: Rules (Nil), (Asg), (Arg), (Exp) of Table 1.1 + (Par) of Table 1.3 +

$$
\begin{array}{llcl}
(\text{Obj}) & \Vdash \mathsf{obj}\ o = \varsigma(x)\,[\sigma], r\ \mathsf{in}\ p & \rightleftharpoons & o' = [\sigma\{{}^{o'}/_{x}\}], r \Vdash p\{{}^{o'}/_{o}\} \\
(\text{Met}) & o =_{m',\widetilde{m}} [m'(\widetilde{x}) = p, \sigma], r \Vdash o.m'(\widetilde{o'}) & \longrightarrow & o = [m'(\widetilde{x}) = p, \sigma], r \Vdash p\{{}^{\widetilde{o'}}/_{\widetilde{x}}\} \\
(\text{Bec}) & o = [\sigma], (\rho : \widetilde{m}, r) \Vdash \mathsf{become}(o.\rho) & \longrightarrow & o =_{\widetilde{m}} [\sigma], (\rho : \widetilde{m}, r) \Vdash
\end{array}
$$

With side condition:

(Obj) o' is a fresh name.

Table 1.4. *The concurrent object calculus with enabled sets*

into an invocation of *ctr* to check whether the object can accept such a call or not. This is possible provided that the method name belongs to the current enabled set. Therefore *ctr* must use or implement the *datatype set* in order to accomplish its task.

Sets are implemented into the basic calculus with the *name-matching* operator $[o = o']\,p, \mathsf{0}$. Less expressive primitives could probably be used at the price of complicating the encoding (such as comparisons between booleans or naturals, rather than names).

To simplify the encoding we abbreviate $\mathsf{obj}\ o_1 = \varsigma(x_1)\,[\sigma_1]\ \mathsf{in}\ \cdots \mathsf{obj}\ o_n = \varsigma(x_n)\,[\sigma_n]\ \mathsf{in}\ p$ with $\mathsf{obj}\ o_i = \varsigma(x_i)\,[\sigma_i]^{i\in 1..n}\ \mathsf{in}\ p$.

Let $\rho : S$, where $S \subseteq \{m_1, \cdots, m_k\}$. We define:

$$
\mathsf{P}_\rho(x, o, o_1, \cdots, o_k) \;\triangleq\; \textstyle\prod_{j\in 1..k} \mathsf{M}_j(x, o, o_1, \cdots, o_k)
$$

$$
\mathsf{M}_j(x, o, o_1, \cdots, o_k) \;\triangleq\; \begin{cases} [x = o_j](o.a_c := o\,;\, \mathit{release}(o.\ell_{m_j})), \mathsf{0} & \text{if } m_j \in S \\ [x = o_j](o.\mathit{ctr}(x, o.a_c) \,|\, \mathit{release}(o.\ell)), \mathsf{0} & \text{otherwise} \end{cases}
$$

The translation of the object calculus with enabled sets relies on two functions $[\![\,\cdot\,]\!]^o_{CTR}$ and $[\![\,\cdot\,]\!]$. The first one takes a sequence of fields, method definitions and enabled sets and uses a fresh object name o and a fresh super name CTR; its result is a sequence of fields and method definitions. The latter applies to terms and gives terms. The definition of $[\![p]\!]$ is by structural induction (we omit those cases where $[\![\cdot]\!]$ is homomorphic with respect to the syntactic operator):

$$[\![a_i = o_i{}^{i\in 1..h},\ m_i(\widetilde{x_i}) = p_i{}^{i\in 1..k},\ \rho_i : \widetilde{m_i}{}^{i\in 1..j}]\!]^o_{CTR} \stackrel{\triangle}{=}$$

$$\begin{array}{l} a_i = o_i{}^{i\in 1..h},\ a_{m_i} = o_{m_i}{}^{i\in 1..k},\ a_c = o, \\ \ell_{m_i} = \mathit{ff}^{i\in 1..k},\ \ell = \mathit{tt}, \\ m_i(\widetilde{x_i}) = (acquire(o.\ell_{m_i})\,;\,[\![p_i]\!])\,|\,(acquire(o.\ell)\,;\,o.ctr(o_{m_i}, o.a_c)^{i\in 1..k}), \\ ctr(x_{mth}, x_c) = \prod_{i\in 1..j}[x_c = o_{\rho_i}]\mathsf{P}_{\rho_i}(x_{mth}, o, o_{m_1}, \cdots, o_{m_k}),\ \mathsf{0} \\ \qquad\qquad\qquad\ |\ o.CTR(x_{mth}, x_c) \end{array}$$

$$[\![\mathsf{obj}\ o' = \varsigma(o)\,[\sigma],\ r\ \mathsf{in}\ p]\!] \stackrel{\triangle}{=} \mathsf{obj}\ o_\rho = \varsigma(x)\,[\,]^{\rho=\widetilde{m}\in r},\ o_m = \varsigma(x)\,[\,]^{m(\widetilde{x})=p\in\sigma}\ \mathsf{in}$$
$$\mathsf{obj}\ o' = \varsigma(o)\,[\,([\![\sigma,\ r]\!]^o_{CTR})/(\,CTR(x,y) = \mathsf{0})\,]\ \mathsf{in}\ [\![p]\!]$$

$$[\![\mathsf{become}(o.\rho)]\!] \stackrel{\triangle}{=} o.a_c := o_\rho\,;\,release(o.\ell)$$

$$[\![o.m(\widetilde{e})]\!] \stackrel{\triangle}{=} o.m(\widetilde{e})\,|\,o.ctr(o.a_m, o.a_c)$$

Foremost observe that labels of enabled sets are encoded through a set of dummy object names that are added beforehand – see the translation of obj. Similarly for names of methods.

The above encoding introduces a mutex ℓ_m for every method m of the object, as well as a mutex ℓ for the ctr method. A method calls ctr to check whether it has the right to continue on. This invocation takes two arguments: a dummy object name related to the method and the current state of the object, stored into the field a_c. The method body is blocked on its own lock.

The method ctr verifies whether a method name belongs to an enabled set through a sequence of two name matchings – see the term P –: in case the method name belongs to the current enabled set, ctr releases the lock of the method, thus triggering its execution. Otherwise ctr emits again the same call to itself – see the definition of M_j. This introduces divergence in the encoding.

The control also invokes the method CTR. This is useless at this stage because the body of CTR is $\mathsf{0}$. However it is important to implement inheritance.

To formalize the correspondence between a term and its translation, we have to lift the encoding $[\![\cdot]\!]$ to solutions. Let $(o = [\sigma])_{o.a:=o'}$ be the object $o = [(a = o')/\sigma]$.

$$
\begin{array}{rcll}
[\![\mathcal{O} \Vdash \mathcal{P}]\!] & \triangleq & \biguplus_{o=[\sigma], r \in \mathcal{O}} \Big(& (o = [[\![\sigma, r]\!]^{o}_{CTR} / (CTR(x,y) = \mathsf{0})])_{o.\ell_c = f\!\!f}, \\
 & & & o_\rho = [\;]^{\rho = \widetilde{m} \in r},\; o_m = [\;]^{m(\widetilde{x}) = p \in \sigma} \Big) \\
 & & \biguplus_{o = _{\widetilde{m}}[\sigma], r \in \mathcal{O}} \Big(& (o = [[\![\sigma, r]\!]^{o}_{CTR} / (CTR(x,y) = \mathsf{0})])_{o.a_c = o_{\rho'}}, \\
 & & & o_\rho = [\;]^{\rho = \widetilde{m} \in r},\; o_m = [\;]^{m(\widetilde{x}) = p \in \sigma} \Big) \\
 & & \Vdash & \biguplus_{p \in \mathcal{P}} [\![p]\!]
\end{array}
$$

where $\rho' = \widetilde{m}$ is an enabled set in r. For simplicity we assume that enabled sets labels of a same object always mark different sets of methods. (The general case can be reduced to this one, up-to barbed congruence).

Lemma 1.5.1 *Let $\mathcal{S}$ and $\mathcal{T}$ be solutions of the* RCHAM *defined in Table 1.4. Then*

(i) *If $\mathcal{S} \rightleftharpoons^* \longrightarrow \rightleftharpoons^* \mathcal{T}$ then $[\![\mathcal{S}]\!] \Longrightarrow^* [\![\mathcal{T}]\!]$;*
(ii) *if $\mathcal{S} \Downarrow_o$ then $[\![\mathcal{S}]\!] \Downarrow_o$.*

The adequacy of our encoding is a straightforward consequence of this lemma. (Barbed congruence may be easily transposed to the object calculus with enabled sets.)

Theorem 1.5.2 *For p, q two processes in Table 1.4, $[\![p]\!] \approx [\![q]\!]$ implies $p \approx q$.*

The encoding of classes and inheritance. In this paragraph we demonstrate that the full class based calculus of Table 1.5, with the operations over enabled sets, may be encoded into the calculus without enabled sets. This definitely fixes the expressive power of enabled sets.

Table 1.5 also collects the syntax and the compilation step of the language with enabled sets. Note that class definitions now carry a not empty sequence of enabled sets and enabled sets are first class entities since it is possible to modify them along the inheritance. These changes are done by $c \,\&\, \widetilde{m}$, meaning that every enabled set of the class c is inherited provided it contains the methods $\widetilde{m}$; and $c + \widetilde{m}$ which means that every enabled set of c is augmented with the methods $\widetilde{m}$.

We define two translation function $[\![\cdot]\!]_E$ and $(\![\cdot]\!)^{o,E}_{CTR}$ taking processes and definitions, respectively. The function E takes a class name c and returns the set of of method names in c. The set of methods in a class c may be determined by a type system or by an abstract interpretation. The name o

The Syntax:

$$
\begin{array}{rcl}
p & ::= & \mathsf{0} \\
 & \mid & o.m(\widetilde{e}) \\
 & \mid & o.a := e\,;\,p \\
 & \mid & \mathsf{obj}\ o = c\ \mathsf{in}\ p \\
 & \mid & \mathsf{class}\ c = \varsigma(x)\,d, r\ \mathsf{in}\ p \\
 & \mid & p \,|\, p \\
 & \mid & \mathsf{become}(o.\rho)
\end{array}
\qquad
\begin{array}{rcl}
d & ::= & [\sigma] \\
 & \mid & c \\
 & \mid & \varsigma(x)\,d \\
 & \mid & [\sigma]\ \mathsf{extends}\ d \\
 & \mid & d\ \mathsf{hide}\ m\ \mathsf{as}\ s \\
 & \mid & c \,\&\, \widetilde{m} \\
 & \mid & c + \widetilde{m}
\end{array}
$$

The Compilation: (d is a shortening for $d, \emptyset$ and $[\sigma] \,\&\, \widetilde{m} = [\sigma] + \widetilde{m} = [\sigma]$)

For definitions: $\overset{x}{\rightsquigarrow}$

$$
\begin{array}{rcl}
\varsigma(z)\ d, r & \overset{x}{\rightsquigarrow} & d\{x/z\}, r \\
[\sigma]\ \mathsf{extends}\ [\sigma'], r & \overset{x}{\rightsquigarrow} & [\sigma/\sigma'], r \\
{[a_i = o_i{}^{i\in 1..h}, m_i(\widetilde{o_i}) = p_i{}^{i\in 1..k}], r} \atop {\mathsf{hide}\ m\ \mathsf{as}\ s} & \overset{x}{\rightsquigarrow} & [a_i = o_i{}^{i\in 1..h}, m_i\{s/m\}(\widetilde{o_i}) = p_i{}^{i\in 1..k}], \\
 & & r \\
d, r \,\&\, \widetilde{m} & \overset{x}{\rightsquigarrow} & d \,\&\, \widetilde{m}, \bigcup_{\rho:\widetilde{m}\cup\widetilde{n}\in r} \rho : \widetilde{m} \cup \widetilde{n} \\
d, r + \widetilde{m} & \overset{x}{\rightsquigarrow} & d + \widetilde{m}, \bigcup_{\rho:\widetilde{n}\in r} \rho : (\widetilde{m} \cup \widetilde{n})
\end{array}
$$

+ closure under contexts $D[\,\cdot\,] ::= [\,\cdot\,] \mid [\sigma]\ \mathsf{extends}\ D[\,\cdot\,] \mid D[\,\cdot\,]\chi$
(χ ranges over "$\mathsf{hide}\ m\ \mathsf{as}\ s$", "$\&\,\widetilde{m}$", "$+\widetilde{m}$", "$, r$")

For processes: $\rightsquigarrow$

$$
\mathsf{class}\ c = \varsigma(x)\,[\sigma], r\ \mathsf{in}\ p \ \rightsquigarrow\ p\{\varsigma(x)\,[\sigma], r/c\}
$$

$$
\frac{D[d, r] \ \overset{x}{\rightsquigarrow}\ D'[d', r']}{\mathsf{class}\ c = \varsigma(x)\,D[d, r] \rightsquigarrow\ \mathsf{class}\ c = \varsigma(x)\,D'[d', r']\ \mathsf{in}\ p}
$$

+ context closure

Table 1.5. *Classes and inheritance in enabled sets*

is the self-object and CTR is the super method name invoked by the control *ctr*.

Let $M = \{m_1, \cdots, m_k\}$ be the set of method names of the class c and let $S = \{s_1, \cdots, s_k\}$ be a fresh set of super names. We always take fresh names o, CTR, $CTR1$, EV. We only detail the cases where $[\![\,\cdot\,]\!]_E$ and $(\![\,\cdot\,]\!)^{o,E}_{CTR}$ are not homomorphic:

$$
\begin{array}{rcl}
[\![\mathsf{class}\ c = \varsigma(x)\, d, r\ \mathsf{in}\ p]\!]_E & \triangleq & \mathsf{obj}\ o_n = [\ ]^{n \in \widetilde{M}},\ o_\rho = [\ ]^{\rho = \widetilde{m} \in r}\ \mathsf{in} \\
& & \mathsf{class}\ c = \varsigma(x)\ (\![d, r]\!)^{x,E}_{CTR}\ \mathsf{in}\ [\![p]\!]_{E+(c \mapsto M)} \\
(\![[\sigma], r]\!)^{o,E}_{CTR} & \triangleq & [[\![\sigma, r]\!]^{o}_{CTR}] \\
(\![c]\!)^{o,E}_{CTR} & \triangleq & c
\end{array}
$$

$$
(\![[\sigma]\ \mathsf{extends}\ d, r]\!)^{o,E}_{CTR} \ \triangleq\ (\![[\sigma], r]\!)^{o,E}_{CTR1}\ \mathsf{extends}\ ((\![d, r]\!)^{o,E}_{CTR}\ \mathsf{hide}\ ctr\ \mathsf{as}\ CTR1)
$$

$$
\begin{array}{rcl}
(\![c \,\&\, \widetilde{m}]\!)^{o,E}_{CTR} & \triangleq & [\ ctr(x_{mth}, x_c) = o.EV(x_{mth}, x_c, o.a_i{}^{m_i \in \widetilde{m}}, o.a_j{}^{m_j \in E(c) \setminus \widetilde{m}}), \\
& & \quad EV(x_{mth}, x_c, x_1, \cdots, x_k) = \\
& & \qquad \prod_{i \in 1..l} [x_{mth} = x_i] o.CTR1(x_{mth}, x_c),\ \mathsf{0} \\
& & \qquad \mid \prod_{i \in l+1..k} [x_{mth} = x_i] (o.ctr(x_{mth}, o.a_c) \mid release(o.\ell)),\ \mathsf{0} \\
& &]\ \mathsf{extends}\ (c\ \mathsf{hide}\ ctr\ \mathsf{as}\ CTR1)
\end{array}
$$

$$
\begin{array}{rcl}
(\![c + \widetilde{m}]\!)^{o,E}_{CTR} & \triangleq & [\ ctr(x_{mth}, x_c) = o.EV(x_{mth}, x_c, x.a_i{}^{m_i \in \widetilde{m}}, o.a_j{}^{m_j \in E(c) \setminus \widetilde{m}}), \\
& & \quad EV(x_{mth}, x_c, x_1, \cdots, x_k) = \\
& & \qquad\qquad \prod_{i \in 1..l} [x_{mth} = x_i] (acquire(o.\ell)\,;\, release(o.\ell_{m_i})),\ \mathsf{0} \\
& & \qquad\qquad \mid \prod_{i \in l+1..k} [x_{mth} = x_i] o.CTR1(x_{mth}, x_c),\ \mathsf{0} \\
& &]\ \mathsf{extends}\ (c\ \mathsf{hide}\ ctr\ \mathsf{as}\ CTR1)
\end{array}
$$

where l is the the cardinality of $\widetilde{m}$.

Note that the dummy names o_n and o_ρ are now introduced once for all the instances of a class. The translation of "$[\sigma]$ **extends** d, r" is done in two steps: first d, r is translated, by renaming the control method with a fresh name *CTR1*. The resulting expression is extended with the translation of $[\sigma], r$, where the control method now invokes *CTR1*. The overall expression has unique locks ℓ and ℓ_m, for every m, because of the updating operation. The translations of "$c \,\&\, \widetilde{m}$" and "$c + \widetilde{m}$" only refine the control method of the inherited class. The new behaviour performs additional checks (according to the primitives $\&\, \widetilde{m}$ and $+\widetilde{m}$) and then forwards the invocation to the old behaviour of *ctr*.

The next theorem states the correctness of the encoding. Its proof consists in providing a barbed congruence for every pair of terms p, p' such that $p \rightsquigarrow p'$.

Theorem 1.5.3 *Let p be a process of the language in Table 1.5 without free class names. If $p \rightsquigarrow p'$ then $[\![p]\!]_\emptyset \approx [\![p']\!]_\emptyset$.*

1.5.2 Method guards

A *method guard* is a boolean expression attached to each method that must be satisfied before the execution of the method may start. Method guards take terms that are constants, variables, object fields and synchronization counters. A *synchronization counter* is a value that specify the number of started method executions, completed executions, pending invocations, etc. These counters are updated by the run-time system.

Method guards have been proposed as the synchronization mechanism of `Guide` [DKM+89]. We illustrate this mechanism by rewriting the asynchronous channel in Example 1.3.1 into a `Guide`-like notation. The notation should be self-explanatory, nevertheless we will clarify it afterwards.

$$
\begin{array}{ll}
\textsf{class}\ ch & = \varsigma(x) \\
& [v = x, \\
& \ \ read(o) = \textsf{obj}\ o' = [v = o']\ \textsf{in}\ o'.v := x.v\,;\, o.back(o'.v)\ : \\
& \ \ \ [\text{completed}(write) = \text{completed}(read) + 1][\text{current}(write) = 0], \\
& \ \ write(o, o') = x.v := o'\,;\, o.ack()\ : \\
& \qquad\quad [\text{completed}(write) = \text{completed}(read)][\text{current}(read) = 0] \\
&]
\end{array}
$$

Observe that *read*, to be executed, requires that a *write* has been already performed – the number of previous *write*s must be greater – and no other *read* is currently executed – because *read*s consume the message. On the contrary, a message can be written provided the previous message has been read – the number of completed *write*s and *read*s is the same.

The technique for translating method guards into our language is the same as for enabled set: we first deal with objects and then with classes.

Table 1.6 collects the syntax and semantics of the object calculus with method guards. We consider synchronization counters as object fields that are updated by the run-time system. For simplicity, we only model the "started" synchronization counter that specifies the number of started executions of methods. Let a_m^s be the field of the "started" counter of the method m. The general case requires a continuation passing style in order to set synchronization counters when a method execution terminates.

Note that expressions that occur in method guards may be standard expressions (hence, accesses to method guards) and integer numbers. The case of expressions with arithmetic operations may be also handled.

The semantics of the calculus with method guards consists of one new rule, managing method invocations. The test of method guards is performed by the judgments $\mathcal{O} \models g$.

The Syntax: p, without the operation become, and e are as in Table 1.4;
the process obj is redefined as follows (n are integer numbers):

$$p \ ::= \ \cdots \mid \mathsf{obj}\ o = [a_i = o_i{}^{i\in 1..h}, m_i(\widetilde{x_i}) = p_i \ : \ g_i{}^{i\in 1..k}]\ \mathsf{in}\ p$$

$$g \ ::= \ [\varepsilon = \varepsilon] \mid g\, g \qquad\qquad \varepsilon \ ::= \ e \mid n$$

The judgment $\mathcal{O} \models g$:

$$\mathcal{O} \models [o = o] \qquad\qquad \frac{\mathcal{O} \models [e = e'] \quad \mathcal{O} \models g}{\mathcal{O} \models [\varepsilon = \varepsilon']\, g}$$

$$\frac{\mathcal{O} \models [o' = \varepsilon] \qquad o = [a = o', \sigma] \in \mathcal{O}}{\mathcal{O}, \models [o.a = \varepsilon]} \qquad \frac{\mathcal{O}, \models [\varepsilon = o'] \qquad o = [a = o', \sigma] \in \mathcal{O}}{\mathcal{O} \models [\varepsilon = o.a]}$$

The Semantics: Rules (Nil), (Asg), (Arg), (Exp), (Obj) of Table 1.1 +
(Par) of Table 1.3 +

$$(\textsc{Met}) \quad \mathcal{O}, o = [a^s_m = h,\ m(\widetilde{x}) = p \ : \ g, \sigma] \Vdash o.m(\widetilde{o'}) \ \longrightarrow \ \mathcal{O}, o = [a^s_m = h+1,\ m(\widetilde{x}) = p \ : \ g, \sigma] \Vdash p\{\widetilde{o'}/\widetilde{x}\}$$

With side condition:

$$(\textsc{Met}) \quad \text{provided } \mathcal{O}, o = [a^s_m = h,\ m(\widetilde{x}) = p \ : \ g, \sigma] \models g.$$

Table 1.6. *The concurrent object calculus with method guards*

The encoding of objects. To implement method guards in our basic calculus we use the full expressive power of name comparison. We also augment the calculus with integers and summations, with the expected semantics.

The following translations $[\![\cdot]\!]$ and $[\![\cdot]\!]_o$ take a process p and a record $[\sigma]$, respectively, and give a process or a record in the calculus in Sections 1.2.4 and 1.3. We focus on the unique cases where $[\![\cdot]\!]$ and $[\![\cdot]\!]_o$ are not homomorphic:

$$\begin{array}{l}
[\![\, [a_i{}^{i\in 1..h}, m_i(\widetilde{x_i}) = p_i \ : \ [e_1 = e_2]\cdots[e_{j-1} = e_j]^{i\in 1..k}]]\!]_o \triangleq \\
[\quad \ell = t\!t,\ \ell_{m_i} = f\!f^{i\in 1..k},\ a^s_{m_i}{}^{i\in 1..k},\ a_i{}^{i\in 1..h}, \\
\quad m_i(\widetilde{x_i}) = acquire(o.\ell)\,;\,(o.do_{m_i}(e_1,\cdots,e_j,\widetilde{x_i}) \mid acquire(o.\ell_{m_i})\,;\,[\![p_i]\!])^{\,i\in 1..k} \\
\quad do_{m_i}(y_1,\cdots,y_j,\widetilde{x_i}) = [y_1 = y_2]\cdots[y_{j-1} = y_j] \\
\qquad\qquad o.a^s_{m_i} := o.a^s_{m_i} + 1\,;\,(release(o.\ell) \mid release(o.\ell_{m_i})), \\
\qquad\qquad (release(o.\ell) \mid o.m_i(\widetilde{x_i})) \\
]
\end{array}$$

$$[\![\mathsf{obj}\ o' = \varsigma(o)\,[\sigma]\ \mathsf{in}\ p]\!] \ \stackrel{\triangle}{=}\ \mathsf{obj}\ o' = \varsigma(o)\,[\![[\sigma]]\!]_o\ \mathsf{in}\ [\![p]\!]$$

The translation $[\![\cdot]\!]$ introduces a global lock ℓ, and, for every method m of the object, a lock ℓ_m and an auxiliary method do_m, that checks the method guard of m. When m is invoked, it acquires the global lock to verify its guard (method guards are tested in a mutual exclusive way) and invokes do_m. If the test is positive both the locks ℓ and ℓ_m are released. Actually, before these releases, the synchronization counters are updated accordingly. If the test of the method guards is negative then ℓ is released and the method invocation is emitted again.

As for enabled sets, to formalize the correspondence between a term and its translation, we lift the encoding $[\![\cdot]\!]$ to solutions:

$$[\![\mathcal{O} \Vdash \mathcal{P}]\!] \ \stackrel{\triangle}{=}\ \biguplus_{o=[\sigma]\in\mathcal{O}} \Big(o = [\![[\sigma]]\!]\Big) \ \Vdash\ \biguplus_{p\in\mathcal{P}} [\![p]\!]$$

Lemma 1.5.4 *Let $\mathcal{S}$ and $\mathcal{T}$ be solutions of the* RCHAM *defined in Table 1.6. Then*

(i) *If $\mathcal{S} \rightleftharpoons^* \longrightarrow \rightleftharpoons^* \mathcal{T}$ then $[\![\mathcal{S}]\!] \Longrightarrow^* [\![\mathcal{T}]\!]$;*

(ii) *if $\mathcal{S} \Downarrow_o$ then $[\![\mathcal{S}]\!] \Downarrow_o$.*

It is an exercise to equip the calculus with method guards with barbed congruence. Then, the adequacy of our encoding is a straightforward consequence of the above lemma.

Theorem 1.5.5 *For p, q two processes in Table 1.6, $[\![p]\!] \approx [\![q]\!]$ implies $p \approx q$.*

The encoding of classes and inheritance. Table 1.7 defines the class extension of the calculus with method guards. The new primitives, with respect to the previous class calculi, are those refining the method guards along the inheritance. There are two basic ones: "$c : m + g$" and "$c : m\#g$". The former adds g to the method guards of m in c (it acts as a conjunction). The latter removes the method guards of m in c and promotes g as new method guards for m. The compilation step $\stackrel{x}{\rightsquigarrow}$ formalizes this behaviour.

The language in Table 1.7 can be encoded into our class language of Sections 1.2.4 and 1.3 with the name comparison operator. We use two encoding functions $[\![\cdot]\!]$, for processes, and $[\![\cdot]\!]_o$, for definitions. These two functions are almost homomorphisms, except for the following cases (DO_m

The Syntax:

$$
\begin{array}{rcl}
p & ::= & \mathsf{0} \\
 & \mid & o.m(\widetilde{e}) \\
 & \mid & o.a := e\,;\,p \\
 & \mid & \mathsf{obj}\ o = c\ \mathsf{in}\ p \\
 & \mid & \mathsf{class}\ c = \varsigma(x)\,d\ \mathsf{in}\ p \\
 & \mid & p \mid p
\end{array}
\qquad
\begin{array}{rcl}
d & ::= & [a_i = o_i{}^{i\in 1..h},\, m_i(\widetilde{x_i}) = p_i \,:\, g_i{}^{i\in 1..k}] \\
 & \mid & c \\
 & \mid & \varsigma(x)\,d \\
 & \mid & [\sigma]\ \mathsf{extends}\ d \\
 & \mid & d\ \mathsf{hide}\ m\ \mathsf{as}\ s \\
 & \mid & c \,:\, w
\end{array}
\qquad
\begin{array}{rcl}
w & ::= & m + g \\
 & \mid & m\#g \\
 & \mid & w \,:\, w
\end{array}
$$

The Compilation: (d is a shortening for $d\,:\,\emptyset$ and $[\sigma] : m + g = [\sigma] : m\#g = [\sigma]$ if m is not a method occurring in σ)

For definitions: $\stackrel{x}{\leadsto}$

$$
\begin{array}{rcl}
\varsigma(z)\ d & \stackrel{x}{\leadsto} & d\{x/z\} \\
[\sigma]\ \mathsf{extends}\ [\sigma'] & \stackrel{x}{\leadsto} & [\sigma/\sigma'] \\
{[a_i = o_i{}^{i\in 1..h}, m_i(\widetilde{o_i}) = p_i \,:\, g_i{}^{i\in 1..k}]} \ \mathsf{hide}\ m\ \mathsf{as}\ s & \stackrel{x}{\leadsto} & [a_i = o_i{}^{i\in 1..h}, m_i\{s/m\}(\widetilde{o_i}) = p_i \,:\, g_i{}^{i\in 1..k}] \\
{[m(\widetilde{x}) = p \,:\, g,\, \sigma]} \,:\, m + g' & \stackrel{x}{\leadsto} & [m(\widetilde{x}) = p \,:\, gg',\, \sigma] \\
{[m(\widetilde{x}) = p \,:\, g,\, \sigma]} \,:\, m\#g' & \stackrel{x}{\leadsto} & [m(\widetilde{x}) = p \,:\, g',\, \sigma]
\end{array}
$$

+ closure under contexts $D[\,\cdot\,] ::= [\,\cdot\,] \mid [\sigma]\ \mathsf{extends}\ D[\,\cdot\,] \mid D[\,\cdot\,]\chi$
(χ ranges over "$\mathsf{hide}\ m\ \mathsf{as}\ s$", "$:\ m + g$", "$:\ m\#g$")

For processes: $\leadsto$

$$\mathsf{class}\ c = \varsigma(x)\,[\sigma]\ \mathsf{in}\ p \leadsto p\{\varsigma(x)\,[\sigma]/c\}$$

$$\frac{d \stackrel{x}{\leadsto} d'}{\mathsf{class}\ c = \varsigma(x)\,d\ \mathsf{in}\ p \leadsto \mathsf{class}\ c = \varsigma(x)\,d'\ \mathsf{in}\ p}$$

+ context closure

Table 1.7. *Classes and inheritance in method guards*

and $DO1_m$ are always fresh names):

$$[\![\mathsf{class}\ c = \varsigma(o)\,d\ \mathsf{in}\ p]\!] \stackrel{\triangle}{=} \mathsf{class}\ c = \varsigma(o)\,[\![d]\!]_o\ \mathsf{in}\ [\![p]\!]$$

$$
\begin{array}{l}
[\![c : w \,:\, (m + [e_1 = e_2]\cdots[e_{j-1} = e_j])]\!]_o \stackrel{\triangle}{=} \\
\qquad [\, do_m(y_1, \cdots, y_k, \widetilde{x}) = o.DO1_m(y_1, \cdots, y_k, e_1, \cdots, e_j, \widetilde{x})\,, \\
\qquad\ DO1_m(y_1, \cdots, y_k, z_1, \cdots, z_j, \widetilde{x}) = \\
\qquad\qquad\qquad [z_1 = z_2]\cdots[z_{j-1} = z_j]\ o.DO_m(y_1, \cdots, y_k, \widetilde{x})\,, \\
\qquad\qquad\qquad\qquad (release(o.\ell) \mid o.m(\widetilde{x})) \\
\qquad]\ \mathsf{extends}\ ((c \,:\, w)\ \mathsf{hide}\ do_m\ \mathsf{as}\ DO_m)
\end{array}
$$

$$
\begin{array}{l}
[\![c : w \, : \, (m\#[e_1 = e_2]\cdots[e_{j-1} = e_j])]\!]_o \stackrel{\triangle}{=} \\
\qquad [\, do_m(y_1, \cdots, y_k, \widetilde{x}) = o.DO_m(e_1, \cdots, e_j, \widetilde{x}) \, , \\
\qquad\; DO_m(z_1, \cdots, z_j, \widetilde{x}) = \\
\qquad\qquad\qquad [z_1 = z_2]\cdots[z_{j-1} = z_j] \, (release(o.\ell) \mid release(o.\ell_m)) \, , \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (release(o.\ell) \mid o.m(\widetilde{x})) \\
\qquad] \; \mathsf{extends} \; (c \, : \, w)
\end{array}
$$

As expected, the inheritance of a class where method guards are refined only concern the control methods *do*. In particular, when the definition is "$m + g$", do_m is refined by firstly checking g and then, in case of positive answer, checking the rest of the guard (forwarding the invocation to the super do_m). In case the test of g fails, the call of m must be delayed and the lock ℓ is released. The encoding of "$m\#g$" is similar, except that no forward to the super do_m is done because the method guard of m in c has been overridden.

The next theorem gives the correctness of the encoding. The proof is similar to the corresponding theorem for enabled sets.

Theorem 1.5.6 *Let p be a process of the language in Table 1.7 without free class names. If $p \rightsquigarrow p'$ then $[\![p]\!]_\emptyset \approx [\![p']\!]_\emptyset$.*

1.6 Conclusions

We have reviewed inheritance both in the sequential and concurrent setting. To this end we have defined a concurrent object-oriented language and used it to systemize the proposals in the literature.

We strongly believe that inheritance is much more important for concurrent objects, due to the variety of forms it shows up. However to benefit of this programming concept one should define a small set of expressive operators, with a formal semantics and for which it is possible to develop proof techniques. Some progress in this direction is done in [FLMR99], where a class based calculus has been developed out of a well-known process calculus, and a small set of algebraic operators for inheriting concurrent code is defined.

Whether the operators proposed in [FLMR99] are the right ones or not is open to debate. Future research could bring strong motivations or positive or negative answers and therefore devise a rigorous *algebraic theory for the inheritance of the control.*

Bibliography

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[ACS98] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[Ame89] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, October 1990.

[Ber94] L. Bergmans. *Composing Concurrent Objects*. Ph.D. thesis, University of Twente, 1994.

[BF96] P. Di Blasio and K. Fisher. A calculus for concurrent objects. In *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, 1996.

[Car94] L. Cardelli. Obliq A language with distributed scope. SRC Research Report 122, Digital Equipment, June 1994.

[CH73] R. H. Campbell and A. N. Habermann. The Specification Of Process Synchronisation by Path Expressions. In *Lecture Notes in Computer Science, No. 16*, pages 89–102. Springer Verlag, 1973.

[CUCH91] C. Chambers, D. Ungar, B. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3):207–222, July 1991.

[DKM+89] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, and X. Rousset de Pina. A synchronization mechanism for typed objects in a distributed system. *ACM SIGPLAN Notices*, 24(4):105–107, April 1989.

[DZ98] S. Dal-Zilio. Quiet and bouncing objects: Two migration abstractions in a simple distributed blue calculus. In H. Hüttel and U. Nestmann, editors, *SOAP '98: Semantics of Objects as Processes*, volume NS-98-5 of *BRICS Notes Series*, pages 35–42, 1998.

[FG96] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Conference record of the 23th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 372–385, 1996.

[FHM93] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 26–38. IEEE Computer Society Press, June 1993.

[FLMR97] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, Lecture Notes in Computer Science 1243, pages 196–212, 1997.

[FLMR99] C. Fournet, C. Laneve, L. Maranget, and D. Remy. Assembling objects with internal concurrency. Submitted for pubblication, July 1999.

[Frø92] S. Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings ECOOP'92*, LNCS 615, pages 185–196. Springer-Verlag, June 1992.

[GH98] A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In *HLCL '98: High-Level Concurrent Languages*, volume 16(3) of *entcs*, 1998.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*, pages 674–681. Addison-Wesley, 1983.

[GR96] A. D. Gordon and G. D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *23rd ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 386–395. ACM Press, 1996.

[KMMPN87] B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. The BETA programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. The MIT Press, Cambridge, Mass, 1987.

[McH94] C. McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance.* PhD thesis, Department of Computer Science, Trinity College, 1994.

[Mes93] J. Meseguer. Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 220–246. Springer-Verlag, 1993.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Automata, Languages and Programming, 19th International Colloquium*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[MY93] 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. MIT Press, 1993.

[NP96] U. Nestmann and B. C. Pierce. Decoding choice encodings. In *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 179–194. Springer-Verlag, 1996.

[Sar97] V. Saraswat. Java is not Type Safe. Technical report, AT&T research, 1997. http://www.att.research.com/ vi.

[Sny87] A. Snyder. Inheritance and the development of encapsulated software components. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. MIT Press, 1987.

[TS89] C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled Sets. In *Proceedings of the OOPSLA '89 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 103–112, 1989.

[Vas94] V. T. Vasconcelos. Typed concurrent objects. *Lecture Notes in Computer Science*, 821:100–117, 1994.

[YBS86] A. Yonezawa, J. P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. *ACM SIGPLAN Notices*, 21(11):258–268, November 1986.