Basic Research in Computer Science

BRICS NS-98-5 Hüttel & Nestmann (eds.): SOAP '98 Proceedings

Proceedings of the Workshop on

Semantics of Objects as Processes

SOAP '98

Aalborg, Denmark, July 18, 1998

Hans Hüttel
Uwe Nestmann
(editors)

BRICS Notes Series **NS-98-5**

ISSN 0909-3206 **June 1998**

Copyright © 1998, BRICS, Department of Computer Science University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/98/5/`

Proceedings of the Workshop on

Semantics of Objects as Processes

SOAP '98

July 18
Aalborg, Denmark

Hans Hüttel
Uwe Nestmann
(editors)

Preface

One of the most widespread programming paradigms today is that of object-oriented programming. With the growing popularity of the language C++ and the advent of Java as the language of choice for the World Wide Web, object-oriented programs have taken centre stage. Consequently, the past decade has seen a flurry of interest within the programming language research community for providing a firm semantic basis for object-oriented constructs.

Recently, there has been growing interest in studying the behavioural properties of object-oriented programs using concepts and ideas from the world of concurrent process calculi, in particular calculi with some notion of mobility. Not only do such calculi, as the well-known π-calculus by Milner and others, have features like references and scoping in common with object-oriented languages; they also provide one with a rich vocabulary of reasoning techniques firmly grounded in structural operational semantics.

The process calculus view has therefore proven to be advantageous in many ways for semantics and verification issues. On the one hand, the use of encodings of object-oriented languages into existing typed mobile process calculi enables formal reasoning about the correctness of programs; on the other hand, using standard techniques from concurrency theory in the setting of calculi for objects may help in reasoning about objects, e.g. by finding appropriate and mathematically tractable notions of behavioural equivalences. Encodings may also help clarify the overlap and differences of objects and processes, and suggest how to integrate them best in languages with both.

The aim of the one-day SOAP workshop, which is a satellite workshop of ICALP 98, has been to bring together researchers working mainly in this area, but in related fields as well, where other process models or calculi are used as a basis for the semantics of objects.

Among the submitted abstracts, six were recommended by the programme committee (Martín Abadi, Hans Hüttel, Josva Kleist, and Uwe Nestmann) and are presented in these proceedings. According to the more informal character of the workshop, there was no formal refereeing process. It is expected that the abstracts presented in these proceedings will appear elsewhere at other conferences or in journals.

We would like to thank the organizers of ICALP '98 for helping us set up the SOAP workshop and BRICS for the publication of these proceedings.

Hans Hüttel
June 1998 Uwe Nestmann

Table of Contents

The workshop will be held in Aalborg Congress and Culture Centre on July 18, 1998, in the order appearing in these proceedings.

Carlos Herrero, Javier Oliver
Object-Oriented Parallel Label-Selective λ-calculus 7

Claudia Balzarotti, Fiorella De Cindio, Lucia Pomello
Observation equivalences for type and implementation inheritances 17

António Ravara, Pedro Resende, Vasco Vasconcelos
Towards an Algebra of Dynamic Object Types 25

Andrew D. Gordon, Paul Hankin
A Concurrent Object Calculus:
Summary of the Operational Semantics 31

Silvano Dal-Zilio
Quiet and Bouncing Objects:
Two Migration Abstractions in a Simple Distributed Blue Calculus 35

Hans Hüttel, Josva Kleist, Uwe Nestmann, Davide Sangiorgi
Surrogates in Øjeblik: Towards Migration in Obliq 43

Object-Oriented Parallel Label-Selective λ-calculus*

Carlos Herrero[§] Javier Oliver[§]

[§] DSIC, UPV, Camino de Vera s/n, Apdo. 22012, 46022 Valencia, Spain.
E-mail: cherrero,fjoliver@dsic.upv.es.

Abstract

LCEP is a calculus for modelling concurrent systems. The efforts to use it to represent object oriented features have been successfully treated in this paper. We present an operational semantics for a parallel object-oriented programming language by means of a phrase-by-phrase translation from the language into an extension of LCEP in which only a few changes from the original LCEP are made.

Keywords: Concurrency, Extensions of λ-calculus, Object-Oriented, Process Algebras.

1 Introduction

We can find many studies in computation to produce an elegant semantics which correctly defines languages with concurrent features. The role played by the λ-calculus in computation theory is well-known but while it was adopted by D.S. Scott and C. Strachey as a semantic basis for programming languages, it cannot be successfully used with concurrence features. In 1980, R. Milner proposed CCS, [7], which is a calculus for modelling concurrent systems. His contribution was to view computational entities as agents whose interaction is the basic behavioural unit. This calculus was the basis for the π-calculus [8] which emerged as a refinement of CCS with two primitive notions as foundations: the *name* and the *agent*. A *name* is used to provide access to agents. Specifically, two agents that share a name can interact by using it. In addition, an agent can send a name in an interaction, and therefore one agent can transfer its ability to interact with other agents to the other. On the other hand, with the label-selective λ-calculus of H. Aït-Kaci [1] the distinction between terms and channels appears. A term plays the role of an agent of the π-calculus and a channel (numeric or symbolic labels) represents the way through which terms can send information. With the extension of this calculus to LCEP [6] (Parallel Label-Selective λ-calculus) we have a complete calculus for modelling concurrency by using numeric and/or symbolic labels and parallel operators, and also by representing polyadic functions.

Among the proposed approaches to parallel programming, the disciplines which are characteristic of parallel object-oriented programming have many desirable features. However, providing adequate semantics for these languages is very complex. The family of POOL languages is outstanding among the works in this field. This is the starting point for this work. In particular, the operational semantics of a member of this family of languages appeared in [2] and was later expanded by the denotacional semantics [3] based on metric spaces. Moreover, there are other semantics available which are associated to the family of POOL languages using process algebras [10]. The nature of the language suggests that correct models are complicated. More precisely, attempts to discern a clear vision of the central concept (i.e. the object) is very complex, although already important advances have been made in this direction. In this article, we show a semantics in LCEP of a slightly modified version of POOL, via a phrase-by-phrase translation. Each syntactic entity is represented as a parameterized agent. The representation of composite entities is constructed by using the operators of the calculus. In particular, we are interested in modelling

*This work was partially supported by CICYT, TIC 95-0433-C03-03.

the communication, the invocation of methods, and the transfer of results between objects. An object is represented as a composed term which is able to interact with others by asking for the execution of a certain method.

The remainder of the work is organized as follows. In Section 2, LCEP is briefly defined. In Section 3, the POOL language is described. In Section 4, we show the translation function of the different instructions from POOL into LCEP. In Section 5, we present an example of a POOL program translation. Finally, Section 6 presents some concluding remarks.

2 The Parallel Label-Selective λ-calculus

The Parallel Label-Selective λ-calculus (LCEP) [9] is the calculus that we use for modelling the LCEPOO language, which originated from a previous proposal by H. Aït-Kaci (the label-selective λ-calculus). This calculus is an extension of the λ-calculus in which function arguments are selected by labels. Symbolic and numeric labels can be used to name the communication channels. This has not been possible in other proposals and it allows for the use of currying and the labeled specification of parameters. It makes the ordering of the actual parameters in a function call independent of the presentation order of the formal parameters in the definition.

Let $V = \{x, y, z, \ldots\}$ be a set of variables, and $C = \{a, b, c, \ldots\}$ be a set of constants. P represents a channel name from a set of labels $\mathcal{L} = \mathbb{N} \cup \mathcal{S}$, where $m, n, \ldots$ denote numerical labels taken from $\mathbb{N}$. $\mathcal{S} = \{p, q, \ldots\}$ is a set of symbolic labels. $\mathbb{N} \cap \mathcal{S} = \emptyset$. 0 represents the null process.

The language of the formal system is $\mathcal{M}$. It is defined as follows:

$$\begin{array}{lcl} \mathcal{M} & ::= & 0 \mid a \mid x \mid \lambda_P x.M \mid \widehat{P}\ M \mid M \parallel M \mid M \circ M \mid (M + M) \mid !M \\ P & ::= & \text{number} \mid \text{symbol} \end{array}$$

Usually, the terms of this signature are called λ_{EP}-terms.

In concurrent programming, the terms of $\mathcal{M}$ represent processes. Operation symbols $\circ, \parallel, +, !$ are considered as process constructors. The communication $\lambda_P x.$ (abstraction or input) and $\widehat{P}$ (parameter passing or output) are also considered as process constructors, whose effect over a process M is to define processes ($\lambda_P x.M$ y $\widehat{P}M$, respectively) which are involved in functional applications under certain given conditions. Constants, variables and the null process form the atomic symbols of the language.

The interpretation of the process constructors is as follows: sequential composition ($\circ$): it defines a process $M \circ N$ from the processes M y N. The process N is executed after M; parallel composition ($\parallel$): it defines a new process $M \parallel N$ from the processes M and N. Both processes are executed simultaneously; non-deterministic choice ($+$): $M + N$ shows the capability of M or N (either one or the other) to take part in a communication. A communication can be made by one of them and the other disappears; replicator ($!$): $!M$ defines a process which produces the multiple generation of M. It acts as a warehouse, where it is possible to take an infinite number of the process M.

Operators have the subsequent priority order: $! > \widehat{P} > \circ > \lambda_P x. > + > \parallel$.

An occurrence of the variable x is *bound* in a process M iff it is in a term $\lambda_P x.M$ (for all $P \in \mathcal{L}$). In any other case, the occurrence of x is *free*. If x has, at least one free occurrence in M, we say that x is a *free* variable of M. We call $FV(M)$ the set of free variables of M. A process is *closed* if it does not have free variables.

The relations between the processes are described in terms of communications. The constructors $\lambda_P x.$ and $\widehat{P}$ provide the processes M and N with the possibility of communicating through a channel labelled with $P \in \mathcal{L}$ as follows: constructor of input ($\lambda_P x.$): through channel P process $\lambda_P x.M$ can receive a process which replaces the free occurrences of x in M by the incoming process through this channel; constructor of output ($\widehat{P}$): process $\widehat{P}M$ can send a process N through channel P. It then becomes inactive.

In addition, a partial order relation is defined in the set of labels which we denote as $\preceq_L$. It is required as a condition for the relation that the numerical label 0 be the minimum of the ordered set. We use the partial order relation in the set of labels (among others) to represent the

application order of the real parameters to the formal parameters of a function, which is analogous to how it is done in the λ-calculus in order to treat the problem of parameterization in function calls.

For more complete information about the calculus, see [9].

3 Parallel Object-Oriented Language (POOL)

When we work with the object-oriented programming style, we always describe a computational system as a collection of self-contained entities possessing data and methods. These entities are called *objects*. Objects interact by sending messages. There are two kinds of messages: a request by the sender for the receiver to execute one of its methods with its parameters (client to server), and a reply (server to client) in response to such a method invocation. The parameters and the reply are object references. The system evolution depends on the communications, and the creation/destruction of objects is made in computing time. In the variant of POOL considered [2], a program is a sequence of class declarations together with an indicator of which class constitutes the object root.

The declaration of a class consists of a sequence of variable and method declarations together with a sentence, the body of the class. Each object is an instance of a class and its creation executes a copy of the body of the class in parallel with all the other existing objects using its own copies of the variables and class methods. The sentences are built by means of the sequential, conditional and iterative operators of the expressions, instructions of assignment and the answer instruction, which means that an object accepts a request of execution of one of its methods. Among the expressions, we can find *new(B)*, whose evaluation creates a new object of the class B and returns the reference to this object; *self*, whose evaluation returns a reference to the object in which it happens; and $E!M(E_1,\ldots,E_n)$, whose evaluation implies the left to right evaluation of the $E, E_1, \ldots, E_n$, expressions, the sending to the object for which the value of E is a request to execute method M with these parameters. The value of the expression is the reference which has been returned by the object owner of the invoked method. The activity of the transmitter is interrupted until this value is received.

Therefore, the *expressions* in POOL, the *declaration of variables*, and the *sentences* are:

$$
\begin{array}{lll}
E & ::= & b \mid new(B) \mid self \mid X \mid E!M(E_1,\ldots,E_n) \mid M(E_1,\ldots,E_n) \mid \\
 & & \mid Vdecs \text{ in } E \mid S;E \\
Vdecs & ::= & var\ X_1 : A_1, \ldots, X_n : A_n \\
S & ::= & X := E \mid answer \mid E \mid S1;S2 \mid \text{if } E \text{ then } S_1 \text{ else } S_2 \mid \text{while } E \text{ do } S
\end{array}
$$

These sentences are *assignment*, *answer* (an object accepts the request), *expression*, *sequential composition*, *choice*, and *iteration*, respectively. Note that an expression is considered as a sentence.

The declaration of *methods* (*Mdec*), *sequences of method declarations* (*Mdecs*), *declarations of a class* (*Cdec*), and *units* (*Udec*) are given by:

$$
\begin{array}{lll}
Mdec & ::= & \text{method } M(Y_1 : C_1, \ldots, Y_p : C_p)\ C \text{ is } E \\
Mdecs & ::= & Mdec_1, \ldots, Mdec_q \\
Cdec & ::= & class\ A\ is\ Vdecs, Mdecs\ with\ body\ S \\
Udec & ::= & unit\ U\ is\ Cdec_1, \ldots, Cdec_r\ with\ root\ A
\end{array}
$$

where C is the resulting method class, $Y_1, \ldots, Y_p$ are the formal parameters, and E is the body.

The language is strongly typed, and several conditions are imposed. For example, in a unit declaration, A must be the name of one of the classes declared in $Cdec_1, \ldots, Cdec_r$ and if an expression $E!M(E_1,\ldots,E_n)$ appears in one of the class definitions and the type of E is C, then it must have a class C and a called method M of the appropriate type. The classes are only considered if they fulfill all the imposed conditions. See [2] for further details.

The language differs from POOL in some small syntactic details. The significant differences are that there is a general sentence *answer* (any method can be invoked) and not a conditional one (*answer + select*), which allows the exclusion of certain methods, and that the standard class

integer does not appear. These differences simplify the proposal language and it can be easily translated to LCEP.

For further details about the operational semantics of POOL and of the current version of POOL see [2] and [11], respectively.

4 The Translation

It is obvious that each syntactic entity is represented as a λ_{EP}-term and composite entities are represented by their components. LCEP cannot use multiple parameters as channels. However, we can communicate through numeric channels (using currying we can model polyadic functions) or by sending some processes in parallel. This problem has an easy treatment in π-calculus, because we can send names of channels through the channels.

In LCEP we have a numeric or symbolic label P, which represents a channel through which we send a process N from an output $\widehat{P}N$ into an input $\lambda_p x.M$, where the occurrences of x in M are replaced by the process N, i.e. every valid λ_{EP}-term (variables, constants, processes, etc. but not channels). The idea of conversion is to use the name of the channel to substitute the name of the agent. This channel is then used to send the parameters and a λ_{EP}-term, which works as *output* of the result instead of the name of the method and the associated parameters. Therefore, the sender process is like an *input* which is waiting for the reply through the same channel. When we need to send a name or channel, we actually send an *output* process through this channel. Consider an expression as a variable value or an *output* process which communicates or starts the expression from the represented channel. In this paper, the process perspective is presented. Therefore, the definition of classes comes from the definition of the objects'behaviour. To simplify the notation, construction blocks are used:

$$\begin{array}{lcl} Value(l,v) & \equiv & \widehat{l}\ v \\ Null(l) & \equiv & \widehat{l}\ 0 \\ Wait(l,l') & \equiv & \lambda_{l'} v.Value(l,v) \\ Return(l,l',l'') & \equiv & \lambda_{l''} v.\widehat{l'}\ v \circ Null(l) \end{array}$$

Value(l,v) represents the evaluation of an expression or variable. *Null(l)* is translated as an *output* through the channel l of the null term. *Wait(l,l')* is an abstraction (a waiting) and a later evaluation. *Return(l,l',l'')* is its complement.

Simple POOL expressions are translated into LCEP as follows:

$$\begin{array}{lcl} [new(B)](l,k) & \equiv & \widehat{k}(\lambda_{ret} x.\widehat{l}\ x) \\ [self](l,v) & \equiv & Value(l,v) \\ [Y](l,y) & \equiv & \widehat{y}(\widehat{r}\ 0) \circ \lambda_y v.Value(l,v) + \widehat{y}(\widehat{u}\ v') \circ \lambda_y v.Value(l,v) \\ [Var\ Y](y,v) & \equiv & Var(y,v) \\ Var(y,v) & \equiv & \lambda_y x_1.(x_1 \parallel \lambda_r x.(\widehat{y}\ v \circ Var(y,v)) \parallel \lambda_u v'.(\widehat{y}\ v' \circ Var(y,v'))) \end{array}$$

The creation of a new object is modelled as a call to the class and the later reception of the new object identifier (*Channel*). This is the only exception in which we can see a label as a variable. Using a variable is equivalent to updating it or to consulting it. In any case, the declaration of a variable is the parallel composition of two answers.

POOL classes'declaration in LCEP is:

$$\begin{array}{lcl} [Cdec](k) & \equiv & !\ \lambda_k x.((x \parallel \widehat{ret}\ c_i) \circ Newobject(c_i)) \\ Newobject(c_i) & \equiv & ([S] \parallel [Vdecs](\tilde{y},\tilde{v}) \parallel [Mdecs](\tilde{m})) \\ [Mdecs](\tilde{m}) & \equiv & [Mdec_1](m_1) \parallel \ldots \parallel [Mdec_i](m_i) \end{array}$$

In addition, we are going to make the request for the execution of a remote method into LCEP. First, let us see the translation of an object c_0 which requests the execution of a remote method m with only one evaluated parameter.

$$[\beta_0 ! M(\beta_1)](l,c_0,m) \equiv \widehat{c_0}(\widehat{m}\ \beta_1 \circ \widehat{m}(\lambda_m x.\widehat{ret}\ x) \circ \lambda_{ret} x.\widehat{l'}\ x) \circ Wait(l,l')$$

In this case, a term is sent to the object (received by an *answer* into the object body), and then the object waits using *Wait(l,l')*. The term which has been sent will execute three actions: (1) send the parameter to the method; (2) send a term which produces the answer of the method; (3) make a term which stays in the *answer* and which really sends the result to the *Wait(l,l')*.

The declaration of a one-parameter method is:

$$\begin{array}{lcl} [Mdec](m) & \equiv & !\, Mmethod(m) \\ Mmethod(m) & \equiv & Mhandle(m) \parallel [Var\ Y_1](y_1, v_1) \parallel \lambda_g x.[E](m) \\ Mhandle(m) & \equiv & \lambda_m x.\widehat{y_1}(\widehat{u}\ x) \circ \lambda_{y_1} x.0 \circ \lambda_m x.(x \parallel \widehat{g}\ 0) \end{array}$$

As you can see, the receipt of a parameter is translated by updating the variable Y_1, joined to the parallel composition with the answer sender term and with another that synchronizes with the evaluation of the method body.

Now let us see a similar process, but using multiple parameters:

$$\begin{array}{lcl} [\beta_0 !M(\beta_1, \ldots, \beta_n)](l, c_0, m) & \equiv & \widehat{c_0}(\widehat{m}\ \beta_1 \circ \ldots \circ \widehat{m}\ \beta_n \circ \widehat{m}(\lambda_m x.\widehat{ret}\ x)) \circ \\ & & \circ(\lambda_{ret} x.\widehat{l'}\ x) \circ Wait(l, l') \end{array}$$

The method declaration into the server object is:

$$\begin{array}{lcl} [Mdec](m) & \equiv & !\, Mmethod(m) \\ Mmethod(m) & \equiv & Mhandle(m) \parallel [Var\ Y_1](y_1, v_1) \parallel \ldots \parallel [Var\ Y_n](y_n, v_n) \parallel \lambda_g x[E](m) \\ Mhandle(m) & \equiv & \lambda_m x.\widehat{y_1}(\widehat{u}\ x) \circ \lambda_{y_1} x.0 \circ \lambda_m x.(\widehat{u}\ x) \circ \widehat{y_n}\ x.0 \circ \widehat{m}\ x(x \parallel \widehat{g}\ 0) \end{array}$$

If, instead of the evaluated parameters, we have expressions (knowing that a POOL expression must be left-to-right evaluated), we require that the evaluation of an expression E_i not start, since the evaluation of the expresion E_{i-1} has finished. This effect is successfully represented by:

$$\begin{array}{lcl} [\beta_0 !M(\beta_1, \ldots, \beta_{i-1}, E_i, \ldots, E_n)](l, c_0, m) & \equiv & \widehat{c_0}(\widehat{m}\ \beta_1 \circ \ldots \circ \widehat{m}\ \beta_{i-1} \circ \lambda_{l_i} \beta_i.\widehat{m}\ \beta_i \circ \widehat{g_{i+1}}\ 0 \circ \\ & & \circ \ldots \circ \lambda_{l_n} \beta_n.\widehat{m}\ \beta_n \circ \widehat{m}(\lambda_m x.\widehat{ret}\ x)) \circ \\ & & \circ \lambda_{ret} x.\widehat{l'}\ x \parallel [E_i](l_1) \parallel \lambda_{g_{i+1}} x.[E_{i+1}](l_{i+1}) \parallel \\ & & \parallel \ldots \parallel \lambda_{g_n} x.[E_n](l_n) \circ Wait(l, l') \end{array}$$

It is easy to see that the treatment of this expression may be similar to the treatment of a local call to execution of a method. Actually, it only differs in one feature. In a remote call, we need to generate a λ_{EP}-term that really executes the sending of the parameters, and makes the answer to the original object that is waiting in a *Wait(l,l')* instruction. In a local call, we do not have an answer because the purpose of a local invocation is execution, and not reply. This is the reason why the λ_{EP}-term, which results after the evaluation, is the sending of the expression. Therefore, the translation of a local call of one evaluated parameter method is:

$$[M(\beta_1)](m, l) \equiv \widehat{m}\ \beta_1 \circ \widehat{m}(\lambda_m x.\widehat{l}\ x)$$

A multiple evaluated arguments method is as follows:

$$[M(\beta_1, \ldots, \beta_n)](m, l) \equiv \widehat{m}\ \beta_1 \circ \ldots \circ \widehat{m}\ \beta_n \circ \widehat{m}(\lambda_m x.\widehat{l}\ x)$$

If we have not evaluated parameters and what we really have are expressions (as in the remote call), then the translation into LCEP is:

$$\begin{array}{lcl} [M(\beta_1, \ldots, \beta_{i-1}, E_i, \ldots, E_n)](m, l) & \equiv & (\widehat{m}\ \beta_1 \circ \ldots \circ \widehat{m}\ \beta_{i-1} \circ \lambda_{l_i} \beta_i.\widehat{m}\ \beta_i \circ \widehat{g_{i+1}}\ 0 \circ \\ & & \circ \ldots \circ \lambda_{l_n} \beta_n.\widehat{m}\ \beta_n \circ \widehat{m}(\lambda_m x.\widehat{l}\ x) \parallel [E_i](l_i) \parallel \\ & & \parallel \lambda_{g_{i+1}} x.[E_{i+1}](l_{i+1}) \parallel \ldots \parallel \lambda_{g_n} x.[E_n](l_n)) \end{array}$$

For a local declaration of variables with $Vdecs \equiv var\ \tilde{Y}$ into an expression, we can model into LCEP as:

$$[Vdecs\ in\ E](l, \ldots, \tilde{y}, \tilde{v}) \equiv [E](l, \ldots) \parallel [Var\ \tilde{Y}](\tilde{y}, \tilde{v})$$

where $[Var\ \tilde{Y}](\tilde{y}, \tilde{v}) \equiv [Var\ Y_1](y_1, v_1) \parallel \ldots \parallel [Var\ Y_v](y_v, v_v)$.

Now, let us see the translation into LCEP of the auxiliary expressions whose definitions were postponed:

$$
\begin{array}{lcl}
[Wait(\beta)](l,l') & \equiv & Wait(l,l') \\
[E \Rightarrow \beta](l,\ldots) & \equiv & [E](l',\ldots) \parallel Return(l,l',l'') \\
[E,\rho](l,\ldots) & \equiv & [E](l,\ldots) \parallel [\rho](\tilde{y},\tilde{v}) \\
[\rho](\tilde{y},\tilde{v}) & \equiv & Var(y_1,v_1) \parallel \ldots \parallel Var(y_n,v_n) \\
[nil](l) & \equiv & Null(l) \\
[Y := E](l,\ldots,y) & \equiv & [E](l',\ldots) \parallel \lambda_{l'} v'.(\widehat{y}(\widehat{u}\ v') \circ \lambda_y v.Null(l)) \\
[answer](a,l) & \equiv & \lambda_a x.x \circ Null(l)
\end{array}
$$

We express the assignment as the parallel composition of two λ_{EP}-terms: one which evaluates the expression and sends the result through the auxiliary channel l', and the other which receives the result of the expression from this channel and updates the variable Y by assigning the value of the expression.

With regard to the *answer* which replies to the request of remote invocation of methods, $[E_0!M(E_1,\ldots,E_n)]$, we can see that it accepts any request of any ready method, allowing to the replicator of the method to execute a copy of itself. It returns the result (to activate the client process which is in a *wait* instruction) and later returns a *null* through its own channel l to finish. Although only the last channel is really defined in the *answer*, all the others are sent from the client, as a λ_{EP}-term like $\widehat{m}\ \beta_1 \circ \ldots \circ \widehat{m}\ \beta_n \circ n\widehat{m}(\lambda_m x.\widehat{ret}\ x) \circ \lambda_{ret} x.\widehat{l'}\ x$ if there is no unevaluated expression.

The fundamental part of the λ_{EP}-term are the last two terms of the sequence. The second to the last term is a sender through the channel *ret* of the reply in a method execution. The last one takes that answer and re-sends it to the *Wait(l,l')* in the other process and it then becomes inactive.

The basic expression of the standard class *Bool* is as follows:

$$
\begin{array}{lcl}
[true](l,b_k) & \equiv & \widehat{b_k}(\widehat{t}(\lambda_{ret} x.\widehat{l'}\ x) \circ \lambda_l x.Value(l,x)) \\
[false](l,b_k) & \equiv & \widehat{b_k}(\widehat{f}(\lambda_{ret} x.\widehat{l'}\ x) \circ \lambda_l x.Value(l,x))
\end{array}
$$

And the standard class *Boolean* is:

$$
\begin{array}{lcl}
[Boolean](b_k) & \equiv & (BoolClass(b_k) \parallel Bool(b,b_k)) \\
BoolClass(b_k) & \equiv & !\ \lambda_{b_k} x.(x \parallel (\lambda_t x.(x \parallel \widehat{ret}\ \widehat{true}\ 0) \parallel \lambda_f x.(x \parallel \widehat{ret}\ \widehat{false}\ 0))) \\
Bool(b,b_k) & \equiv & (!\ BoolBody(b) \parallel !\ BoolVal(m_1) \parallel !\ BoolNot(m_2,m_1) \parallel \\
 & & \parallel !\ BoolAnd(m_3,m_1)) \\
BoolBody(b) & \equiv & \lambda_b x.x \\
BoolVal(m_1) & \equiv & \lambda_{m_1} x.(\widehat{m_1'}\ x \parallel \lambda_{m_1} y.(\lambda_{m_1'} z.(y \parallel \widehat{m_1}\ z))) \\
BoolNot(m_2,m_1) & \equiv & \lambda_{m_2} x.(\widehat{m_1}\ x \circ \widehat{m_1}(\lambda_{m_1} x.\widehat{m_2} x)) \circ \\
 & & \circ \lambda_{m_2} x.(x \parallel \lambda_{true} y.\widehat{m_2}\ \widehat{false}\ 0 \parallel \lambda_{false} y.\widehat{m_2}\ \widehat{true}\ 0) \\
BoolAnd(m_3,m_1) & \equiv & \lambda_{m_3} x.(\widehat{m_1}\ x \circ \widehat{m_1}(\lambda_{m_1} x.\widehat{m_3'} x)) \circ \lambda_{m_3} x.(x \parallel \lambda_{true} y.(\lambda_{m_3} x.(\widehat{m_1} x \circ \\
 & & \widehat{m_1}(\lambda_{m_1} x.\widehat{m_3'} x)) \circ \lambda_{m_3'} x.(x \parallel \lambda_{true} y.(\widehat{m_3}\ \widehat{true}\ 0) \parallel \\
 & & \parallel \lambda_{false} y.(\widehat{m_3}\ \widehat{false}\ 0)) \parallel \lambda_{false} y.(\lambda_{m_3} z.\widehat{m_3}\ \widehat{false}\ 0))
\end{array}
$$

To model the conditional instruction we have:

$$
\begin{array}{lcl}
[if\ E\ then\ S_1\ else\ S_2](l) & \equiv & BoolEval(l',l_1,l_2) \parallel [E](l') \parallel \lambda_{l_1} x.[S_1](l) \parallel \lambda_{l_2} x.[S_2](l) \\
BoolEval(l',l_1,l_2) & \equiv & \lambda_{l'} x.\widehat{b}(\widehat{m_1}\ x \circ (\widehat{m_1}(\lambda_{m_1} x.\widehat{ret}\ x))) \circ \\
 & & \circ(\lambda_{ret} x.x \parallel \lambda_{true} x.\widehat{l_1}\ 0 \parallel \lambda_{false} x.\widehat{l_2}\ 0)
\end{array}
$$

With regard to the sequence of sentences, if we have the sentence $[S_1;S_2]$, and we suppose that the sentence S_1 is an expression, then:

$$
[S_1;S_2](l,\ldots) \equiv [S_1](l',\ldots) \parallel \lambda_{l'} v.[S_2](l,\ldots)
$$

where v is a variable which appears in S_2 and is a bound variable.

```
CLASS PHILOSOPHERS
   VAR
      eat : boolean
      sig : boolean
      chopl : CHOPS
      chopr : CHOPS
   METHOD ToEat( ) : PHILOSOPHERS
      BEGIN
         WHILE not Eat DO
            sig := true;
            WHILE sig DO
               sig := chopl!Takechop( )
            OD;
            sig := chopr!Takechop();
            IF sig
               THEN eat := false;
                  chopl!Leavechop( )
               ELSE eat := true
            FI
         OD
         RESULT SELF
      END;

   METHOD Tothink( ) : PHILOSOPHERS
      BEGIN
         chopl!Leavechop( );
         chopr!Leavechop( );
         eat := false;
         RESULT SELF
      END;
   BODY
      eat := false;
      WHILE true DO
         Toeat( )
         Tothink( )
      OD
   YDOB

CLASS CHOPS
   VAR
      freechop : boolean
   METHOD Takechop ( ) : boolean
      BEGIN
         IF freechop
            THEN
               freechop := false;
               RETURN false
            ELSE RETURN true;
         FI
      END;
   METHOD Leavechop ( ) : CHOP
      BEGIN
         freechop := true;
         RESULT nil
      END;
   BODY
      freechop := true;
      WHILE true DO
         answer
      OD
   YDOB
```

Figure 1: The POOL philosophers problem.

On the contrary, if the sentence S_1 is not an expression, therefore, v is not bound in S_2, and the behaviour of the process is like $[S_1](l', \ldots) \circ [S_2](l, \ldots)$, with $l' : Null$. Knowing this and using the dual behaviour of expressions and non-expression sentences in the translation to LCEP of iterative instructions, we can find a new example of the use of the standard class *Bool* by *BoolEval*.

$$[while\ E\ do\ S](l, \ldots) \quad \equiv \quad Loop(l, l', \ldots) \parallel\ !\ \lambda_{l'} v.Loop(l, l', \ldots)$$

If S is an expression then c is a bound variable and therefore:

$$Loop(l, l', \ldots) \quad \equiv \quad (BoolEval(l'', l_1, l_2) \parallel [E](l'') \parallel \lambda_{l_1} x.[S](l', \ldots) \parallel \lambda_{l_2} x.Value(l, v))$$

If S is not an expression, then v is a free variable in S and then:

$$Loop(l, l', \ldots) \quad \equiv \quad (BoolEval(l'', l_1, l_2) \parallel [E](l'') \parallel \lambda_{l_1} x.[S](l', \ldots) \parallel \lambda_{l_2} x.Null(l))$$

5 Example of translation

To reflect the translation of a POOL program into LCEP, we use an easy variant of the well-known philosophers problem [5], in which we pay attention to the communication between the different objects. To model that behaviour in POOL, we consider two object classes: the *philosopher* class and the *chop* class.

We are basically interested in communication and method invocation. Therefore, we center our attention on the implementation of the classes, and we do not show *units* or specifications in our solution. We are not looking for an efficient solution to the concurrence problem. Therefore, we do not pay attention to aspects such as the non-deterministic choice between left and right chop (we necessarily take the left one before the right one). We can see the POOL example in Fig. 1.

From lack of space, we only show some parts of the translation following a similar scheme as in Section 4.

The translation of the philosophers'class declaration to LCEP is as follows:

$$\begin{array}{lcl} [Cphils] & \equiv & !\ \lambda_{phils} x.((x \parallel \widehat{ret}\ phil_i) \circ Newobject(phil_i)) \\ Newobject(phil_i) & \equiv & (BODYphil(phil_i)) \parallel [Vdecs](eat, sig, chopl, chopr) \parallel \\ & & \parallel [Mdecs](toeat, tothink)) \end{array}$$

An example of variable declaration is:

$$\begin{array}{lcl} Var(eat, v_b) & \equiv & \lambda_{eat} x_1.(x_1 \parallel \lambda_r x.(\widehat{eat}\ v_b \circ Var(eat, v_b)) \parallel \\ & & \parallel \lambda_u v'.(\lambda_{eat} v_{b'} \circ Var(eat, v_{b'}))) \end{array}$$

The invocation of the method *takechop* of an object *chop* from a *philosopher* is modelled in LCEP as:

$$\begin{array}{lcl}
[chopl\ !\ takechop(\ )](l, chopl, tchop) & \equiv & \widehat{chopl}(\widehat{tchop}(\lambda_{tchop} x.\widehat{ret}\ x) \circ \lambda_{ret} x.\widehat{l'}\ x) \circ Wait(l, l') \\
[chopr\ !\ takechop(\ )](l, chopr, tchop) & \equiv & \widehat{chopr}(\widehat{tchop}(\lambda_{tchop} x.\widehat{ret}\ x) \circ \lambda_{ret} x.\widehat{l'}\ x) \circ Wait(l, l')
\end{array}$$

Furthermore, the definition of the method *takechop( )* is:

$$\begin{array}{lcl}
[Mtakechopdec](tchop) & \equiv & !\ Mtakechop(tchop) \\
Mtakechop(tchop) & \equiv & Mhtakechop(tchop) \parallel \lambda_g x.[E_takechop](tchop) \\
Mhtakechop(tchop) & \equiv & \lambda_{tchop} x.(x \parallel \widehat{g}\ 0)
\end{array}$$

An example of a local call appears at the method *toeat( )* in a philosopher body:

$$[toeat()](toeat, l') \equiv \widehat{toeat}(\lambda_{toeat} x.\widehat{l'}\ x)$$

And the declaration of method *toeat( )* is:

$$\begin{array}{lcl}
[Mtoeatdec](toeat) & \equiv & !\ Mtoeat(toeat) \\
Mtoeat(toeat) & \equiv & Mhtoeat(toeat) \parallel \lambda_g x.[E_toeat](toeat) \\
Mhtoeat(toeat) & \equiv & \lambda_{toeat} x.(x \parallel \widehat{g}\ 0)
\end{array}$$

Our example contains a conditional instruction into the body of method *toeat( )* in the class *chop*:

IF sig
  THEN eat := false;
    chopl ! Leavechop()
  ELSE eat := true
FI

Therefore:

$$\begin{array}{lcl}
[if\ SIG\ then\ S_1\ else\ S_2](l) & \equiv & (BoolEval(l', l_1, l_2) \parallel (\widehat{sig}(\widehat{r}\ 0) \circ \lambda_{sig} v.Value(l', v)) \parallel \\
 & & \parallel \lambda_{l_1} x.[S_1](l) \parallel \lambda_{l_2} x.[S_2](l)
\end{array}$$

The following is a practical example of the translation of the loop into the body of object class *philosopher*:

WHILE true
  DO
    Toeat( )
    Tothink( )
  OD

Therefore:

$$\begin{array}{lcl}
While_0(l) & \equiv & [while\ true\ do\ S_3](l) \equiv Loop_0(l, l') \parallel !\ \lambda_{l'} c.Loop_0(l, l') \\
Loop_0(l, l') & \equiv & (BoolEval(l'', l_1, l_2) \parallel [true](l'', b_k) \parallel \lambda_{l_1} x.[S_3](l') \parallel \lambda_{l_2} x.null(l))
\end{array}$$

S_3 is the sequential composition of procedures *Toeat()* and *Tothink()*, which can be translated as:

$$\begin{array}{lcl}
[S_3](l') & \equiv & [toeat(\ )](toeat, l') \parallel \lambda_{l'} v.[tothink(\ )](tothink, l) \\
[toeat(\ )](toeat, l') & \equiv & \widehat{toeat}\ \lambda_{toeat} x.\widehat{l'}\ x) \\
[tothink(\ )](tothink, l) & \equiv & \widehat{tothink}\ \lambda_{tothink} x.\widehat{l'}\ x)
\end{array}$$

where v is a free variable in $[tothink(\ )](tothink, l)$.

6 Conclusions and future work

An extension of LCEP has been presented. Under a process perspective, this extension can model a simple version of a parallel object-oriented language, LCEPOO. The phrase-by-phrase translation process has been made by modelling object definition, management, communication, and evolution features. The object activities are synchronized by sending messages which contain references to other system objects. Among the characteristics of LCEP, we can not find the sending of communicating channels. The absence of this feature has not been an obstacle for the modelling of the behaviour of the chosen POOL variant. We have incorporated the possibility of sending variables (channels) only for the modelling of the object generation.

As we have shown, LCEP is an effective tool which has sucessfully been used to model parallel and concurrent processes as well as to model object-oriented language features. A translator system from a high level language (ALEPH) into LCEP is already available [4]. By transferring the proposal extension to that system, we can obtain an explicitly parallel high-level language with object-oriented features.

References

1. H. Aït-Kaci and J. Garrigue. Label-Selective λ-Calculus: Syntax and Confluence. In *Proc. of the 13th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1993.
2. P. America, J. de Bakker, J. Kok, and J. Rutten. Operational Semantics of a Parallel Object-Oriented language. *In Proc. of the 13^{th} Symposium on Principles of Programming Languages*, pages 194–208, 1986.
3. P. America, J. de Bakker, J. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. In *Information and Computation*, volume 83, pages 152–205. 1989.
4. L. Climent, M.L. Llorens, and J. Oliver. LCEP as an Abstract Machine. Technical Report DSIC-II/38/97, UPV, 1997.
5. E.W. Dijkstra. Cooperating Sequential Processes. In F. Genus, editor, *Programming Languages*, pages 43–112. Academic Press, London, 1968.
6. S. Lucas and J. Oliver. Parallel label-selective λ-calculus (LCEP). In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proc. of 1994 Joint Conference on Declarative Programming GULP-PRODE'94*, pages 125–139, 1994.
7. R. Milner. A Calculus of Communicating Systems. volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.
8. R. Milner. The polyadic π-calculus: A tutorial. In F.L. Brauer, W. Bauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specifications*. Springer-Verlag, Berlin, 1993.
9. J. Oliver. *Extensions of the λ-calculus for Modelling Concurrent Processes*. PhD thesis, DSIC (UPV), 1996.
10. F. Vaandrager. A process algebra semantics in POOL. *Applications of Process Algebra, Tracts in Theoretical Computer Science*, 17:173–236, 1990.
11. D. Walker. Objects in the π-calculus. In *Information and Computation*, volume 116, pages 253–271. 1995.

Observation equivalences for type and implementation inheritances

Claudia Balzarotti, Fiorella De Cindio, Lucia Pomello

Dipartimento di Scienze dell'Informazione, Università di Milano
via Comelico 39 - 20135 Milano (Italy)
e-mail: decindio@dsi.unimi.it, pomello@dsi.unimi.it

EXTENDED ABSTRACT

The main entity of the object-oriented languages is the *object*, which is generally defined by a class. A fundamental relation between classes is *inheritance*, that originally indicated a relation *is-a* among classes. On the other hand, the success of object-oriented programming languages is due to the use of inheritance for *code-reuse*. As shown in [Ame89], inheritance is often used in these two different ways. Our aim is studying the inheritance relation in concurrent object-oriented languages, formalising these aspects.

The integration of object-orientation with concurrency, yielding systems of active objects modelled as concurrent processes, has given the possibility of using theoretical tools, such as the notions of observation equivalences, for the study of inheritance and subtype relations. In particular, the place-transition duality in Petri nets [BRR87] allows the definition of two types of equivalences between nets, based on action observation and on state observation [PRS92]. They have been used for defining the semantics of inheritance relations. Among the various proposals in the literature (see for example [NP92], [AD95,96]), we take into consideration those proposed by Nierstrasz [Nie93] and van der Aalst and Basten [AB97] for what concerns action observability, while we consider the approach of CLOWN [BCD97] for what concerns the state observability.

Nierstrasz's hypothesis is that the sequences of requests that an object can accept constitute a regular language. Moreover, Nierstrasz defines a preorder based on *failures*, the *Request Substitutability* preorder (RS-preorder, $\leq_{RS}$), and considers this preorder as the semantics of the subtyping relation according to the Wegner and Zdonik's substitutability principle [WZ88]. Van der Aalst and Basten introduce preorders too, based on the *branching bisimulation* equivalence and two operators, the *encapsulation* operator δ_H and the *abstraction* operator τ_I, that, respectively, remove and label as not observable the transitions corresponding to the methods of the subclass not inherited from the superclass. CLOWN uses an inheritance relation based on the *State Transformation* preorder (ST-preorder, $\leq_{ST}$) [PS91], that compares systems with respect to their state space. Preorders are better suited than equivalence notions for modelling the behaviour extension.

We consider Elementary Net (EN) systems [BRR87], the basic class of Petri Nets. We denote as LEN systems the labelled EN systems in which the actions are observable. In the following, we discuss through examples the previous approaches.

Example 1

This example is taken from Nierstrasz's paper [Nie93]. In figure 1, the class VAR, that represents a variable, is subclass of the class BUF, that represents a one place buffer. Rephrasing Nierstrasz from CCS to Petri nets, the RS-preorder between the LEN system associated to the class VAR and the LEN system associated to the class BUF is satisfied, i.e. $\sum_{VAR} \leq_{RS} \sum_{BUF}$.

However, we believe that this example is somehow misleading: in fact a variable is not a buffer, but a buffer can be implemented through a variable.

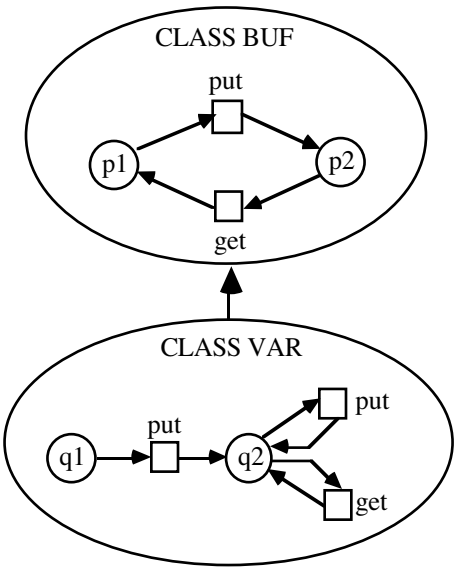


Figure 1

Example 2

This example is taken from [BCD97]. Figure 2 shows a multiple inheritance hierarchy between printers. In CLOWN, this hierarchy is studied through the ST-preorder, which considers the state (place) observation. however, the exhibited behaviour of objects that are instances of these classes, is better captured by observation of actions, i.e. of transitions. Therefore we would like to build the same hierarchy by using notions based on action observation.

Therefore our work aims at giving the semantic characterisation of two forms of inheritance in concurrent context. According to the terminology used in [CHC90], we use the term *type inheritance* to indicate the subtyping, i.e. the *is-a* relation between classes, and the term *implementation inheritance* to denote the code-reuse. In the first case, we study the external observable behaviour of an object, while in the second case we study the internal structure of an object. Therefore, we base the first preorder on *failure*, since *bisimulation* distinguishes systems also with respect to some aspect of internal structure, while in the second case we consider ST-preorder.

While this latter does not require modifications w.r.t. the original definition, we need to integrate the two approaches by Nierstrasz and van der Aalst and Basten, since the notions defined by them doesn't deal with all cases where there exists the relation *is-a*

between a subclass and a superclass. To this purpose we define a further operator, namely the *renaming* operator $\rho_{R,S}$, and new preorders based on action observation.

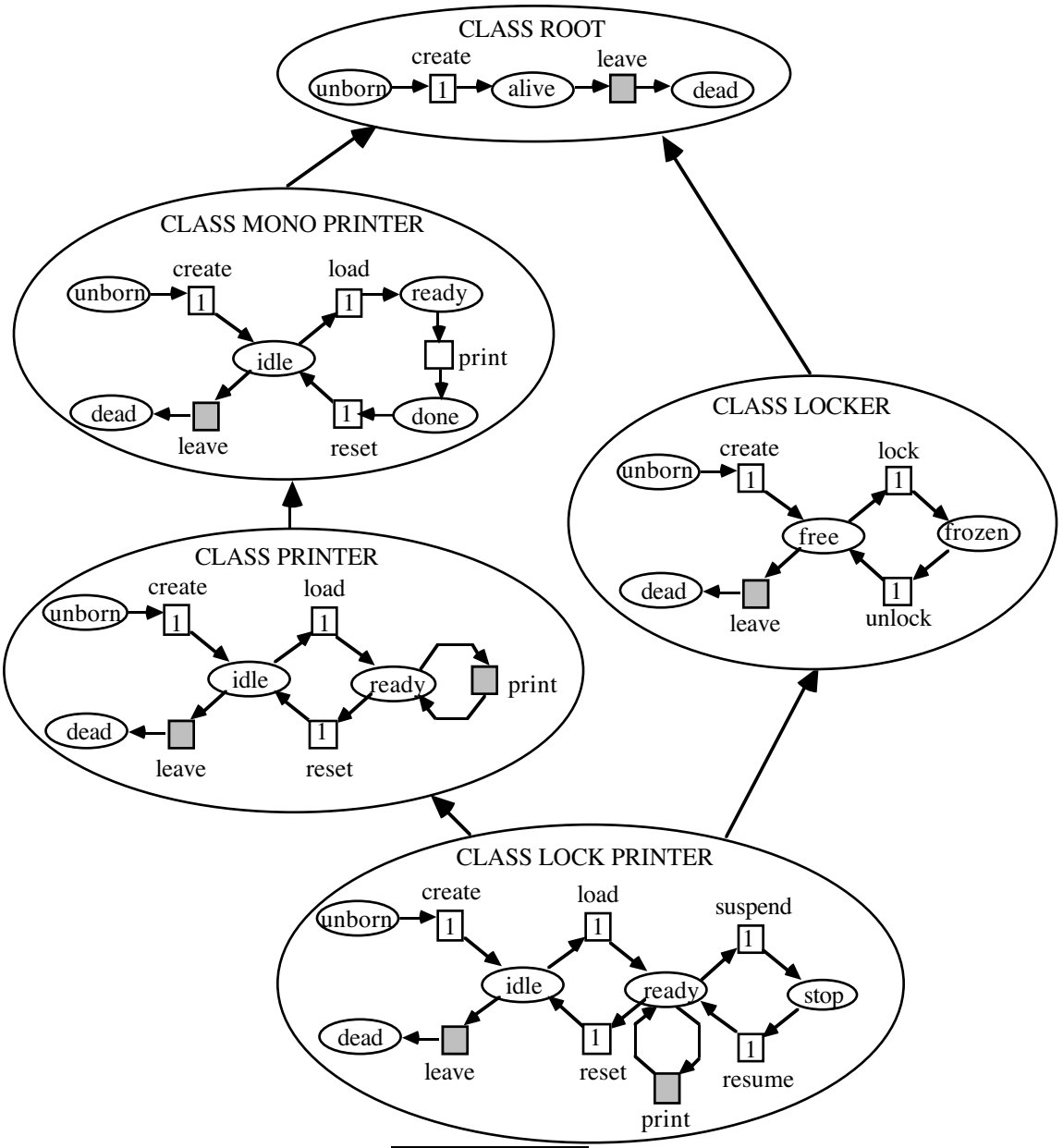


Figure 2

Formally, we associate to [redacted] system $\sum = (B,E,F,c_{in},h)$, where (B,E,F) is an Elementary Net, $c_{in} \subseteq B$ is the initial case, $h: E \to L$ is the labelling function that associates to each event a class method belonging to L. For each $R \subseteq L$ and for each set of labels S such that $S \cap L = \emptyset$ and $|S| = |R|$, we define a bijective function $s : R \to S$, that maps each [redacted]. Let L' be a superset of S containing L - R, i.e. $S \subseteq L'$ and $L - R \subseteq L'$. On the basis of the function s we define the *renaming* function: $r : L \to L'$ that maps each label into a new label in the following way:

(1) $r(x) = x$ if $x \notin R$
(2) $r(x) = y$ if $x \in R$, $y \in S$ and $s(x) = y$.
This function r is injective, since the function s is bijective.

We define the *renaming* operator $\rho_{R,S}$ in the following way:
$\rho_{R,S}(\sum) = \sum'$ such that $\sum' = (B,E,F,c_{in},h')$, where $h' = r \circ h$.

Now we can give the definitions of four preorders based on the Nierstrasz's RS-preorder and on the *encapsulation*, *abstraction* and *renaming* operators. In the following, $\sum_A$ and $\sum_B$ are LEN systems associated to a class A and a class B, while L_A and L_B are the sets of event labels associated respectively to $\sum_A$ and $\sum_B$ and corresponding to the methods of the classes A and B.

The first preorder, the strong substitutability, is equivalent to the RS-preoder.

Definition 1

$\sum_B$ is less or equal to $\sum_A$ w.r.t. *strong substitutability*, denoted $\sum_B \leq_{SF} \sum_A$, if and only if there exists a set $H _ L_B$ such that: $\delta_H(\sum_B) \leq_{RS} \sum_A$.

If $\sum_B \leq_{SF} \sum_A$ then an object of class A can be substituted for an object of class B and the environment will not be able to notice the difference, i.e.: if an object of class A may accept after a request sequence w another request a, then an object of class B must accept the request a after the request sequence w, whatever is the reached state.

The new methods added in class B are considered as not available through the encapsulation operator δ_H.

The second preorder, the strong substitutability with renaming, is less restrictive than strong substitutability.

Definition 2

$\sum_B$ is less or equal to $\sum_A$ w.r.t. *strong substitutability with renaming*, denoted $\sum_B \leq_{SFR} \sum_A$, if and only if there exist a set $H _ L_B$, a set $R _ L_A$ and a set $S _ L_B$ such that: $S \cap L_A = \varnothing$ and $\delta_H(\sum_B) \leq_{RS} \rho_{R,S}(\sum_A)$.

If $\sum_B \leq_{SFR} \sum_A$, then an object of class A can be substituted for an object of class B but the environment must make allowance for class A methods that are renamed in class B by the renaming operator $\rho_{R,S}$.

The third preorder, the weak substitutability, is another extension of the strong substitutability.

Definition 3

$\sum_B$ is less or equal to $\sum_A$ w.r.t. *weak substitutability*, denoted $\sum_B \leq_{SD} \sum_A$, if and only if there exist a set $H _ L_B$ and a set $I _ L_B$ such that :
$I \cap H = \varnothing$ and $\tau_I \circ \delta_H(\sum_B) \leq_{RS} \sum_A$.

If $\sum_B \leq_{SD} \sum_A$, then an object of class A can be substituted for an object of class B and the environment will not be able to notice the difference since the new methods added in class B are either considered unobservable, through the abstraction operator τ_I , or considered unavailable, through the encapsulation operator δ_H.

[redacted] preorder, the weak substitutability with renaming, is the weaker preorder.

Definition 4

$\sum_B$ is less or equal to $\sum_A$ w.r.t. *weak substitutability with renaming* ($\sum_B \leq_{SDR} \sum_A$) if there exist a set $H \subseteq L_B$, a set $I \subseteq L_B$, a set $R \subseteq L_A$ and a set $S \subseteq L_B$ such that: $I \cap H = \emptyset$, $S \cap L_A = \emptyset$ and $\tau_I \circ \delta_H(\sum_B) \leq_{RS} \rho_{R,S}(\sum_A)$.

If $\sum_B \leq_{SDR} \sum_A$, then an object of class A can be substituted for an object of class B and the environment must consider the renamed methods, while the new methods are either considered unobservable, through the abstraction operator τ_I , or considered unavailable, through the encapsulation operator δ_H.

The formal proofs that these definitions are sound, i.e.: that the relationships so defined are indeed preorders, are in [Bal98].

Figure 3 shows the relations between preorders.

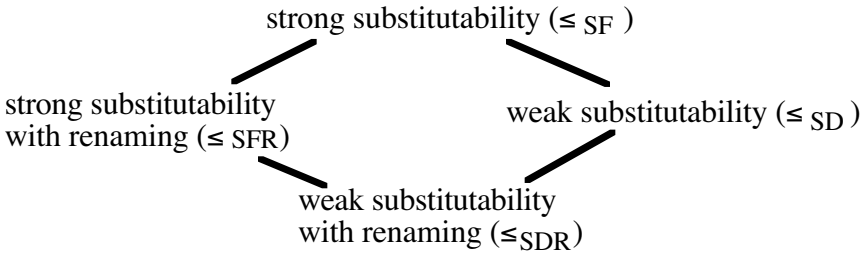


Figure 3

Now we can formalise the notion of type inheritance and implementation inheritance.

Definition 5

Let $\sum_A$ and $\sum_B$ be LEN systems associated to a class A and a class B.
Then B is *subclass* of A with respect to *type inheritance* if and only if $\sum_B \leq_{SDR} \sum_A$.

The notion of implementation inheritance is based on the State Transformation preorder ($\leq_{ST}$) [PS91], which compares systems by observing local states and is such that $\sum_A \leq_{ST} \sum_B$ if and only if the state space of $\sum_A$ is a substructure of the state space of $\sum_B$ such that for any observable local state transformation in $\sum_A$ there is a corresponding observable local state transformation in $\sum_B$.

Definition 6

Let $\sum_A$ and $\sum_B$ be EN systems, associated to a class A and a class B, in which some appropriate places are observable.
Then B is *subclass* of A with respect to *implementation inheritance* if and only if $\sum_A \leq_{ST} \sum_B$.

These definitions solve Example 1 and 2 above, as discussed in the following.

Example 3

In figure 4, the class BUF represents a one place buffer, while the class VAR represents a variable. The class V-BUF represents a one place buffer implemented through a variable, which inherits from both classes BUF and VAR. The class V-BUF is subclass of BUF with respect to *type inheritance*, as it preserves the requests that can be accepted from clients: $\sum_{\text{V-BUF}} \leq_{\text{SDR}} \sum_{\text{BUF}}$. It is subclass of VAR with respect to *implementation inheritance*: $\sum_{\text{BUF}} \leq_{\text{ST}} \sum_{\text{V-BUF}}$, as the VAR methods "assign" and "get_item" are used for implementing the buffer's "put" and "get" methods. This captures the intuition that one cell buffer can be implemented with variables.

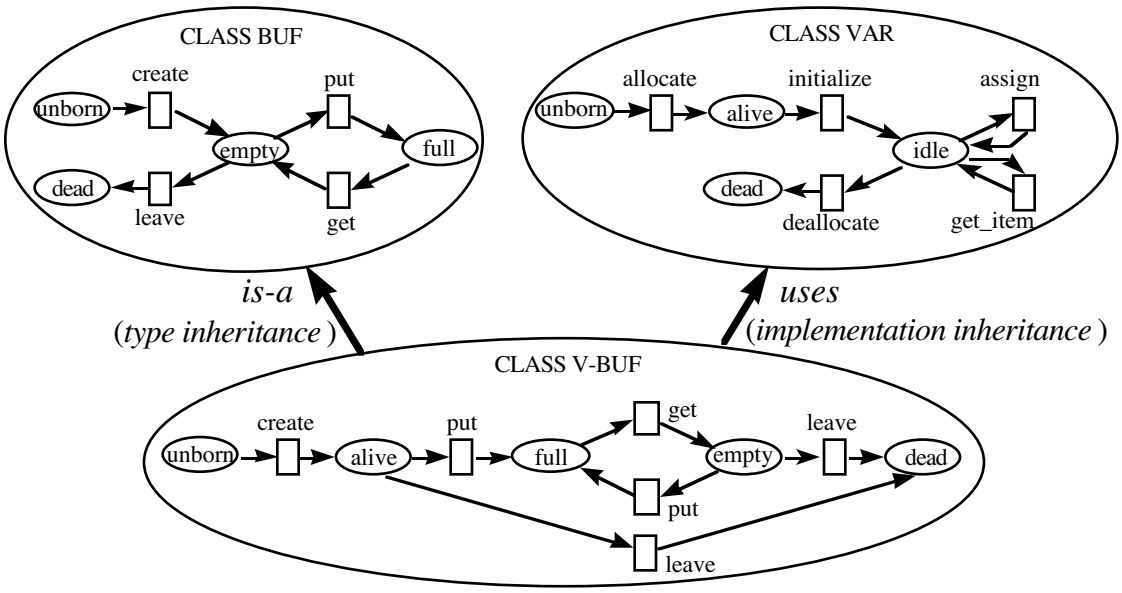


Figure 4

Example 4

The classes of the printer hierarchy in figure 2 satisfy the type inheritance. That is, the hierarchy is now captured observing methods instead of states, as it was in Example 2.

The open problem is now to remove Nierstrasz's constraint, i.e.: to admit that sequences of requests, that an object can accept, constitute a non-regular language. In this case, the object behaviour would be described by an high-level net, with guards associated to transitions. (In CLOWN, e.g., the class semantics is described by a modular algebraic high level net, which integrates the Superposed Automata nets with the OBJ language). The problem is not easy since the notions of equivalence and preorder between nets are defined in terms of EN or PT systems, i.e. low level systems, while there aren't equivalences defined for high level models like OBJSA nets.

References

[AB97] W.M.P. van der Aalst, T. Basten, *Life-Cycle Inheritance. A Petri-Net-Based Approach*, in P. Azéma G. Balbo (eds.), Proc. 18th Int. Conf. on Applications and Theory of Petri Nets, LNCS 1248, Springer Verlag, 1997.

[AD95,96] G. Agha, F. De Cindio, eds., Proc. of the Workshop on Object-Oriented Programming and Models of Concurrency, 1995 and 1996.

[Ame89] P. America, *Issues in the Design of a Parallel Object-Oriented Language*, in "Formal Aspects of Computing", vol. 1, pp. 336-411, 1989.

[Bal98] C. Balzarotti, *Equivalenze all'osservazione per la caratterizzazione semantica della ereditarietà in linguaggi a oggetti concorrenti*, Tesi di laurea, Università degli Studi di Milano, 1998.

[BCD97] E. Battiston, A. Chizzoni, F. De Cindio, *CLOWN as a Testbed for Concurrent Object-Oriented Concepts*, G. Agha, F. De Cindio, G. Rozenberg (eds.), Advences in Petri Nets, Springer Verlag (to appear).

[BRR87] W. Brauer, W. Reisig, G. Rozenberg, *Petri Nets. Central Models and their Properties. Advances in Petri Nets 1986, part 1. Proceedings of an Advanced Course*. LNCS 254, Berlin, Springer Verlag, 1987.

[CHC90] W.R. Cook, W.L. Hill, P.S. Canning, *Inheritance Is Not Subtyping*, in Proc. of the ACM Symp. on Principles of Programming Languages, pp. 125-135, 1990.

[Nie93] O. Nierstrasz, *Regular Types for Active Objects*, in ACM Sigplan Notices, 28(10), Proceedings of the 8th annual conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'93, Washington DC, pp. 1-15, 1993.

[NP92] O. Nierstrasz, M. Papathomas, eds., *Object based Concurrency and Reuse*, Workshop W6, in Proc. 6th European Conf. on Object-Oriented Programming, Springer Verlag, 1992.

[PRS92] L. Pomello, G. Rosenberg, C. Simone, *A Survey of Equivalence Notions for Net Based Systems*, in G. Rosenberg (ed.), "Advanced in Petri Nets", LNCS 609, Springer-Verlag, pp. 410-472, 1992.

[PS91] L. Pomello, C. Simone, *A State Transformation Preorder over a class of EN Systems*, in G. Rozenberg (ed.) APN'90, LNCS 483, pp.436-456, 1991.

[WZ88] P. Wegner, S. B. Zdonik, *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like*, in S. Gjessing and K. Nygaard (eds.), Proc.ECOOP '88, LNCS 322, Springer Verlag, pp.55-77, 1988.

This research has been supported by MURST.

Towards an Algebra of Dynamic Object Types

António Ravara * Pedro Resende * Vasco T. Vasconcelos †

Abstract

We propose an algebra of object types that characterises the semantics of concurrent objects in a process calculus setting where the communication is asynchronous. The types are non-uniform, and provide an internal (and synchronous) view of the objects that inhabit them. These ideas, along with the algebraic laws, are based on a notion of bisimulation that is unlike other notions in the literature.

1 Introduction

Non-uniform types for concurrent objects constitute the object of study of several authors [7, 2, 3, 6, 8]. The aim is to build type systems capable of ensuring more than the usual safety properties (such as subject-reduction); for instance, the absence of some deadlocks. These types reflect a dependency of the interface of an object upon its internal state, conveying information about some dynamic properties of objects.

In a process calculus setting such as TyCO [10], processes denote the behaviour of a community of interacting objects, where each object has a location identified by a name. Each process determines an assignment of types to names reflecting a discipline for communication. The usual types-as-records paradigm [11] gives each name a static type that contains information about all the methods of the object, regardless of whether they are enabled or not. Hereby we propose an algebra of object types, where each type is essentially a collection of enabled methods, and it is dynamic in the sense that the execution of a method can change this collection, i.e. the type. Therefore, the type of an object can also be seen as a partial representation of its behaviour.

We assume that objects communicate via asynchronous message-passing; nevertheless, types, as defined in this paper, essentially correspond to a notion of object behaviour as it would be perceived by an internal observer located within an object (the object's private gnome). This observer can see methods being invoked and it can detect whether the object is blocked, even though its methods may be internally enabled. Hence, this notion of behaviour is synchronous, as essentially the gnome can detect refusals of methods. The action of unblocking an object, denoted by υ, cannot be observed by the gnome because it corresponds to an invocation of some method in another object. Thus, this action is similar to Milner's τ [4], because it is hidden, but it is external, rather than an internal action.

In this paper we define a structural operational semantics for the algebra of object types, both for finite and infinite types. We also introduce two behavioural equivalences based on notions of bisimulation that characterise the referred aspects of an object, and show that the two coincide, at least in the case of finite types.

*Computer Science Section, Department of Mathematics, Instituto Superior Técnico. Lisbon, Portugal. Email: {amar,pmr}@math.ist.utl.pt

†Department of Computer Science, Faculty of Sciences, University of Lisbon, Portugal. Email: vv@di.fc.ul.pt

2 Algebra of Finite Object Types

We start by presenting the algebra of finite object types. Objects are records of methods, and we can represent unavailable (blocked) objects.

Assume a countable set of *method names* l, m, n, possibly subscripted or primed.

DEFINITION 2.1. The set $\mathcal{O}$ of *finite types of objects* is given by the following grammar.

$$\alpha \quad ::= \quad \mathbf{0} \quad | \quad \sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i \quad | \quad \biguplus_{j \in J} \alpha_j$$

where I and J are non-empty finite indexing sets, with $\forall_{i,j \in I}\ i \neq j \Rightarrow l_i \neq l_j$, and, $\widetilde{\alpha}$ is a finite sequence of types.

We call a term such as $l(\widetilde{\alpha}).\alpha$ a *method type*. It corresponds to a method with name l and parameters of type $\widetilde{\alpha}$, which behaves as prescribed by α. The type composition operator sum ("$\sum$") puts various method types together to form a type of an object that offers the corresponding methods. The term $\mathbf{0}$ is the empty type. The disjoint union $\biguplus_{j \in J} \alpha_j$ is the type of a blocked object that will behave later according to one of the types α_j, after being released.

NOTATION 2.2. 1. We abbreviate the type $l(\widetilde{\alpha}).\mathbf{0}$ to $l(\widetilde{\alpha})$, and $l()$ to l.

2. We assume the following abbreviations:
   (a) $l(\widetilde{\alpha}).\alpha$ is $\sum_{i \in \{1\}} l_i(\widetilde{\alpha}_i).\alpha_i$;
   (b) $\uplus \alpha$ is $\biguplus_{i \in \{1\}} \alpha_i$;
   (c) $l_1(\widetilde{\alpha}_1).\alpha_1 + l_2(\widetilde{\alpha}_2).\alpha_2$ is $\sum_{i \in \{1,2\}} l_i(\widetilde{\alpha}_i).\alpha_i$;
   (d) $\alpha_1 \uplus \alpha_2$ is $\biguplus_{i \in \{1,2\}} \alpha_i$.

We define a structural operational semantics for the finite types of objects via a labelled transition relation.

DEFINITION 2.3. The set of *labels* is given by the following grammar.

$$\pi \quad ::= \quad \upsilon \quad | \quad l(\widetilde{\alpha})$$

The label υ denotes a silent transition that releasees a blocked object; a label $l(\widetilde{\alpha})$ denotes a method invocation.

DEFINITION 2.4. The *labelled transition relation* is inductively defined by the following rules.

$$\textsc{Act} \quad \sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i \xrightarrow{l_j(\widetilde{\alpha}_j)} \alpha_j \quad (j \in I) \qquad\qquad \textsc{Union} \quad \biguplus_{i \in I} \alpha_i \xrightarrow{\upsilon} \alpha_j \ (j \in I)$$

The axiom ACT gives the basic transition: the invocation of a method with name l and parameters of type $\widetilde{\alpha}$ results in the type of the body of that method. The axiom UNION captures a side effect: $\biguplus_{i \in I} \alpha_i$ is blocked, and one of its behaviours can only become available after an unblocking action occurs.

NOTATION 2.5. Let $\Longrightarrow$ denote $\xrightarrow{\upsilon}{}^*$, and $\overset{\upsilon}{\Longrightarrow}$ denote $\xrightarrow{\upsilon}{}^+$; furthermore, we write $\alpha \overset{\pi}{\not\longrightarrow}$ when $\alpha \xrightarrow{\pi}$ does not hold.

We want two object types to be equivalent if they have equivalent method types and if after each transition they continue to be equivalent, in a bisimulation style. Furthermore, from the point of view of each object type, transitions of other object types can be regarded as hidden transitions, which would suggest weak bisimulation as the right notion of equivalence for our object types, with υ playing the role of Milner's τ. However, we want our types to distinguish an object that immediately makes available a method from one that makes it available only after being unblocked by another object. This is because, although υ is supposed to be unobservable, we assume that from the point of view of an internal observer (the object's gnome) it is detectable that the object is blocked. Hence, we would expect $\uplus l$ to be different from l, but $\uplus l$ and $\uplus \uplus l$ to be equivalent; in both, all the internal observer can see is that the object is blocked, and after being released it can eventually execute the method l. We also want to distinguish $l.\uplus m$ from $l.m$, on the grounds that for the latter a blocking after l cannot be observed. This also discards Milner's observational congruence [4] and rooted bisimulation [1] as possible candidates for object equivalence. A notion such as progressing bisimulation [5] is however too strong because it would distinguish $\uplus\alpha$ from $\uplus \uplus \alpha$.

These considerations lead to the choice of a notion of equivalence that essentially strengthens weak bisimulation by requiring that if α and β are bisimilar and α offers a particular method then also β offers that method.

DEFINITION 2.6. *Bisimilarity on object types.*

1. A symmetric binary relation $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{O}$ is an *object bisimulation* if whenever $\alpha\mathcal{R}\beta$ then
   (a) $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ implies $\exists \beta', \widetilde{\beta}, \gamma$ $(\beta \xrightarrow{l(\widetilde{\beta})} \gamma \Longrightarrow \beta'$ and $\alpha'\mathcal{R}\beta'$ and $\widetilde{\alpha}\mathcal{R}\widetilde{\beta})$[1];
   (b) $\alpha \xrightarrow{\upsilon} \alpha'$ implies $\exists \beta'$ $(\beta \Longrightarrow \beta'$ and $\alpha'\mathcal{R}\beta')$;
2. Two types α and β are *object-bisimilar*, or simply *bisimilar*, and we write $\alpha \approx_o \beta$, if there is an object bisimulation $\mathcal{R}$ such that $\alpha\mathcal{R}\beta$.

The usual properties of bisimilarities hold, namely $\approx_o$ is an equivalence relation, and $\alpha \approx_o \beta$ holds if and only if

1. $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ implies $\exists \beta', \widetilde{\beta}, \gamma$ $(\beta \xrightarrow{l(\widetilde{\beta})} \gamma \Longrightarrow \beta'$ and $\alpha' \approx_o \beta'$ and $\widetilde{\alpha} \approx_o \widetilde{\beta})$;
2. $\alpha \xrightarrow{\upsilon} \alpha'$ implies $\exists \beta'$ $(\beta \Longrightarrow \beta'$ and $\alpha' \approx_o \beta')$;

We can strengthen this notion of equivalence even more by dropping another double arrow, as follows.

DEFINITION 2.7. *Strict bisimilarity on object types.*

1. A symmetric binary relation $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{O}$ is a *strict object bisimulation* if whenever $\alpha\mathcal{R}\beta$ then
   (a) $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ implies $\exists \beta', \widetilde{\beta}$ $(\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha'\mathcal{R}\beta'$ and $\widetilde{\alpha}\mathcal{R}\widetilde{\beta})$;
   (b) $\alpha \xrightarrow{\upsilon} \alpha'$ implies $\exists \beta'$ $(\beta \Longrightarrow \beta'$ and $\alpha'\mathcal{R}\beta')$;
2. Two types α and β are *strictly bisimilar*, and we write $\alpha \approx_s \beta$, if there is a strict object bisimulation $\mathcal{R}$ such that $\alpha\mathcal{R}\beta$.

Again, $\approx_s$ is an equivalence relation and $\alpha \approx_s \beta$ holds if and only if conditions 1(a) and 1(b) of the above definition hold with $\mathcal{R}$ replaced by $\approx_s$. Although for an arbitrary labelled transition system the two bisimilarity relations do not coincide, on our particular system they do, as the following result shows. This provides a sense in which our definition of object type equivalence is robust.

[1] Let $(\alpha_1 \cdots \alpha_k)\mathcal{R}(\beta_1 \cdots \beta_k) \stackrel{\text{def}}{=} \alpha_1\mathcal{R}\beta_1 \wedge \cdots \wedge \alpha_k\mathcal{R}\beta_k$.

THEOREM 2.8. Let $\alpha, \beta \in \mathcal{O}$. Then $\alpha \approx_o \beta$ if and only if $\alpha \approx_s \beta$.

Proof. The right to left implication is immediate. For the other implication we first remark that our labelled transition system satisfies the following conditions:

1. if $\alpha \xrightarrow{l(\widetilde{\alpha}_1)} \alpha_1$ and $\alpha \xrightarrow{l(\widetilde{\alpha}_2)} \alpha_2$ then $\alpha_1 = \alpha_2$ (label determinism);
2. no α can have both an v and an l transition, i.e., for all α either $\alpha \overset{v}{\not\to}$ or $\alpha \not\xrightarrow{l(\widetilde{\alpha})}$.

Let $\alpha \approx_o \beta$. We will show that

1. $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ implies $\exists \beta', \widetilde{\beta}$ ($\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha' \approx_o \beta'$ and $\widetilde{\alpha} \approx_o \widetilde{\beta}$),
2. $\alpha \xrightarrow{v} \alpha'$ implies $\exists \beta'$ ($\beta \Longrightarrow \beta'$ and $\alpha' \approx_o \beta'$);

that is, we will prove that $\approx_o$ is a strict object bisimulation, which will conclude the proof. The second condition is trivial, so let $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$. There exist $\widetilde{\beta}, \beta', \beta''$ such that $\beta \xrightarrow{l(\widetilde{\beta})} \beta' \Longrightarrow \beta''$, with $\widetilde{\alpha} \approx_o \widetilde{\beta}$ and $\alpha' \approx_o \beta''$. The condition $\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ in turn implies, together with label determinism, that $\alpha' \Longrightarrow \alpha''$ with $\alpha'' \approx_o \beta'$, for some α''. All we have to do now is prove that $\alpha' \approx_o \beta'$. If $\alpha' = \alpha''$ or $\beta' = \beta''$ this is immediate. Otherwise we have the following situation:

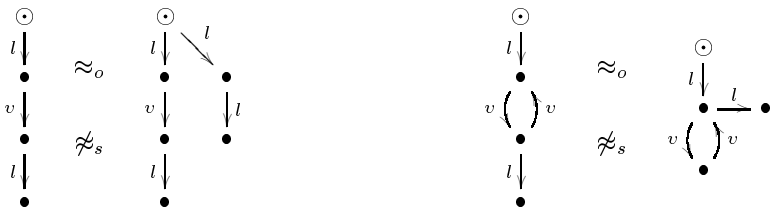
$$\begin{array}{ccc} \alpha' & \approx_o & \beta'' \\ {\scriptstyle v}\Vert\downarrow & & \uparrow\Vert{\scriptstyle v} \\ \alpha'' & \approx_o & \beta' \end{array}$$

In this case all the transitions from α' must be labelled with v; let then $\alpha' \xrightarrow{v} \alpha'''$; it follows that $\beta'' \Longrightarrow \beta'''$ for some β''' such that $\alpha''' \approx_o \beta'''$, hence also $\beta' \Longrightarrow \beta'''$. Similarly, β' can only do v, and if $\beta' \xrightarrow{v} \beta'''$ we can find α''' such that $\alpha' \Longrightarrow \alpha'''$ and $\alpha''' \approx_o \beta'''$, which concludes the proof. □

Therefore, in our type algebra the two equivalences coincide; henceforth we refer to both $\approx_o$ and $\approx_s$ as *object equivalence*, and write $\approx_o$.

The proof of the theorem relies on the fact that our transition system is deterministic on labels, and no state can have a transition labelled with l and another with v. If either of these conditions is violated the theorem no longer holds, as the following examples show:

EXAMPLE 2.9.



Naturally, the counterexamples above are not expressible in our language.

PROPOSITION 2.10. Object equivalence is a congruence relation.

Proof. Straightforward, since the sum is guarded. □

PROPOSITION 2.11 (ALGEBRAIC LAWS). 1. The operators "+" and "$\uplus$" are commutative; that is, for any permutation $\sigma : I \to I$ we have $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \approx_o \sum_{i\in I} l_{\sigma(i)}(\widetilde{\alpha}_{\sigma(i)}).\alpha_{\sigma(i)}$, and $\uplus_{i\in I}\alpha_i \approx_o \uplus_{i\in I}\alpha_{\sigma(i)}$;

2. the operator "$\uplus$" enjoys a weak form of associativity, $\uplus(\uplus_{i\in I}\alpha_i) \approx_o \uplus_{i\in I}\alpha_i$;

3. if $\alpha_1 \uplus \cdots \uplus \alpha_n \xrightarrow{\upsilon}{}^{+} \beta$ then $\beta \uplus \alpha_1 \uplus \cdots \uplus \alpha_n \approx_o \alpha_1 \uplus \cdots \uplus \alpha_n$.

Proof. The respective bisimulations are straightforward. □

Notice that law 3 is not really algebraic, but rather it gives us an infinity of laws that express a form of idempotence. For instance, we have

$$\begin{array}{rcl} \alpha_1 \uplus (\alpha_1 \uplus \cdots \uplus \alpha_n) & \approx_o & \alpha_1 \uplus \cdots \uplus \alpha_n \\ \alpha_1 \uplus ((\alpha_1 \uplus \cdots \uplus \alpha_n) \uplus \beta_1 \uplus \cdots \uplus \beta_m) & \approx_o & (\alpha_1 \uplus \cdots \uplus \alpha_n) \uplus \beta_1 \uplus \cdots \uplus \beta_m \\ & \vdots & \end{array}$$

A more thorough discussion of the algebraic laws will appear in the full version of this report.

REMARK 2.12. One can easily recognise that what corresponds to the law $\tau.\tau.\alpha = \tau.\alpha$ of process algebra, namely $\uplus\uplus\alpha \approx_o \uplus\alpha$, holds in this setting, since it is an instance of the weak associativity law presented above. Notice however that other laws like $\tau.\alpha = \alpha$ and $a.\tau.\alpha = a.\alpha$ do not hold. Also, $\uplus$ is not associative; e.g., $l\uplus(m\uplus n) \not\approx_o l\uplus m\uplus n$, which means that although υ is unobservable, sometimes it can be indirectly counted.

3 The Algebra of Object Types

Now we briefly discuss how the algebra of object types can be extended to deal with multiple objects located at the same name (with a parallel composition operator) and infinite types (with recursion).

DEFINITION 3.1. Assume a countable set of variables, denoted by t, disjoint from the set of labels. The set $\mathcal{O}$ of *types of objects* is defined by the following grammar.

$$\alpha \quad ::= \quad \mathbf{0} \quad | \quad \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \quad | \quad \biguplus_{j\in J} \alpha_j \quad | \quad \alpha \parallel \alpha \quad | \quad t \quad | \quad \mu t.\alpha$$

where I and J are non-empty finite index sets, with $\forall_{i,j\in I}\ i \neq j \Rightarrow l_i \neq l_j$, and $\widetilde{\alpha}$ is a finite sequence of types.

The parallel composition ("$\parallel$") of types denotes the existence of several objects located at the same name in parallel (interpreted as different copies of the same object, possibly several in different states).

DEFINITION 3.2. The *labelled transition relation* is defined by the rules of Definition 2.4 together with the following rules.

$$\text{RPAR} \quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\alpha \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta} \qquad \text{LPAR} \quad \frac{\beta \xrightarrow{\pi} \beta'}{\alpha \parallel \beta \xrightarrow{\pi} \alpha \parallel \beta'} \qquad \text{REC} \quad \frac{\alpha[\mu t.\alpha/t] \xrightarrow{\pi} \alpha'}{\mu t.\alpha \xrightarrow{\pi} \alpha'}$$

This transition system does not satisfy the two conditions that were used in proving the equality of the two notions of bisimilarity in the previous section, as the types $l\|l$ and $l\|\uplus l$ show. However, the counterexamples of Example 2.9 are still not expressible in our language; we are currently checking whether the two bisimilarities coincide in the presence of parallel composition and recursion. Apart from this, it is simple to verify that both $\approx_o$ and $\approx_s$ are congruences, and that standard algebraic laws hold. For instance, $\langle \mathcal{O}/\approx_{\{o,s\}}, \|, \mathbf{0}\rangle$ is a commutative monoid, and $\mu t.\alpha \approx_{\{o,s\}} \alpha[\mu t.\alpha/t]$.

4 Concluding remarks

We propose an algebraic treatment of non-uniform types for concurrent objects, with an operational semantics and a behavioural equivalence. A type characterises the semantics of an object in a concurrent setting with asynchronous message passing. It is an internal view of the object behaviour. Operationally, a type is a state transition system, where the basic transition is an object method execution. A silent (hidden) transition corresponds to the execution of a method of another object, and is not directly observable.

Further work includes the study of infinite processes and the search for a complete axiomatization of the algebra of object types. So far, candidate axiomatic systems tend to be cumbersome; we view this essentially as a consequence of the lack of associativity of $\uplus$. We also expect to apply the ideas described in this paper to the TyCO type system proposed in [9], and to relate the type algebra to a process calculus, for instance relating type equivalence to a process equivalence.

Acknowledgements

This work is partially supported by FCT, as well as by PRAXIS XXI Projects 2/2.1/MAT/262/94 SitCalc, 2/2.1/MAT/46/94 Escola, PCEX/P/MAT/46/96 ACL plus 2/2.1/TIT/1658/95 LogComp, and ESPRIT IV Working Groups 22704 ASPIRE and 23531 FIREworks.

References

[1] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51:129–176, 1987.

[2] G. Boudol. Typing the use of resources in a concurrent calculus. In *Asian Computing Science Conference*, volume LNCS 1345, pages 239–253. Springer-Verlag, 1997.

[3] J.-L. Colaço, M. Pantel, and P. Sallé. A set constraint-based analyses of actors. In *2nd IFIP conference on Formal Methods for Open Object-based Distributed Systems*, 1997.

[4] R. Milner. *Communication and Concurrency*. C. A. R. Hoare Series Editor – Prentice-Hall Int., 1989.

[5] U. Montanari and V. Sassone. Dynamic congruence vs. progressing bisimulation for CCS. *Fundamenta Informaticae*, 16 (2):171–199, 1992.

[6] E. Najm and A. Nimour. A calculus of object bindings. In *2nd IFIP conference on Formal Methods for Open Object-based Distributed Systems*, 1997.

[7] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[8] A. Ravara and V. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *3rd International Euro-Par Conference*, volume LNCS 1300, pages 554–561. Springer-Verlag, 1997.

[9] A. Ravara and V. Vasconcelos. A type algebra for concurrent objects. Research report, Department of Mathematics, Instituto Superior Técnico, Av. Rovisco Pais 1096 Lisboa, Portugal, 1998.

[10] V. Vasconcelos. *A Process-Calculus Approach to Typed Concurrent Objects*. PhD thesis, Departament of Computer Science, Keio University, Japan, 1994.

[11] V. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *1st International Symposium on Object Technologies for Advanced Software*, volume LNCS 742, pages 460–474. Springer-Verlag, 1993.

A Concurrent Object Calculus: Summary of the Operational Semantics. Extended Abstract

Andrew D. Gordon
Microsoft Research

Paul D. Hankin
University of Cambridge

A great deal of software is coded in terms of concurrent processes and objects. The purpose of our work is to develop a new formalism for expressing, typing, and reasoning about computations based on concurrent processes and objects.

Our concurrent object calculus $\mathbf{conc}\varsigma$ consists of Abadi and Cardelli's imperative object calculus $\mathbf{imp}\varsigma$ extended with primitives for parallel composition. Our work extends the analysis by Abadi and Cardelli of object-oriented features to concurrent languages. At the heart of their work is a series of type systems able to express a great variety of object-oriented idioms. Given $\mathbf{conc}\varsigma$, we may smoothly and soundly extend these type systems to accommodate concurrency.

There are by now many formalisms capable of encoding objects and concurrency. Our calculus supports Abadi and Cardelli's type systems, includes sequential composition of expressions. We describe the semantics of $\mathbf{conc}\varsigma$ directly without introducing auxiliary notions of stores, configurations or labelled transitions by means of a reduction relation and a structural congruence relation in the style of Milner's reduction semantics for the π-calculus.

This is an extended abstract of a full paper, available from the authors. The full paper includes examples, an extension of the calculus to include synchronisation, a collection of type systems, an alternative semantics for the calculus in terms of configurations as well as full definitions and proofs.

Concurrent Objects

We extend the imperative object calculus by adding names to objects, and adding parallel composition and name scoping operators from the π-calculus.

Syntax We assume there are disjoint infinite sets of *names*, *variables*, and *labels*. We let p, q, and r range over names. We let x, y, and z range over variables. We

let ℓ range over labels. We define the sets of *results*, *denotations*, and *terms* by the grammars:

Syntax of the concς-calculus

$u, v ::=$	results
$\quad x \mid p$	
$d ::=$	denotations
$\quad [\ell_i = \varsigma(x_i)b_i \, ^{i\in 1..n}]$	
$a, b, c ::=$	terms
$\quad u \mid p \mapsto d \mid u.\ell \mid u.\ell \Leftarrow \varsigma(x)b$	
$\quad clone(u) \mid let\ x{=}a\ in\ b \mid a \Rsh b \mid (\nu p)a$	

Semantics We may interpret a term of our object calculus either as a *process* or as an *expression*. A process is simply a concurrent computation. An expression is a concurrent computation that is expected to return a result. In fact, an expression may be regarded as a process, since we may always ignore any result that it returns.

A result u is an expression that immediately returns itself.

A denomination $p \mapsto [\ell_i = \varsigma(x_i)b_i \, ^{i\in 1..n}]$ is a process that confers the name p on the object $[\ell_i = \varsigma(x_i)b_i \, ^{i\in 1..n}]$. We say that the object $[\ell_i = \varsigma(x_i)b_i \, ^{i\in 1..n}]$ is the denotation of the name p. Intuitively, the process represents an object stored at a memory location and the name p represents the address of the object.

A method select $p.\ell$ is an expression that invokes the method labelled ℓ of the object denoted by p. In the presence of a denomination $p \mapsto [\ell_i = \varsigma(x_i)b_i \, ^{i\in 1..n}]$, where $\ell = \ell_j$ for some $j \in 1..n$, the effect of $p.\ell$ is to run the expression $b_j\{\!\{x_j \leftarrow p\}\!\}$, that is, to run the body b_j of the method labelled ℓ, with the variable x_j bound to the name of the object itself.

A method update $p.\ell \Leftarrow \varsigma(x)b$ is an expression that updates the method labelled ℓ of the object denoted by p. In the presence of a denomination $p \mapsto [\ell_i = \varsigma(x_i)b_i \, ^{i\in 1..n}]$, where $\ell = \ell_j$ for some $j \in 1..n$, the effect of $p.\ell \Leftarrow \varsigma(x)b$ is to update the denomination to be $p \mapsto [\ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i \, ^{i\in(1..n)-\{j\}}]$, and to return p as its result.

A clone $clone(p)$ is an expression that makes a shallow copy of the object denoted by p. In the presence of a denomination $p \mapsto [\ell_i = \varsigma(x_i)b_i \, ^{i\in 1..n}]$, the effect of $clone(p)$ is to generate a fresh name q with denomination $q \mapsto [\ell_i = \varsigma(x_i)b_i \, ^{i\in 1..n}]$ and to return q as its result. After a clone, the names p and q denote two copies of the same denotation $[\ell_i = \varsigma(x_i)b_i \, ^{i\in 1..n}]$; updates to one will not affect the other.

A let $let\ x{=}a\ in\ b$ is an expression that first runs the expression a, and if it returns a result, calls it x, and then runs the expression b.

A parallel composition $a \Rsh b$ is either an expression or a process, depending on whether b is an expression or a process. In $a \Rsh b$ the terms a and b are running in parallel. If b is an expression then $a \Rsh b$ is an expression, whose result, if any, is the result returned by b. Any result returned by a is ignored.

A restriction $(\nu p)a$ is either an expression or a process, depending on whether a is an expression or a process. A restriction $(\nu p)a$ generates a fresh name p whose scope is a.

Structural congruence $a \equiv b$ is the least congruence on terms to satisfy:

$(a \Rsh b) \Rsh c \equiv a \Rsh (b \Rsh c)$	
$(a \Rsh b) \Rsh c \equiv (b \Rsh a) \Rsh c$	
$(\nu p)(\nu q)a \equiv (\nu q)(\nu p)a$	
$(\nu p)(a \Rsh b) \equiv a \Rsh (\nu p)b$	if $p \notin fn(a)$
$(\nu p)(a \Rsh b) \equiv ((\nu p)a) \Rsh b$	if $p \notin fn(b)$
$let\ x{=}(let\ y{=}a\ in\ b)\ in\ c \equiv let\ y{=}a\ in\ (let\ x{=}b\ in\ c)$	if $y \notin fv(c)$
$(\nu p) let\ x{=}a\ in\ b \equiv let\ x{=}(\nu p)a\ in\ b$	if $p \notin fn(b)$
$a \Rsh let\ x{=}b\ in\ c \equiv let\ x{=}(a \Rsh b)\ in\ c$	

Reduction $a \to b$ is the least relation on terms to satisfy:

For the first three rules, let $d = [\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$.	
$(p \mapsto d) \Rsh p.\ell_j \to (p \mapsto d) \Rsh b_j\{\!\{x_j \leftarrow p\}\!\}$	if $j \in 1..n$
$(p \mapsto d) \Rsh (p.\ell_j \Leftarrow \varsigma(x)b) \to (p \mapsto d') \Rsh p$	if $j \in 1..n$, $d' = [\ell_j = \varsigma(x)b,$ $\ell_i = \varsigma(x_i)b_i \ ^{i \in (1..n)-\{j\}}]$
$(p \mapsto d) \Rsh clone(p) \to (p \mapsto d) \Rsh (\nu q)(q \mapsto d \Rsh q)$	if $q \notin fn(d)$
$let\ x{=}p\ in\ b \to b\{\!\{x \leftarrow p\}\!\}$	
$(\nu p)a \to (\nu p)a'$	if $a \to a'$
$a \Rsh b \to a' \Rsh b$	if $a \to a'$
$b \Rsh a \to b \Rsh a'$	if $a \to a'$
$let\ x{=}a\ in\ b \to let\ x{=}a'\ in\ b$	if $a \to a'$
$a \to b$	if $a \equiv a'$, $a' \to b'$, $b' \equiv b$

In the full paper, we show that this chemical semantics is equivalent to a more conventional structural operational semantics. Moreover, we identify a deterministic fragment that is closed under reduction and show that it includes Abadi and Cardelli's imperative object calculus.

Quiet and Bouncing Objects: Two Migration Abstractions in a Simple Distributed Blue Calculus

Silvano Dal-Zilio*

INRIA Sophia-Antipolis

Abstract. In this paper, we study a model of *migrating objects* based on the *blue calculus* extended with a very simple system of localities and we show how two migration behaviors can be defined, namely those of bouncing and quiet objects. These "*migration control abstractions*" are defined separately from other aspects of the object definition and can be easily reused, thus providing more flexibility in the definition of "migration constraints".

1 Introduction

The purpose of this extended abstract is to study how the behavior of concurrent objects with respect to migration, can be defined orthogonally from other aspects of the object definition, such as synchronization constraint for example. To give the reader an intuition: many researches have been conducted on the problem of defining *synchronization abstractions* for concurrent objects [10, 7], likewise, we are interested here in the definition of migration abstractions that can be reused to implement distributed objects. To this end, we give two examples of objects that act, respectively, according to the well-known client/server and agent paradigms.

We first present the calculus used to define objects, namely a version of the blue calculus [4] enriched with a simple model of localities so that we can deal with migration. In this distributed blue-calculus ($\mathbf{D}\pi^\star$), objects can be represented "as processes" [8]. In particular, we study in Sect. 3, the canonical example of the mutable cell. Then we show, using a slight modification in the process definition, how one can mimic two migration behaviors: the object that always resides at the same location and the object that migrates to the location of its clients.

2 The Calculus

The blue calculus ($\pi^\star$) is a direct extension of both the λ and the π calculi. In this paper, we consider a very simple distributed version of the original calculus introduced in [4] (see Table 1) obtained by adding locations, located processes ($[a :: P]$) and a primitive for code transfer ($\mathbf{go}\ a.P$).

In $\mathbf{D}\pi^\star$, terms are defined using three disjoint kinds of names: references: $u, v, w \cdots \in \mathcal{R}$, labels: $k, l, m \cdots \in \mathcal{L}$ and localities (or places) $a, b, c \cdots \in \mathcal{P}$. The definition $\mathbf{def}\ D\ \mathbf{in}\ P$ (where D is a sequence of mutually recursive definitions $u_1 = P_1, \ldots, u_n = P_n$, with the u_i's pairwise distinct) is a restricted an replicated version of the declaration $\langle u \Leftarrow P \rangle$, that can be understood as a resource, located at u, accessible only once. Indeed, the declaration $\langle u \Leftarrow (\boldsymbol{\lambda} x)P \rangle$

* Email: Silvano.Dal_Zilio@sophia.inria.fr. Address: INRIA Sophia-Antipolis, BP 93, 2004 route des Lucioles. F-06902 Sophia Antipolis cedex. Fax: (+33) 492.38.79.98

is the equivalent of the π-calculus input guard $u(x).P$. This construct is useful to model processes with a mutable state.

An important restriction imposed over terms is that no declaration can be defined on an abstracted reference (e.g., $(\boldsymbol{\lambda} u)\langle u \Leftarrow P\rangle$ is not a valid process). This restriction, equivalent in $\boldsymbol{\pi}$ to the one that forbids reception on received names, ensures that no new declaration on a given reference can be dynamically created.

Table 1 Syntax of the Blue Calculus with Simple Location System: $\mathbf{D}\boldsymbol{\pi}^{\star}$

x	$::=$	$u \mid a$	values
P	$::=$	$\mathtt{0} \mid (P \mid P) \mid (\boldsymbol{\nu}\, u)P \mid \mathbf{go}\ a.P \mid$	processes
		$u \mid (\boldsymbol{\lambda} x)P \mid (P\ x) \mid$	agents
		$\langle u \Leftarrow P\rangle \mid \mathbf{def}\ u_1 = P_1, \ldots, u_n = P_n\ \mathbf{in}\ P \mid$	declarations
		$[\, l_i = P_i^{\,1 \le i \le n}\,] \mid (P \cdot l)$	records
S	$::=$	$[a :: P] \mid (S \mid S) \mid (\boldsymbol{\nu}\, u)S$	locations

The model chosen to deal with distribution is very simple. A *site* is a named box, $[a :: P]$, containing a running threads P. For the sake of simplicity, we consider a flat system of locations (that is located processes are not nested) as in [9, 2]. We consider also the operation: $\mathbf{go}\ a.P$, that spawns a thread P in the location a and we denote $u@a$ the process $\mathbf{go}\ a.u$ that sends a message at location a. The reduction semantics of $\mathbf{D}\boldsymbol{\pi}^{\star}$ is given in a chemical style [3] and uses a structural equivalence ($\equiv$). The definition of the reduction relation ($\rightarrow$) uses the standard notion of *evaluation context* ($\mathrm{E}\,[\,]$), i.e. contexts such that the hole does not occur under a guard[1]. The definition includes three general rules:

$$\frac{Q \equiv P \quad P \rightarrow P'}{Q \rightarrow P'} \qquad \frac{P \rightarrow P'}{\mathrm{E}\,[P] \rightarrow \mathrm{E}\,[P']} \qquad \frac{P \rightarrow P'}{[a :: P] \rightarrow [a :: P']}$$

We refer the reader to [4, 8] for a full presentation of the reduction semantics for the calculus without localities. Axioms for structural equivalences, see Table 2, are the usual axioms for the π-calculus (including scope extrusion) extended with rules to manage application. We have omitted the rules for record selection, $(P \cdot l)$, that acts like application and we refer the reader to [5] for details. We also add two new axioms in the distributed calculus to allow spawning of restricted names over locations: $(\imath)$ to distribute references restricted by a $(\boldsymbol{\nu}\, u)P$ statement, and $(\jmath)$ to distribute definition over parallel composition. Note that the equivalent of relation $(\jmath)$ in the π-calculus, is obtained by using a behavioral equivalence. This is the the well-known replication theorem of [11, 12] in the case of channels with output-only capabilities. The reduction relation embeds different mechanisms. "Small" β reduction (r1),

[1] $\mathrm{E}\,[\,] ::= [.] \mid (\mathrm{E}\,[\,]\ x) \mid (\mathrm{E}\,[\,] \mid P) \mid (P \mid \mathrm{E}\,[\,]) \mid (\boldsymbol{\nu}\, u)(\mathrm{E}\,[\,]) \mid \mathbf{def}\ D\ \mathbf{in}\ \mathrm{E}\,[\,] \mid (\mathrm{E}\,[\,] \cdot l)$

Table 2 Structural Equivalence

$$\begin{array}{l}
\mathbf{def}\ D\ \mathbf{in}\ (\mathbf{def}\ D'\ \mathbf{in}\ P) \equiv \mathbf{def}\ D, D'\ \mathbf{in}\ P \\
(\mathbf{def}\ D\ \mathbf{in}\ P)\ x \equiv \mathbf{def}\ D\ \mathbf{in}\ (P\ x) \\
(P \mid Q)\ x \equiv (P\ x) \mid (Q\ x) \\
(\langle v \Leftarrow P \rangle\ x) \equiv \langle u \Leftarrow P \rangle \\
(\mathbf{go}\ a.P)\ x \equiv \mathbf{go}\ a.(P\ x) \\
\mathbf{def}\ D\ \mathbf{in}\ (\mathbf{go}\ a.P) \equiv \mathbf{go}\ a.(\mathbf{def}\ D\ \mathbf{in}\ P) \\
(\imath)\ [a :: (\boldsymbol{\nu} u)P] \equiv (\boldsymbol{\nu} u)([a :: P]) \\
(\jmath)\ \mathbf{def}\ D\ \mathbf{in}\ (P \mid Q) \equiv (\mathbf{def}\ D\ \mathbf{in}\ P \mid \mathbf{def}\ D\ \mathbf{in}\ Q)
\end{array}$$

definition folding (r2), resource fetching (r3) and record selection (r4):

$$\begin{array}{lll}
(\mathrm{r1}) & (\boldsymbol{\lambda} x)P\ y \ \rightarrow\ P\{y/x\} & \\
(\mathrm{r2}) & \left(\begin{array}{l}\mathbf{def}\ D, u = R, D' \\ \mathbf{in}\ \mathrm{E}\,[u\ x_1 \ldots x_n]\end{array}\right) \rightarrow \left(\begin{array}{l}\mathbf{def}\ D, u = R, D' \\ \mathbf{in}\ \mathrm{E}\,[R\ x_1 \ldots x_n]\end{array}\right) & (u \notin \mathrm{bn}(\mathrm{E})) \\
(\mathrm{r3}) & \langle u \Leftarrow P \rangle \mid u\ x_1 \ldots x_n \rightarrow P\ x_1 \ldots x_n & \\
(\mathrm{r4}) & [\,l = P\,,\ Q\,] \cdot l \rightarrow P \quad \text{and} \quad [\,l = P\,,\ Q\,] \cdot k \rightarrow Q \cdot k & (k \neq l)
\end{array}$$

We also add the reduction rule for the **go** statement (note that process P cannot execute under the guard $\mathbf{go}\ a.P$)

$$(\mathrm{r5}) \quad [b :: (\mathbf{go}\ a.P \mid Q)] \mid [a :: R] \ \rightarrow\ [b :: Q] \mid [a :: (P \mid R)]$$

Example 1. A typical reduction sequence in $\mathbf{D}\boldsymbol{\pi}^\star$ is the one such that a message carrying a private reference is send remotely.

$$\begin{array}{rcl}
\left(\begin{array}{c}[b :: \mathbf{def}\ u = R\ \mathbf{in}\ (v@a\ u \mid P)] \\ \mid [a :: \langle v \Leftarrow Q \rangle]\end{array}\right) & \equiv & \left(\begin{array}{c}[b :: \mathbf{go}\ a.(\mathbf{def}\ u = R\ \mathbf{in}\ v\ u) \mid \mathbf{def}\ u = R\ \mathbf{in}\ P] \\ \mid [a :: \langle v \Leftarrow Q \rangle]\end{array}\right) \\
& \rightarrow^* & \left(\begin{array}{c}[b :: \mathbf{def}\ u = R\ \mathbf{in}\ P] \\ \mid [a :: \mathbf{def}\ u = R\ \mathbf{in}\ (Q\ u)]\end{array}\right) \qquad (\text{if } u \notin \mathrm{fn}(Q))
\end{array}$$

To conclude, let us just state that, while the blue calculus is a *name passing* calculus (that is a process can only be applied to a name and not to another process), the "high-order" λ-calculus application can be recovered using the definition:

$$(P\ Q) =_{\mathrm{def}} \mathbf{def}\ u = Q\ \mathbf{in}\ (P\ u) \qquad (u \notin \mathrm{fn}(P) \cup \mathrm{fn}(Q))$$

Moreover this definition of application is coherent with our model of distribution since we can prove that $(\mathbf{go}\ a.P)\ Q \equiv \mathbf{go}\ a.(P\ Q)$.

3 Modeling Objects in the Blue Calculus

In this extended abstract, we will concentrate on a single example of object, namely the "mutable cell". Although it is only an example, it is a representative one, since in [8] we

show how to derived a "complete" calculus of concurrent objects using cells and extensible records. Thus, the result given for the cell example can be derived for more general objects. The constructs of this object calculus, together with its derived operational rules, are given for information in Sect. 3 (consideration on types are omitted). Let $\mathrm{R}_o(b)$ denotes the record:

$$\mathrm{R}_o(b) \ =_{\mathrm{def}} \ \big[\, get = (\boldsymbol{\lambda} x)(o\ b \mid x\ b),\ put = (\boldsymbol{\lambda} x)(o\ x) \,\big]$$

The cell process with "name " O is defined by:

$$\textsc{Cell}(O) \ =_{\mathrm{def}} \ \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\mathbf{in}\, o$$

and the application: $(\textsc{Cell}(O)\ a_0)$, initializes the cell to the value a_0. It is easy to see that[2] $(\textsc{Cell}\ a_0 \mid O \cdot get\ r)$ and $(\textsc{Cell}\ a_0 \mid O \cdot put\ a)$ evaluate in a deterministic way:

$$\begin{array}{rl} (\textsc{Cell}\ a_0) \mid (O \cdot get\ r) & \rightarrow \ \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\mathbf{in}\, (\langle O \Leftarrow \mathrm{R}_o(a_0)\rangle \mid O \cdot get\ r) \\ & \rightarrow \ \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\mathbf{in}\, (\mathrm{R}_o(a_0) \cdot get\ r) \\ & \rightarrow^* \ \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\mathbf{in}\, (o\ a_0 \mid r\ a_0) \ \equiv \ (\textsc{Cell}\ a_0) \mid r\ a_0 \end{array}$$

$$(\textsc{Cell}\ a_0) \mid (O \cdot put\ a) \rightarrow^* \ \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\mathbf{in}\, (o\ a) \ \equiv \ (\textsc{Cell}\ a)$$

It is interesting to notice the linear use of the reference O in $(\textsc{Cell}(O)\ a)$. If the cell is invoked, we consume the unique declaration $\langle O \Leftarrow \mathrm{R}_o(a)\rangle$. Thus, a unique message $(o\ a')$, acting like a lock, is freed in the evaluation process, which, in turn, frees a single declaration $\langle O \Leftarrow \mathrm{R}_o(a')\rangle$. Thus, we have the invariant that there is exactly one resource available at address O, and that this resource keeps the last value passed in a $(O \cdot put)$ call.

4 A Concurrent Calculus of Objects

More elaborate objects than the (canonical) example of the mutable cell can be defined. In this section, we introduce a calculus of concurrent objects by specifying a set of operators and their operational semantics, and we define these operators (and their associated reduction rules) with an encoding in $\pi^\star$. In the specification of this calculus (Table 3), we distinguish a subset $\mathcal{O}$ of references (which we call objects names, $O, A, B, \cdots \in \mathcal{O}$) and we use L to denote an "object body": $L = l_1 = \varsigma(x_1)P_1, \ldots, l_n = \varsigma(x_n)P_n$. An example of object is the one that produces an infinite copy of itself. Let L be the body: $l = \varsigma(x)(\mathbf{clone}(x) \mid x \Leftarrow\!\!\!\!= l)$, then:

$$\mathbf{obj}\, O = \{L\}\,\mathbf{in}\, O \Leftarrow\!\!\!\!= l \ \rightarrow^*_{\pi^\star_\varsigma} \ \mathbf{obj}\, O = \{L\}\,\mathbf{in}\, (\mathbf{obj}\, A = \{L\}\,\mathbf{in}\, (A \mid O \Leftarrow\!\!\!\!= l)) \ \rightarrow^*_{\pi^\star_\varsigma} \ \ldots$$

another example, using method overriding, is:

$$\begin{array}{rl} \mathbf{obj}\, O = \{L\}\,\mathbf{in}\, (O \leftarrow l = \varsigma(x)x) \Leftarrow\!\!\!\!= l & \rightarrow_{\pi^\star_\varsigma} \ \mathbf{obj}\, O = \{l = \varsigma(x)x\}\,\mathbf{in}\, O \Leftarrow\!\!\!\!= l \\ & \rightarrow_{\pi^\star_\varsigma} \ \mathbf{obj}\, O = \{l = \varsigma(x)x\}\,\mathbf{in}\, O \end{array}$$

4.1 Interpretation of the Derived Object Constructs

Processes modeling objects are inspired from the encoding of the mutable cell. In the definition given in Table 4, an object $(\mathbf{obj}\, O = \{l_i = \varsigma(x_i)P_i{}^{1 \le i \le n}\}\mathbf{in}\ P)$, is a cell with an additional field $clone$ to allow object cloning. In this definition, a method $\varsigma(x)P$ is an abstraction $(\boldsymbol{\lambda} x)P$. This function, also called premethod, has one argument: the name of the current object (also called the self-parameter). The cell "memorizes" a record of premethods:

[2] to simplify the examples, we use $\textsc{Cell}$ to denote $\textsc{Cell}(O)$

Table 3 Specification of Operators and Reduction Rules for Objects in $\pi^\star$

$\varsigma(x)P$	method with self parameter x and body P
$\mathbf{obj}\, O = \left\{ l_i = \varsigma(x_i)P_i^{\,1\le i\le n} \right\} \mathbf{in}\, P$	object with n methods $l_1, \ldots, l_n$
$P \Leftarrow l$	invocation of method l
$O \leftarrow l = \varsigma(x)Q$	update of method l with body $\varsigma(x)P$
$\mathbf{clone}(O)$	cloning of object O

Let $L =_{\mathrm{def}} l_1 = \varsigma(x_1)P_1, \ldots, l_n = \varsigma(x_n)P_n$

$$\mathbf{obj}\, O = \{L\} \mathbf{in}\, \mathrm{E}\,[O \Leftarrow l_j] \;\rightarrow_{\pi^\star_\varsigma}\; \mathbf{obj}\, O = \{L\} \mathbf{in}\, \mathrm{E}\,[P_j\{O/x_j\}]$$

$$\mathbf{obj}\, O = \{L\} \mathbf{in}\, \mathrm{E}\,[O \leftarrow l_j = \varsigma(x)P] \;\rightarrow_{\pi^\star_\varsigma}\; \mathbf{obj}\, O = \{l_j = \varsigma(x)P, l_i = \varsigma(x_i)P_i^{\,i\neq j}\} \mathbf{in}\, \mathrm{E}\,[O]$$

$$\mathbf{obj}\, O = \{L\} \mathbf{in}\, \mathrm{E}\,[\mathbf{clone}(O)] \;\rightarrow_{\pi^\star_\varsigma}\; \mathbf{obj}\, O = \{L\} \mathbf{in}\, (\mathbf{obj}\, A = \{L\} \mathbf{in}\, \mathrm{E}\,[A]) \quad (A \notin \mathrm{bn}(\mathrm{E}))$$

$[\, l_i = (\boldsymbol{\lambda} x_i)P_i^{\,1\le i\le n}\,]$, as in the classical recursive records semantics [6]. Note that we restrict the scope of an object name to the definition of the object it refers to. Thus we have the invariant that there is a unique declaration for each object names. Moreover, method update returns "a reference" to the modified object. This is almost the behaviour of (sequential) objects in the ς calculus of Abadi and Cardelli [1].

$$\mathrm{S}_o(b) \;=_{\mathrm{def}}\; \left[\begin{array}{l} get = (\boldsymbol{\lambda} x)(o\ b \mid x\ b\ O), \\ modify = (\boldsymbol{\lambda} x)(x\ o\ b \mid O), \\ clone = (o\ b \mid x_{clone}\ b) \end{array}\right]$$

$$\mathrm{OBJ}(O) \;=_{\mathrm{def}}\; \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{S}_o(b)\rangle \mathbf{in}\, o$$

Table 4 Definition of the Derived Operators for Objects

$$P \Leftarrow l =_{\mathrm{def}} (P \cdot get\ (\boldsymbol{\lambda} x)(x \cdot l)) \qquad \mathbf{clone}(O) =_{\mathrm{def}} (O \cdot clone)$$

$$O \leftarrow l = \varsigma(x)Q =_{\mathrm{def}} (O \cdot modify\ (\boldsymbol{\lambda} ob)(o\,[\, l = (\boldsymbol{\lambda} x)Q\,,\ b\,]))$$

$$\mathbf{obj}\, O = \left\{ l_i = \varsigma(x_i)P_i^{\,1\le i\le n} \right\} \mathbf{in}\, P =_{\mathrm{def}} \left\{\begin{array}{l} \mathbf{def}\ x_{clone} = (\boldsymbol{\lambda} b)(\boldsymbol{\nu} A)(\mathrm{OBJ}(A)\ b \mid A) \\ \mathbf{in}\ (\boldsymbol{\nu} O)((\mathrm{OBJ}(O)\,[\, l_i = (\boldsymbol{\lambda} x_i)P_i^{\,1\le i\le n}\,]) \mid P) \end{array}\right.$$

Another remark is that we use only field selection and application in the definition of cloning, method invocation and method update. Thus the definition of structural equivalence allows us, for example, to derive the following laws, showing that these (derived) operators acts like application:

$$(\mathbf{def}\, D \mathbf{in}\, P \mid Q) \Leftarrow l \;\equiv\; \mathbf{def}\, D \mathbf{in}\, (P \Leftarrow l \mid Q \Leftarrow l) \qquad (\langle u \Leftarrow P\rangle \Leftarrow l) \;\equiv\; \langle u \Leftarrow P\rangle$$

The next result states that there is an operational correspondence between $\rightarrow_{\pi^\star_\varsigma}$ (defined in Table 3) and $\rightarrow$. These properties are proved using a simple induction.

Theorem 1 (Operational Equivalence). *The specification of the object reduction rules is* complete *with respect to the encoding of objects in* $\pi^\star$*:* $P \rightarrow_{\pi^\star_\varsigma} P' \Rightarrow P \rightarrow^* P'$*. It is also* sound*:* $P \rightarrow Q$ *implies that* $Q \rightarrow^* Q'$ *with* $P \rightarrow_{\pi^\star_\varsigma} Q'$*.*

5 Quiet Versus Bouncing Cells

In our model of distribution, synchronization (rule (r3)) does not extend over location boundaries. Thus communication is local, and, to interact with a remote declaration at location a, one has to first spawn a message at a. For example, the process $([b :: u] \mid [a :: \langle u \Leftarrow P \rangle])$ is inert while $([b :: u@a] \mid [a :: \langle u \Leftarrow P \rangle])$ reduces to $([b :: 0] \mid [a :: P])$. Another remark is that the result of the communication is always executed at the location of the declaration. This is reminiscent of the client-server paradigm of computation such that clients "controls" the computation, that takes place at the server location, by sending remote messages.

The "migration behavior" of the mutable cell object and of the declaration $\langle O \Leftarrow \mathrm{R}_o(b) \rangle$ are equal. For example, to read the content of the mutable cell from a remote location, one has to (1) move to the location of the cell, then (2) invoke the method read and finally (3) fetch the result back. After completion, the cell is still at the same location.

$$\begin{pmatrix} [a :: (\textsc{Cell}\ a_0) \mid P] \mid \\ [b :: (O@a \cdot get\ r@b) \mid Q] \end{pmatrix} \rightarrow^* [a :: (\textsc{Cell}\ a_0) \mid (r@b\ a_0) \mid P] \mid [b :: Q] \rightarrow^* [a :: (\textsc{Cell}\ a_0) \mid P] \mid [b :: (r\ a_0) \mid Q]$$

Likewise, objects created using the cell also share the same "migration behavior", i.e. they are *quiet* objects since they never leave the location of their creation.

In this section, we define a new migration abstraction, namely the *agent* behavior, based on a new declaration statement: $\langle u \Leftarrow P \rangle_{\mathrm{agt}}$ (see Table 6), that can be derived in $\mathbf{D}\pi^\star$. Roughly speaking, the result of a remote communication involving $\langle u \Leftarrow P \rangle_{\mathrm{agt}}$ takes place at the client location instead of the declaration one.

Table 5 Two Distributed Cells

$$\textsc{Cell}_{\mathrm{c/s}}(O) =_{\mathrm{def}} \mathbf{def}\ o = (\lambda b)\langle O \Leftarrow \mathrm{R}_o(b) \rangle_{\mathrm{c/s}}\ \mathbf{in}\ o$$

$$\textsc{Cell}_{\mathrm{agt}}(O) =_{\mathrm{def}} \mathbf{def}\ o = (\lambda b)\langle O \Leftarrow \mathrm{R}_o(b) \rangle_{\mathrm{agt}}\ \mathbf{in}\ o$$

The operational semantics of an agent declaration strongly depends on the distribution of the processes. Indeed in $\pi^\star$ (and in π) there are no explicit locations and (therefore) *where* a communication comes from is not observable. But this information does count in a distributed setting. If we redefine the cell object using $\langle O \Leftarrow [\ldots] \rangle_{\mathrm{agt}}$ instead of $\langle O \Leftarrow [\ldots] \rangle$,

denoting it CELL_{agt}, we obtain a mutable cell bouncing from locations to locations according to communications with the client.

$$\left(\begin{array}{l}[a :: (\text{CELL}_{\text{agt}}\ a_0) \mid P] \mid \\ [b :: (O@a \cdot get\ r) \mid Q]\end{array}\right) \rightarrow^* \left(\begin{array}{l}[a :: P] \mid \\ [b :: (\text{CELL}_{\text{agt}}\ a_0) \mid (r\ a_0) \mid Q]\end{array}\right)$$

to sketch the encoding of this new construct, let us just say that remote messages $[b :: u@a]$ are translated to $[b :: \mathbf{go}\ a.(u\ b)]$ (that is the same message with, as extra argument, the "departure location"), and that the declaration $\langle u \Leftarrow P \rangle_{\text{agt}}$ and $\langle u \Leftarrow P \rangle_{\text{c/s}}$ are defined by:

$$\langle u \Leftarrow P \rangle_{\text{agt}} =_{\text{def}} \langle u \Leftarrow (\boldsymbol{\lambda} a)(\mathbf{go}\ a.P) \rangle$$

$$\langle u \Leftarrow P \rangle_{\text{c/s}} =_{\text{def}} \langle u \Leftarrow (\boldsymbol{\lambda} a)P \rangle \qquad (\text{with } a \notin \text{fn}(P))$$

Other object migration abstractions can be uniformly defined by first defining a new kind of declaration. For example, an *applet* object can be interpreted as an object that does not change location but that spawn a copy (or clone) of itself at the location of its client. We can give this behavior to an object using, for its definition, a new declaration construct, $\langle u \Leftarrow P \rangle_{\text{applet}}$ (see Table 6), that can also be derived in $\mathbf{D}\pi^\star$. For example $\langle u \Leftarrow P \rangle_{\text{applet}}$ can be translated to:

$$\langle u \Leftarrow P \rangle_{\text{applet}} =_{\text{def}} \mathbf{def}\ x = \langle u \Leftarrow (\boldsymbol{\lambda} a)(\mathbf{go}\ a.P \mid x) \rangle\ \mathbf{in}\ x$$

Table 6 Client/Server, Agent and Applet Communications

$$\left(\begin{array}{l}[a :: \langle u \Leftarrow P \rangle_{\text{c/s}} \mid R] \mid \\ [b :: (\mathbf{def}\ D\ \mathbf{in}\ u@a\ v_1 \ldots v_n) \mid Q]\end{array}\right) \rightarrow^* \left(\begin{array}{l}[a :: (\mathbf{def}\ D\ \mathbf{in}\ P\ v_1 \ldots v_n) \mid R] \mid \\ [b :: Q]\end{array}\right)$$

$$\left(\begin{array}{l}[a :: \langle u \Leftarrow P \rangle_{\text{agt}} \mid R] \mid \\ [b :: (\mathbf{def}\ D\ \mathbf{in}\ u@a\ v_1 \ldots v_n)) \mid Q]\end{array}\right) \rightarrow^* \left(\begin{array}{l}[a :: R] \mid \\ [b :: (\mathbf{def}\ D\ \mathbf{in}\ P\ v_1 \ldots v_n) \mid Q]\end{array}\right)$$

$$\left(\begin{array}{l}[a :: \langle u \Leftarrow P \rangle_{\text{applet}} \mid R] \mid \\ [b :: (\mathbf{def}\ D\ \mathbf{in}\ u@a\ v_1 \ldots v_n)) \mid Q]\end{array}\right) \rightarrow^* \left(\begin{array}{l}[a :: \langle u \Leftarrow P \rangle_{\text{applet}} \mid R] \mid \\ [b :: (\mathbf{def}\ D\ \mathbf{in}\ P\ v_1 \ldots v_n) \mid Q]\end{array}\right)$$

6 Conclusions and Future Work

A challenging problem encountered in the design of programming languages with concurrent objects, is to be able to express the synchronization control of objects in a compositional and reusable way. Now that "network-oriented" languages have added functionalities to remotely download code and to migrate objects, a similar problem arises in the description of

the migration behavior.

In the present paper, we have presented how two abstractions for the definition of the migration behavior can be applied to define mutable cell objects that react differently when invoked by a remote client. Using a translation of a calculus of concurrent objects in $\pi^\star$ defined in [8], we claim that those abstractions can be uniformly applied to every object creation. Moreover, These abstractions can be transposed to other process calculi, and in particular to distributed versions of the π-calculus [14, 9, 2], and to other interpretation of objects [13].

The principal advantage of defining "standardized behaviors of migration", is that we can define migrating objects from objects designed in an non-distributed language without adding any explicit thread of control, thus providing a flexible and simple way to automatically add migration features to objects. But further works must be done to define more elaborated behaviors than those presented in this abstract. For example it will be interesting to give an accurate model of the mobile agents behavior as defined in TELESCRIPT [15].

References

1. Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 2(125):78–102, March 1996.
2. Roberto Amadio. An asynchronous model of locality, failure and process mobility. In *COORDINATION 97*, volume 1282 of *Lecture Notes in Computer Science*, 1997.
3. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
4. Gérard Boudol. The π-calculus in direct style. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, Paris, France, 15–17 January 1997.
5. Gérard Boudol. Typing the use of resources in a concurrent calculus. In *ASIAN'97, the Asian Computing Science Conference*, Lecture Notes in Computer Science, Kathmandu, December 1997.
6. L. Cardelli and J. C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1):3–48, 1991.
7. Denis Caromel. Programming abstractions for concurrent programming. In *Technology of Object-Oriented Languages and Systems (TOOLS Pacific'90)*, pages 245–253, Sydney, November 1990.
8. Silvano Dal-Zilio. Concurrent objects in the blue calculus. (draft) `http://www.inria.fr/meije/personnel/Silvano.Dal_Zilio/`.
9. Matthew Hennessy and James Riely. Ressource access control in systems of mobile agents. Technical Report 2/98, University of Sussex, February 1998.
10. Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 4, pages 107–150. The MIT Press, 1993.
11. Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.
12. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 11, 1995.
13. Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. Technical Report 3000, INRIA, 1996.
14. Peter Sewell. Global/local subtyping for a distributed pi-calculus. Technical Report 435, University of Cambridge, 1997.
15. Jim White. Mobile agent white paper. Technical report, General Magic, 1996.

Surrogates in Øjeblik: Towards Migration in Obliq

Hans Hüttel, Josva Kleist, Uwe Nestmann, Davide Sangiorgi
BRICS*Aalborg University INRIA Sophia-Antipolis

June 15, 1998

Abstract

In Cardelli's lexically scoped, distributed, object-based programming language Obliq, object migration was suggested as creating a (remote) copy of an objects' state at the target site, followed by turning the (local) object itself into a *surrogate*, i.e. a pointer to the just created remote copy. This kind of migration is only *safe*—migrated objects shall behave the same before and after migration—if it is *protected* and *serialized*. Protected objects can only be accessed by clients via selection, and within a serialized object at most one method can be active at any time. Yet, since Obliq does not have a formal semantics, there is no proof of this claim.

In this abstract, we consider the act of creating object surrogates as an abstraction of the above-mentioned style of migration. We introduce the language Øjeblik, a simplified distribution-free subset of Obliq, and give its formal semantics in terms of an encoding into the π-calculus. This semantics shall provide the ground for proving that surrogation is safe, thus suggesting that migration in Obliq is safe, accordingly.

1 Motivation

Our language Øjeblik for studying surrogation, is an object-based language which is not only inspired by Obliq [Car95], but rather represents its concurrent core. Obliq is a lexically scoped, distributed, object-based programming language. Lexical scoping in distributed settings makes program analysis easier since the binding of variables is only determined by their location in the program text and not by the site at which execution takes place.

It can be advantageous to migrate an object from one site to another, which is also called for in Obliq. Here, however, mutable values in general are never sent over the network; instead, only network references are transmitted. Accordingly, migration of objects is carried out in Obliq by creating a copy of the object at the target site and then modify the original (local) object such that it forwards all future requests to the new (remote) object. In Øjeblik, we ignore all distributed aspects of migration—they are not essential for the results of Obliq computations anyway—and concentrate on just the concurrent aspects: *surrogation* of an object, say: a, can be described as creating a copy b of a and then turn a itself into a 'pointer' to b, i.e., which forwards all future request for methods of a to b.

*Basic Research in Computer Science, Centre of the Danish National Research Foundation.

2 Surrogation in Øjeblik

In this section, we present Øjeblik as an untyped language[1], although we sometimes refer to types when we think it helps us in explaining our design decisions or it eases the understanding. Øjeblik-expressions are given by the following grammar:

$$
\begin{array}{rcll}
a, b, c, \ldots & ::= & \{\mathsf{P}, \mathsf{S}, r, [\mathrm{l}_j{=}m_j]_{j\in J}\} & \text{object construction} \\
& | & a.\mathrm{l}\langle\, a_1 \ldots a_n \,\rangle & \text{method invocation} \\
& | & a.\mathrm{l}{\Leftarrow}m & \text{method update} \\
& | & a.\mathsf{alias}(b) & \text{object aliasing} \\
& | & a.\mathsf{clone} & \text{shallow copy} \\
& | & a.\mathsf{surrogate} & \text{object surrogation} \\
& | & \mathsf{let}\, x = a\, \mathsf{in}\, b & \text{local definition} \\
& | & s, x, y, z & \text{variables} \\
& | & \mathsf{fork}(a) & \text{thread creation} \\
& | & \mathsf{join}(a) & \text{thread destruction} \\
m & ::= & \varsigma(s, \tilde{x})b & \text{method}
\end{array}
$$

where method labels l and (run-time) object references r are taken from disjoint sets.

An object $\{\mathsf{P}, \mathsf{S}, r, [\mathrm{l}_j{=}m_j]_{j\in J}\}$ consists of a finite collection of named methods $\mathrm{l}_j{=}m_j$, more generally called fields, for pairwise distinct labels l_j; the P and S keywords are optional and refine how an object reacts to external messages, while the aliasing reference r can be used to forward most requests to another object, so a non-aliased object has just no entry. Note that we introduce object references, actual run-time entities, explicitly in the syntax of Øjeblik only in order to clarify the semantics of aliasing; an actual user-level language would not need any entry at all. In a method $\varsigma(s, \tilde{x})b$ the letter ς is a binder for the self variable s and argument variables $\tilde{x}$ within the method body b.

Let a be a non-aliased object. Methods can only be activated, when also supplying the required actual parameters: $a.\mathrm{l}\langle\, a_1 \ldots a_n \,\rangle$ with l denoting the method $\varsigma(s, \tilde{x})b$ results in the body b with the enclosing object a bound to the self variable s, and the actual parameters $a_1 \ldots a_n$ of the invocation bound to the formal parameters $\tilde{x}$. The expression $a.\mathrm{l}{\Leftarrow}m$ updates the content of the named field l in a with method m and evaluates to the modified object. On aliased objects, by constrast, invocations and updates are forwarded to the aliasing target instead of operating on the aliasing source.

Every object in Øjeblik comes equipped with three special methods for aliasing, cloning, and surrogation, none of which can be overwritten by the update operation, and only one of which is subject to aliasing. In contrast to Obliq, we do not allow aliasing of individual fields, but only aliasing of objects themselves[2]: $a.\mathsf{alias}(b)$ with $a = \{\mathsf{P}, \mathsf{S}, r, [\mathrm{l}_j{=}m_j]_{j\in J}\}$ results in $\{\mathsf{P}, \mathsf{S}, [b], [\mathrm{l}_j{=}m_j]_{j\in J}\}$, overriding the current alias r with the value $[b]$ of b, assuming

[1] Obliq also is an untyped language, although types can be added in a rather straightforward manner.

[2] In Obliq, object aliasing (called redirection) is derived from field aliasing.

that b evaluates to an object reference with 'type' matching that of a. Thus, as a special case the aliasing method itself (like cloning) is not captured by aliasing[3]. Consequently, the behavior of an Øjeblik-object can only be changed directly via method update or else indirectly by aliasing.

The operation a.clone creates an object with the same (named and optional) fields as the original object and initializes the fields to the same values as in the original object.

The operation a.surrogate shall behave as $\mathsf{let}\, x = a.\mathsf{clone}\, \mathsf{in}\, a.\mathsf{alias}(x); x$. Here, as usual, the expression $\mathsf{let}\, x = a\, \mathsf{in}\, b$ first evaluates a, binding the result to x, and then evaluates b within the scope of the new binding. Moreover, $a; b$ abbreviates $\mathsf{let}\, x = a\, \mathsf{in}\, b$ for $x \notin \mathrm{fv}(b)$. Since surrogation is meant to be an abstraction of migration, the correct interpretation of double-surrogation—a.surrogate; a.surrogate equivalent to a.surrogate.surrogate—requires that surrogate methods are, in general, subject to aliasing, otherwise the migration of an already migrated object would mistakenly migrate the surrogate.

To create a new concurrent thread we use the fork command. The expression $\mathsf{fork}(a)$ returns a thread identifier to denote a new thread evaluating a. To get the result of a computation in fork'ed thread the join command is used. If a evaluates to a thread identifier, then $\mathsf{join}(a)$ either returns the value that the thread denoted by a has evaluated to, blocks until the thread finishes and then returning the resulting value, or blocks forever if a join on the a thread has already been performed[4].

Serialization and Protection based on Self-Infliction

The three basic kinds of operation on Øjeblik-objects—invocation, update, and the special operations—can be performed as either external operations on an object or through self as internal operations. A *self-inflicted operation* is an operation, that is performed on the current self, i.e., the self of the last method invoked in the current thread that has not completed; an operation is *external* if it is not self-inflicted.

In concurrent object-based settings, the invariant that at most one thread at a time may be active within an object is called *serialization*. The simplest way to ensure serialization is to associate with an object a mutex that is locked when a tread enters the object and released when the thread exits the object. However, this approach is too restrictive—it is not possible for a method to call one of its siblings. Instead, *self-serialization* requires that the mutex is only acquired for external operations. This allows a method to call its siblings through self, but excludes mutual recursion, where a method in an object a calls a method in another object b, which then tries to "call back" to activate a method in a.

With self-infliction, it is also easy to define a sensible method for the *protection* of objects against external modifications: for protected objects, updates and the special operations are only allowed, if these operations are self-inflicted. Both, protection and self-serialization are used Obliq and also in Øjeblik: we use the keyword S for self-serialized and the keyword P for protected Øjeblik-objects.

[3]Note that this is consistent with re-aliasing in Obliq.

[4]In Obliq, an exception will be raised.

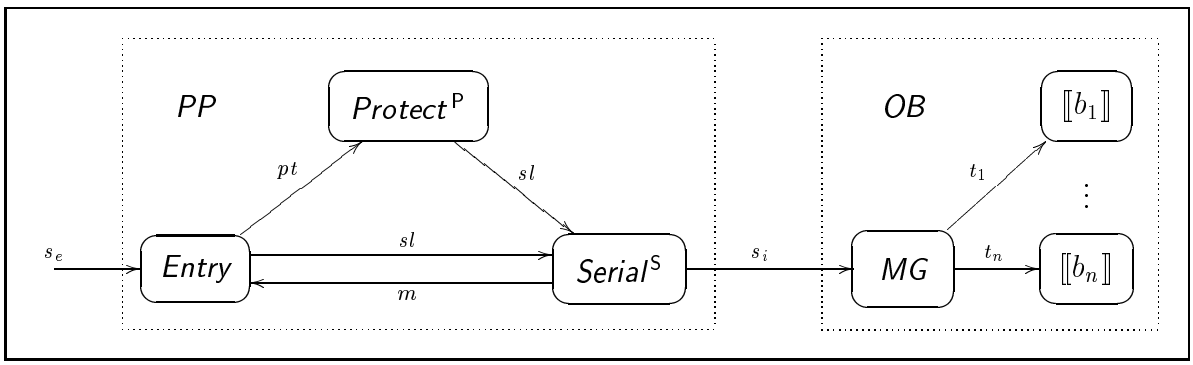


Figure 1: Structure of Øjeblik-objects

3 A formalization using the π-Calculus

We give the semantics of Øjeblik as a translation into the local asynchronous π-calculus Lπ of Merro and Sangiorgi [MS98], equipped with matching. This gives us the possibility of using process-algebraic proof techniques to reason about surrogation.

Based on previous work (e.g. by Kleist and Sangiorgi on the imperative object calculus IOC [KS98]), the main new idea of the translation is the distinction between two different self identifiers: s_e to serve external requests, and s_i to serve internal requests. By this distinction, the requirements of protection and serialization can be decided solely based on the access to the self identifiers. With $\mathbb{O} := [\mathrm{l}_j{=}\varsigma(s,\tilde{x})b_j]_{j\in J}$, a (non-aliased) object is translated into the composition of an *object* module *OB* and a *pre-processing* module *PP*:

$$[\![\{\mathsf{P},\mathsf{S},-,\mathbb{O}\}]\!]_p^{s_c} \quad \stackrel{\text{def}}{=} \quad (\boldsymbol{\nu}\, s_e, s_i)\,\big(\,\overline{p}\langle s_e\rangle \mid PP_{\mathbb{O}}^{\mathsf{P},\mathsf{S}}\langle\, s_e, s_i\,\rangle \mid OB_{\mathbb{O}}\langle\, s_e, s_i\,\rangle\,\big)$$

where s_c represents the current self of the execution. The *OB*-module is similar to the encoding of imperative objects: it consists of processes representing the method bodies, and an object *manager* process *MG* that deals with the different kinds of requests. The *PP*-module models the refined functionality for distinguishing and handling requests based on an object's protection and serialization. The structure of the encoding and its code are shown in Figures 1, 2, 3, 4, and 5. In the following paragraphs, we comment on the details.

Pre-processing The *Entry* serves as receptor for external requests; only method invocations are passed on to *Serial*, all other request are forwarded to *Protect*. Note that external requests can be grabbed only after some signal has arrived on channel m. This guarantees mutual exclusion (not by itself, but in cooperation with *Serial*, which is to release signals on m) concerning the activities between the external and internal interfaces.

Protect has a very simple definition. Note that all incoming requests on channel pt are necessarily external and, due to the behavior of *Entry*, they can only be requests for updating or one of the special operations. If protection is turned on (P=T), all of theses requests lead to a run-time error according to Obliq's informal semantics. Otherwise, incoming requests are simply forwarded to the serialization module.

$$
\begin{array}{|rcl|}
\hline
\mathit{PP}^{\mathsf{P},\mathsf{S}}_{\mathbb{O}}\langle\, s_e, s_i \,\rangle & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} m, sl, pt)\ \big(\ \overline{m}\langle\rangle \mid \mathit{Entry}_{\mathbb{O}}\langle\, s_e, pt, sl, m \,\rangle \\
& & \qquad\qquad \mid \mathit{Protect}^{\mathsf{P}}_{\mathbb{O}}\langle\, pt, sl \,\rangle \\
& & \qquad\qquad \mid \mathit{Serial}^{\mathsf{S}}_{\mathbb{O}}\langle\, sl, m, s_i \,\rangle\ \big) \\
\hline
\mathit{Entry}_{\mathbb{O}}\langle\, s_e, pt, sl, m \,\rangle & \stackrel{\text{def}}{=} & !\, m().s_e(x).\mathsf{case}\ x\ \mathsf{of}\ \mathrm{l_inv_}(\ldots) : \overline{sl}\langle x\rangle \\
& & \qquad\qquad\qquad\qquad\quad \mathsf{else} \quad\ : \overline{pt}\langle x\rangle \\
\hline
\mathit{Protect}^{\mathsf{P}=\mathsf{T}}_{\mathbb{O}}\langle\, pt, sl \,\rangle & \stackrel{\text{def}}{=} & !\, pt(x).\mathsf{wrong} \\
\mathit{Protect}^{\mathsf{P}=\mathsf{F}}_{\mathbb{O}}\langle\, pt, sl \,\rangle & \stackrel{\text{def}}{=} & !\, pt(x).\overline{sl}\langle x\rangle \\
\hline
\mathit{Serial}^{\mathsf{S}=\mathsf{F}}_{\mathbb{O}}\langle\, sl, m, s_i \,\rangle & \stackrel{\text{def}}{=} & !\, sl(\mathrm{h_}\langle p, s, \tilde{x}\rangle).\qquad \big(\ \overline{s_i}\langle \mathrm{h_}\langle p, s_i, \tilde{x}\rangle\rangle \mid \qquad\qquad \overline{m}\langle\rangle\ \big) \\
\mathit{Serial}^{\mathsf{S}=\mathsf{T}}_{\mathbb{O}}\langle\, sl, m, s_i \,\rangle & \stackrel{\text{def}}{=} & !\, sl(\mathrm{h_}\langle p, s, \tilde{x}\rangle).(\boldsymbol{\nu} r)\ \big(\ \overline{s_i}\langle \mathrm{h_}\langle r, s_i, \tilde{x}\rangle\rangle \mid r(y).(\overline{p}\langle y\rangle \mid \overline{m}\langle\rangle)\ \big) \\
\hline
\end{array}
$$

Figure 2: Encoding of Øjeblik-objects: Pre-processing

Serial is not so complicated either. In Figure 2, we use enhanced pattern matching to unify single- and double-selections on records, which helps us to emphasize the essential idea: the label h may range over requests of the form l_j_inv, l_j_upd, as well as ali, cln, and sur, each carrying as parameters at least a result channel r and a current-self identifier s ($\tilde{x}$ may be empty; Figure 4 shows the generation of such requests by objects' clients).

The main task of *Serial*—mutual exclusion—is performed by the rather simple and standard technique of using a lock, i.e. a message on some local channel m which obeys the invariant that at any time there is at most one message on it available in the system. If serialization is turned on (S=T), we need to extract the return channel p on which the final result of the invocation is expected at the invoker's side. Here, we create a fresh return channel r, before we pass on the whole request—where the result channel p is replaced with r—to the internal self. Thus, with respect to external requests, which need to serialize, we have now obtained complete local control on the internal self, until some result y of the current externally invoked method is coming back on r. Only then, we are allowed to proceed by forwarding the result y to the intended result channel p and by releasing a signal on the mutex channel m in order to let the next (external) request enter the object's implementation. If serialization is turned off (S=F), then the only obligations are to forward requests to the internal self and, which is important, to *immediately* signal back (on m) to the entry module that the next (external) request shall be allowed to enter.

Apart from mutually exclusive access to the internal self s_i, the serialization module also serves the purpose of authorization: every external request (received on s_e) that will be passed on to the object manager (on s_i), gets implanted the new identity s_i as its self parameter—the former 'current self' is discarded. By this mechanism, we will be able to distinguish between *authorized* external requests, which went through the official entrance s_e, from invalid external ones that somehow (see below) might have gotten direct access to the internal entrance s_i.

$$OB_{\mathbb{O}}\langle\, s_e, s_i \,\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu}\tilde{t})\,(\; MG_{\mathbb{O}}\langle\, s_e, s_i, \tilde{t} \,\rangle \;\mid\; \prod_{j\in J} !\, t_j(r,s,\tilde{x}).[\![b_j]\!]_r^s \;)$$

$$\begin{array}{rcl}
MG_{\mathbb{O}}\langle\, s_e, s_i, \tilde{t} \,\rangle & \stackrel{\text{def}}{=} & s_i(x).\mathsf{case}\ x\ \mathsf{of} \\
_(, s{\neq}s_i, *) & : & MG_{\mathbb{O}}\langle\, s_e, s_i, \tilde{t} \,\rangle \mid \overline{s_e}\langle x\rangle \\
\mathrm{l}_j_\mathsf{inv}_(r, s{=}s_i, \tilde{x}) & : & MG_{\mathbb{O}}\langle\, s_e, s_i, \tilde{t} \,\rangle \mid \overline{t_j}\langle r, s_i, \tilde{x}\rangle \\
\mathrm{l}_j_\mathsf{upd}_(r, s{=}s_i, t') & : & MG_{\mathbb{O}}\langle\, s_e, s_i, t_1 \ldots t_{j-1}, t', t_{j+1} \ldots t_n \,\rangle \mid \overline{r}\langle s_e\rangle \\
\mathsf{ali}_(r, s{=}s_i, y) & : & MG^A_{\mathbb{O}}\langle\, s_e, s_i, y \,\rangle \mid \overline{r}\langle s_e\rangle \\
\mathsf{cln}_(r, s{=}s_i) & : & MG_{\mathbb{O}}\langle\, s_e, s_i, \tilde{t} \,\rangle \;\big|\; (\boldsymbol{\nu} s'_e s'_i)\,\big(\, \overline{r}\langle s'_e\rangle \mid PP^{\mathsf{P},\mathsf{S}}_{\mathbb{O}}\langle\, s'_e, s'_i \,\rangle \mid MG_{\mathbb{O}}\langle\, s'_e, s'_i, \tilde{t} \,\rangle \;\big) \\
\mathsf{sur}_(r, s{=}s_i) & : & MG_{\mathbb{O}}\langle\, s_e, s_i, \tilde{t} \,\rangle \mid [\![\ \mathsf{let}\ n = s_i.\mathsf{clone}\ \mathsf{in}\ s_i.\mathsf{alias}(n); n\ ]\!]_r^{s_i}
\end{array}$$

$$\begin{array}{rcl}
MG^A_{\mathbb{O}}\langle\, s_e, s_i, s_a \,\rangle & \stackrel{\text{def}}{=} & s_i(x).\mathsf{case}\ x\ \mathsf{of} \\
_(, s{\neq}s_i, *) & : & MG^A_{\mathbb{O}}\langle\, s_e, s_i, s_a \,\rangle \mid \overline{s_e}\langle x\rangle \\
\mathrm{l}_j_\mathsf{inv}_(r, s{=}s_i, \tilde{x}) & : & MG^A_{\mathbb{O}}\langle\, s_e, s_i, s_a \,\rangle \mid \overline{s_a}\langle x\rangle \\
\mathrm{l}_j_\mathsf{upd}_(r, s{=}s_i, t') & : & MG^A_{\mathbb{O}}\langle\, s_e, s_i, s_a \,\rangle \mid \overline{s_a}\langle x\rangle \\
\mathsf{ali}_(r, s{=}s_i, y) & : & MG^A_{\mathbb{O}}\langle\, s_e, s_i, y \,\rangle \mid \overline{r}\langle s_e\rangle \\
\mathsf{cln}_(r, s{=}s_i) & : & MG^A_{\mathbb{O}}\langle\, s_e, s_i, s_a \,\rangle \;\big|\; (\boldsymbol{\nu} s'_e s'_i)\,\big(\, \overline{r}\langle s'_e\rangle \mid PP^{\mathsf{P},\mathsf{S}}_{\mathbb{O}}\langle\, s'_e, s'_i \,\rangle \mid MG^A_{\mathbb{O}}\langle\, s'_e, s'_i, s_a \,\rangle \;\big) \\
\mathsf{sur}_(r, s{=}s_i) & : & MG^A_{\mathbb{O}}\langle\, s_e, s_i, s_a \,\rangle \mid \overline{s_a}\langle x\rangle
\end{array}$$

Figure 3: Encoding of Øjeblik-objects: Object manager

Managing The object manager, receiving on the internal self, is the core administrator of an object's functionality, which is either invoked after successful pre-processing for external requests or directly due to internal requests of the aforementioned forms. So, object managers in particular need to deal correctly with self-infliction. To this end, we use an abbreviation that combines the case-construct with a built-in π-calculus matching operator for names. The crux for handling self-infliction resides in the matching for the self parameter: if a request reaches the object on the internal self channel s_i carrying a self parameter different from s_i, then the request is regarded as originating from a different thread of control and, thus, is forwarded to the external self s_e in order to run through the serialization and protection mechanisms. Note that this is necessary for the case '*forked extrusion*' of the internal self s_i, which is explicitly allowed, but needs special treatment: blocking if the extruder waits on the forked results involving invocation of s_i (cf. [Car95]). Remember also that external requests that have successfully passed the pre-processing, are explicitly authorized such that they are not mistakenly re-forwarded to the external self.

The basic purpose of object managers is to invoke the appropriate instances of method bodies (case l_j_inv: activate the method body b_j bound to l_j along trigger name t_j; Figure 3

$$
\begin{array}{rcl}
[\![a.\mathrm{l}_j\langle\, a_1 \ldots a_n \,\rangle]\!]^{s_c}_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\,(\boldsymbol{\nu} q_1 \cdots q_n)\ \big(\ [\![a]\!]^{s_c}_q \mid \prod_{i=1\cdots n} [\![v_i]\!]^{s_c}_{q_i} \\
& & \qquad\qquad \mid\ q(y).q_1(x_1).\cdots.q_n(x_n). \\
& & \qquad\qquad\quad \overline{y}\,\mathrm{l}_j_\mathsf{inv}_\langle p, s_c, x_1 \ldots x_n\rangle\ \big) \\
[\![a.\mathrm{l}_j \Leftarrow \varsigma(s,\tilde{x})b]\!]^{s_c}_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\ \big(\ [\![a]\!]^{s_c}_q \\
& & \qquad \mid\ q(y)\,.\,(\boldsymbol{\nu} t)\,\big(\ !\,t(r,s,\tilde{x}).[\![b]\!]^{s}_r \\
& & \qquad\qquad \mid\ \overline{y}\,\mathrm{l}_j_\mathsf{upd}_\langle p, s_c, t\rangle\ \big)\ \big) \\
\hline
[\![a.\mathsf{alias}(b)]\!]^{s_c}_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q_x, q_y)\ \big(\ [\![a]\!]^{s_c}_{q_y} \mid [\![b]\!]^{s_c}_{q_x} \mid q_y(y)\,.\,q_x(x)\,.\,\overline{y}\,\mathsf{ali}_\langle p, s_c, x\rangle\ \big) \\
[\![a.\mathsf{clone}]\!]^{s_c}_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\ \big(\ [\![a]\!]^{s_c}_q \mid q(y)\,.\,\overline{y}\,\mathsf{cln}_\langle p, s_c\rangle\ \big) \\
[\![a.\mathsf{surrogate}]\!]^{s_c}_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\ \big(\ [\![a]\!]^{s_c}_q \mid q(y)\,.\,\overline{y}\,\mathsf{sur}_\langle p, s_c\rangle\ \big)
\end{array}
$$

Figure 4: Encoding of Øjeblik-clients

shows that method bodies are run in the context of the current self s and the result channel r), and to carefully administrate updating (case l_j_upd: install a new trigger name t') and aliasing (case ali) such that respective future invocations will be dealt with correctly.

We distinguish between aliased (MG^A) and non-aliased (MG) object managers: aliasing is encoded by starting an aliased object manager instead of re-starting the non-aliased one. The aliased manager makes explicit the forwarding of requests to the aliasing target s_a in all cases of invocation, updating, and surrogation, as required. Note that invalid requests (those where $s \neq s_i$) are still forwarded to the external self s_e.[5]

Cloning results in the restart of the current object manager in parallel with a freshly created copy that uses the same method bodies, i.e. the same access names $\tilde{t}$ to them, so only the MG/MG^A-part of a new OB-module is installed with the new self parameters.

Surrogation is translated as we specified earlier as a combined cloning and aliasing, so we have embedded the setting for our main problem in the encoding of Øjeblik.

Clients It is here that the encodings current-self parameter s_c is actually used, when clients pass it on together with their requests. In each case, the responsibility for returning a result on channel p is forwarded to the respective object manager. Furthermore, each of the following translations obeys the same idea: the involved expressions are evaluated at private locations, their results are grabbed and then used to forward low-level request to the corresponding object managers. We chose a parallel evaluation order for object and parameters, but the results are grabbed corresponding to CBV-order (leftmost-innermost).

[5] This could be changed to forward the request x to the aliasing target s_a instead of the external self s_e. It would be correct as long as we assume that we never create aliases between objects with different protection/serialization settings; the required pre-processing is then carried out by the aliasing target. This is always the case for surrogation; for other cases, it can be guaranteed by imposing it in a type system, as we do in the forthcoming long version of the current document. Obliq leaves this rather open.

$$
\begin{array}{rcl}
[\![\mathsf{let}\ x = a\ \mathsf{in}\ b]\!]_p^{s_c} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\ (\ [\![a]\!]_q^{s_c} \mid q(x)\,.\,[\![b]\!]_p^{s_c}\ ) \\
[\![x]\!]_p^{s_c} & \stackrel{\text{def}}{=} & \overline{p}\langle x\rangle \\
\hline
[\![\mathsf{fork}(a)]\!]_p^{s_c} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} r)\ (\ \overline{p}\langle r\rangle \mid [\![a]\!]_r^{\boldsymbol{\nu} s}\ ) \\
[\![\mathsf{join}(b)]\!]_p^{s_c} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\ (\ [\![b]\!]_q^{s_c} \mid q(r).r(y).\overline{p}\langle y\rangle\ )
\end{array}
$$

Figure 5: Encoding of miscellaneous Øjeblik-expressions

Miscellaneous Variables and the let-construct is encoded as usual (see [KS98]).

Fork-and-join To fork a thread means to create a new activity running in parallel with the current one(s). Since Obliq is making some subtle assumptions about the interplay between the current thread and the other threads, we need to express the concept of threads to some extent. It suffices to create a new unrelated self identifier upon thread creation and to implant it as the forked threads' current self. We use $[\![a]\!]_q^{\boldsymbol{\nu} s}$ to abbreviate $(\boldsymbol{\nu} s)\,([\![a]\!]_q^s)$. Note that a fork is never blocking: we immediately return a (linear) private name r, which in turn might eventually be used to get some result (the evaluation of the forked expression a) back from the forked thread. This is achieved by waiting on this implicit return channel, when executing the translation of join. Here, b is an expression that evaluates to some thread identifier—in the translation, the thread is identified by the private channel r that has been returned from the corresponding fork-expression. Consequently, we block the (continuation of the) join-statement until some result is coming back along r from the forked thread and use this result also as the result of the join-expression itself.

4 Current and future work

We are on the verge of proving that

- Our π-calculus semantics preserves typing.
- Our π-calculus semantics implements self-serialization.
- Surrogation in Øjeblik is safe ($[\![a.\mathsf{surrogate}]\!] \approx [\![a]\!]$) for a suitable $\approx$.

References

[Car95] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.

[KS98] J. Kleist and D. Sangiorgi. Imperative Objects and Mobile Processes. In *Proceedings of PROCOMET '98*. IFIP, 1998. To appear.

[MS98] M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In K. G. Larsen, ed, *Proceedings of ICALP '98*, volume 1443 of *LNCS*. Springer, July 1998. To appear.

Recent BRICS Notes Series Publications

NS-98-5 **Hans Hüttel and Uwe Nestmann, editors.** ***Proceedings of the Workshop on Semantics of Objects as Processes, SOAP '98,*** **(Aalborg, Denmark, July 18, 1998), June 1998. 50 pp.**

NS-98-4 **Tiziana Margaria and Bernhard Steffen, editors.** ***Proceedings of the International Workshop on Software Tools for Technology Transfer, STTT '98,*** **(Aalborg, Denmark, July 12–13, 1998), June 1998. 86 pp.**

NS-98-3 **Nils Klarlund and Anders Møller.** MONA ***Version 1.2 — User Manual*****. June 1998. 60 pp.**

NS-98-2 **Peter D. Mosses and Uffe H. Engberg, editors.** ***Proceedings of the Workshop on Applicability of Formal Methods, AFM '98,*** **(Aarhus, Denmark, June 2, 1998), June 1998. 94 pp.**

NS-98-1 **Olivier Danvy and Peter Dybjer, editors.** ***Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,*** **(Gothenburg, Sweden, May 8–9, 1998), May 1998.**

NS-97-1 **Mogens Nielsen and Wolfgang Thomas, editors.** ***Preliminary Proceedings of the Annual Conference of the European Association for Computer Science Logic, CSL '97*** **(Aarhus, Denmark, August 23–29, 1997), August 1997. vi+432 pp.**

NS-96-15 **CoFI.** ***CASL – The CoFI Algebraic Specification Language; Tentative Design: Language Summary*****. December 1996. 34 pp.**

NS-96-14 **Peter D. Mosses.** ***A Tutorial on Action Semantics*****. December 1996. 46 pp. Tutorial notes for FME '94 (Formal Methods Europe, Barcelona, 1994) and FME '96 (Formal Methods Europe, Oxford, 1996).**

NS-96-13 **Olivier Danvy, editor.** ***Proceedings of the Second ACM SIGPLAN Workshop on Continuations, CW '97*** **(ENS, Paris, France, 14 January, 1997), December 1996. 166 pp.**

NS-96-12 **Mandayam K. Srivas.** ***A Combined Approach to Hardware Verification: Proof-Checking, Rewriting with Decision Procedures and Model-Checking; Part II: Articles. BRICS Autumn School on Verification*****. October 1996. 56 pp.**