INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Two Semantics for a Language of Reactive Objects

Frédéric Boussinot , Cosimo Laneve

N° 2511

Mars 1995

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

Rapport de recherche

1995

ISSN 0249-6399

Two Semantics for a Language of Reactive Objects

Frédéric Boussinot , Cosimo Laneve

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet MEIJE

Rapport de recherche n° 2511 — Mars 1995 — 23 pages

Abstract: We are studying semantics of a small object-based language, with the following main characteristics: parallelism, dynamicity, high order parameters, notion of a global instant, and reactivity. We give formal semantics using two related formalisms , namely π-calculus and the so-called "Chemical Abstract Machine" (CHAM). These formalisms are both powerful enough to express dynamicity and high order parameters, but they give distinct insights into the language. In π-calculus, emphasis is put on communications through channels; on the other hand, in CHAM, emphasis is put on more abstract and general interactions between program parts. We finally prove adequacy of the π-calculus semantics w.r.t. the CHAM semantics.

Key-words: Parallelism, objects, reactive approach, pi-calculus, chemical abstract machine.

(Résumé : tsvp)

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Deux sémantiques pour un langage à objets réactifs

Résumé : On étudie la sémantique d'un petit langage "basé objets", ayant les principales caractéristiques suivantes : parallélisme, création dynamique, paramètres d'ordre supérieur, notion d'instant global et réactivité. La sémantique formelle du langage est donnée en utilisant deux formalismes : le π-calcul et l'approche par "machine chimique" (CHAM). Ces formalismes sont tous deux assez puissants pour exprimer la création dynamique et les paramètres d'ordre supérieur, mais ils donnent des visions différentes du noyau du langage. En π-calcul, l'accent est mis sur la communication à travers des canaux, tandis qu'en CHAM, il est mis sur les interactions de plus haut niveau entre les composants parallèles. On prouve finalement l'adéquation de la sémantique en π-calcul par rapport à celle en CHAM.

Mots-clé : Parallélisme, objets, approche réactive, pi-calcul, machine chimique.

Contents

1 Introduction

In this paper, we are studying semantics of a small object-based language, with the following main characteristics:

- Parallelism: object methods are executed concurrently when called; method calls are asynchronous: after a call, the called method and the caller executes in parallel.
- Dynamicity: objects are dynamically created, and new method instances can be dynamically added to objects.
- High order parameters: method names can be passed as arguments during calls. As in the *actor* model[1], this feature is essential to allow objects to synchronize when only asynchronous method calls are allowed.
- Instants: there is a notion of an instant global to all methods, and the same method cannot be executed more than once in the same instant; thus, the current instant is finished when all methods called during that instant have been executed once.
- Reactivity: when called, a method continues to execute, starting from the point where it was stopped at the previous call, and reaching a new control point, which is the starting point for the next call (at a later instant).

This language that we call from now $\mathcal{L}_{ro}$, is the kernel of a new formalism under design with the support of France Télécom[1].

Presence of instants, we call this the *reactive approach*, is the main novelty of the proposed language. To briefly illustrate the interest of such an approach, just consider two graphical objects O_1 and O_2 that we want to consider as a linked couple when moved. There is a very simple solution to link together the two objects: each object, on reception of a `move` order, send the same order to the other object. The well known problem is thus to avoid loops where for instance, O_1 send the `move` order to O_2, which in return, send back the same order to O_1, and so on. Notice that these loops do not exist anymore if the `move` method cannot be executed more than once during one instant: when receiving for the second time the `move` order, O_1 simply rejects it, as it has already moved. Thus, presence of instants allows to give a very simple and symmetrical solution to the problem. We will not give more justifications on the approach, as, in this paper, we focus on semantics.

To be able to give $\mathcal{L}_{ro}$ a formal semantics, we choose to use two related formalisms, namely π-calculus[4] and the so-called "Chemical Abstract Machine" (CHAM)[2]. These formalisms are both powerful enough to express dynamicity and high order parameters, but they give distinct insights into the language. In π-calculus, emphasis is put on communications through channels; on the contrary, in CHAM, emphasis is put on more abstract and general interactions between program parts.

[1] Marché d'étude France Télécom-CNET, 93 1B 141, #506

The paper has the following structure: in section 2 we describe the language $\mathcal{L}_{ro}$. Then, we give in section 3 a CHAM semantics of $\mathcal{L}_{ro}$. The π-calculus semantics of $\mathcal{L}_{ro}$ is described in section 4. Finally, we prove in section 5 adequacy of the π-calculus semantics w.r.t. the CHAM semantics.

2 The language

We first intuitively describe the language $\mathcal{L}_{ro}$, and then give syntax for it.

2.1 Intuition

An *object* defines data which are shared by all methods that are associated to it. *Methods* are parallel and concurrent treatments on objects data. *Systems* are made of objects that are run in parallel. There are three levels: system, objects in the system, and methods associated to objects, and the same notion of an *instant* goes through the three levels:

- Execution of a method is divided into instants: one can speak of the first instant of the method, of the second instant, and so on. The same method cannot be run several times during one instant.
- Execution of an object for the current instant is terminated when all its methods have finished their execution for that instant.
- Execution of a system for the current instant is terminated when all its objects have finished their execution for that instant.

Therefore, instants give systems a *global synchronizing mechanism*: the system, its objects, and their methods all run to the rhythm of the same global clock, and no component is free to take some advance on the others.

Method calls are *asynchronous*: the caller does not wait for the called method to terminate, to continue its execution. Parameters may be transmitted during calls; they may be of any sort and even be objects or methods.

Objects can be dynamically created. Methods can also be dynamically added to objects. New objects are created as *clones* of existing ones. A clone of an object O is a new object whose methods are copies of those of O.

There exists an *initial* objet, with only one method; this method is cyclicly executed by the system and each execution of it defines a new instant. During one instant, objects are created and methods are called; execution of called methods can cause creation of new objects and calls of new methods, that will be executed in the same way. The current instant is terminated when all called methods have terminated their execution for that instant. Then, the initial method is called another time, defining a new instant, and so on.

2.2 Syntax

One defines the abstract syntax (in BNF form) of the language $\mathcal{L}_{ro}$ of reactive objects in the following way:

Object and Method Declarations. Objects and method declarations have the following syntax:

$$D \quad ::= \quad \varepsilon \mid D \;\; D \mid \mathsf{object}(O, \widetilde{x_O}) \mid \mathsf{method}(O, m, \widetilde{x}) \; B \; \mathsf{end}$$

The declaration $\mathsf{object}(O, \widetilde{x})$ introduces an object named O, having $\widetilde{x}$ as t-uple of fields.

The declaration $\mathsf{method}(O, m, \widetilde{a}) \;\; B\,\mathsf{end}$ introduces a method named m, associated to the object O, having $\widetilde{a}$ as list of formal parameters, and B as body.

We assume that in D never two objects have the same name O and never two methods have the same name m.

Method Bodies. A method body is made of a list of variable declarations C followed by a statement S:

$$B \quad ::= \quad C \;\; S$$

$$C \quad ::= \quad \varepsilon \mid \mathsf{var}\; x = V \; ; \; C$$

It is assumed that the scope of a variable inside a method is exactly the body of the method where it appears (variables are local to methods).

Programs. A program P of $\mathcal{L}_{ro}$ is made of a list D of object and method declarations, followed by the body B of the *initial method*:

$$P \quad ::= \quad D \;\; B$$

Statements. A statement S is defined by the following syntax:

$$\begin{array}{lll} S & ::= & \mathsf{stop} \mid \mathsf{stop}\,; S \mid \mathsf{clone}(O, O')\,; S \mid \mathsf{add}(O, m, P)\,; S \mid \mathsf{send}(O, P, \widetilde{d})\,; S \mid \\ & & \mathsf{field}(x) \; := \; V\,; S \;\; \mid x \; := \; V\,; S \mid \mathsf{if}\; B \;\mathsf{then}\; S \;\mathsf{else}\; S \; ; \; S \mid \mathsf{while}\; B \;\mathsf{do}\; S \;\mathsf{od}\; ; \; S \end{array}$$

Statement stop stops the execution for the current instant.

Statement $\mathsf{clone}(O, O')$ creates a new object O', which is a clone of object O. O' get a new fresh copy of all the method O owns at the moment the statement is executed. Names of methods of O' are inherited from those of O; fields of O' are undefined.

Statement $\mathsf{add}(O, m, P)$ adds to object O a new fresh copy of method m named P.

Statement $\mathsf{send}(O, P, \widetilde{d})$ send the order to execute method known as P in O, with parameters $\widetilde{d}$.

Statement $\mathsf{field}(x) \ := \ V$ assigns the value of the expression V to the field named x belonging to the current object O to which the method is attached. Correspondingly, $x \ := \ V$ assigns value of V to the local variable x. Remark that V is computed into the environment composed of the local variables of the method, its formal parameters, and the fields of O.

Statements $\mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ S$ and $\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}$ are boolean test and loop.

3 The CHAM Semantics

A CHAM[2] *solution* is a finite (multi-) set of items named *molecules*, that are terms built according to a syntax. Solutions are of the form: $\{\!|m_1, ..., m_k|\!\}$. The *chemical rule* is the basic rule; it says that molecule reactions can be performed freely within any solution (there is no way to "inhibits" a reaction). Moreover, some transformations of solutions are reversible (notation: $\rightleftharpoons$).

The CHAM semantics of $\mathcal{L}_{ro}$ consists in defining molecules accordingly to the syntax, and rules expressing how these molecules reacts.

3.1 Declarations

Declaration composition. Each declaration can be added to, or extracted from a solution:

$$D\,D \quad \rightleftharpoons \quad D\,,D$$

Object Declarations. The molecule associated to an object O is a triple of the form $\langle O',\ \varphi,\ f\rangle$ where:

- O' is the name of the basic object of O, that is O itself if the molecule corresponds to an object declaration, or the object that has been copied from, if the molecule corresponds to a clone operation. (O' is used for type checking purposes only).
- φ is a function that associate values to O fields. We note $\perp_{\widetilde{x}}$ the function that associates the undefined value to fields $\widetilde{x}$.
- f is a function that associates methods to names; more precisely, $f(P) = (m, p)$ means that a copy p of method m is known in O as P. We note $\perp$ the function that is undefined for all names.

Declaration of an object O with fields $\widetilde{x}$ creates a new molecule whose name is also O. The rule for object declaration is:

$$\mathsf{object}(O, \widetilde{x}) \quad \rightarrow \quad O :: \langle O,\, \perp_{\widetilde{x}},\, \perp \rangle$$

Method Declarations. Molecules associated to methods declarations are triple of the form $\langle O,\, S,\, \widetilde{a}\rangle$ where O is the object the method is belonging to, S is the method body, and $\widetilde{a}$ are the formal parameters. Declaration of a method m creates a new molecule whose name is also m. The rule for method declaration is:

$$\mathsf{method}(O, m, \widetilde{a})\ B\ \mathsf{end} \quad \rightarrow \quad m :: \langle O,\, B,\, \widetilde{a}\rangle$$

3.2 Statements

Executable molecules have the following form: $\langle S, O, \rho, \sigma\rangle$ where S is a statement which is the body of a method, O is the object to which the method belongs, ρ is a function that associates values to the method formal parameters, and σ is a function that gives value for the method local variables.

3.2.1 Programs

A program "$D\ C\ S$" creates two new molecules. The first one is definitions D; the second one is the 4-uple "$\langle S, \mathrm{INIT},\, \perp_{\emptyset},\, \sigma_C\rangle$", where we note σ_C is the store obtained from C:

$$D\ C\ S \quad \rightleftharpoons \quad D\ ,\, p_{\mathrm{INIT}} :: \langle S, \mathrm{INIT},\, \perp_{\emptyset},\, \sigma_C\rangle$$

Remark that the initial method is marked by the process name p_{INIT}, and belongs to the initial object, also called INIT.

3.2.2 Instants

Initial method. When the initial method is terminated for the current instant (body of the form "$\mathsf{stop}\,;\, S$"), a new molecule is created; this new molecule is a set, called "termination set", and the initial method is put into it:

$$p_{\mathrm{INIT}} :: \langle \mathsf{stop}\,;\, S, O,\, \rho,\, \sigma\rangle \quad \rightarrow \quad \mathsf{stop}\cdot\{p_{\mathrm{INIT}} :: \langle S, O,\, \rho,\, \sigma\rangle\}$$

Next instant. When the solution is completely absorbed into the termination set ("$\mathsf{stop}\cdot\mathcal{S}$"), one can go to the next instant, by simply removing the stop:

$$\{\!|\ \mathsf{stop}\cdot\mathcal{S}\ |\!\} \quad \rightarrow \quad \{\!|\ \mathcal{S}\ |\!\}$$

Absorption of objects and methods. Let H be any molecule of the form $m :: \langle O, C; S, \widetilde{a}\rangle$ or $O :: \langle O, \varphi, f\rangle$: Then H can be absorbed into the termination set:

$$\mathsf{stop} \cdot \mathcal{S}\ ,\ H \quad \rightarrow \quad \mathsf{stop} \cdot (\mathcal{S} \cup \{H\})$$

Absorption of processes. A process which has terminated to execute for the current instant ($\mathsf{stop}; S$), is absorbed into the termination set; but the code becomes guarded ($\mathsf{stop} \triangleright S$) to prevent the process to be automatically executed at the next instant. The $\triangleright$ operator means that a **send** statement must be performed before the process be executed:

$$\mathsf{stop} \cdot \mathcal{S}\ ,\ p :: \langle \mathsf{stop}; S, O,\ \rho,\ \sigma\rangle \quad \rightarrow \quad \mathsf{stop} \cdot (\mathcal{S} \cup \{p :: \langle \mathsf{stop} \triangleright S, O,\ \rho,\ \sigma\rangle\})$$

There is a special case for completely finished processes (of the form stop); these processes are absorbed into the termination set.

$$\mathsf{stop} \cdot \mathcal{S}\ ,\ p :: \langle \mathsf{stop}, O,\ \rho,\ \sigma\rangle \quad \rightarrow \quad \mathsf{stop} \cdot (\mathcal{S} \cup \{p :: \langle \mathsf{stop} \triangleright \mathsf{stop}, O,\ \rho,\ \sigma\rangle\})$$

A guarded process can be absorbed in the termination set. However, it can also be extracted from the termination set if needed:

$$\mathsf{stop} \cdot \mathcal{S}\ ,\ p :: \langle \mathsf{stop} \triangleright S, O,\ \rho,\ \sigma\rangle \quad \rightleftharpoons \quad \mathsf{stop} \cdot (\mathcal{S} \cup \{p :: \langle \mathsf{stop} \triangleright S, O,\ \rho,\ \sigma\rangle\})$$

Accepted Calls. To call a method, one simply catenate the arguments to the method (see 3.2.5). The call is accepted if the method is waiting to be called ($\mathsf{stop} \triangleright S$). Then, the call is performed by associating the arguments to the method formal parameters (we note $[\widetilde{x} \mapsto \widetilde{a}]$ the function that pointwise associates $\widetilde{a}$ to $\widetilde{x}$).

$$p :: \langle \mathsf{stop} \triangleright S, O,\ \rho_{\widetilde{x}},\ \sigma\rangle[\,\widetilde{a}\,] \quad \rightarrow \quad p :: \langle S, O,\ [\widetilde{x} \mapsto \widetilde{a}],\ \sigma\rangle$$

Refused Calls. A call is refused if the called method is not waiting to be called (that is, the method has already been called during the current instant). Then, the arguments are simply thrown away.

$$p :: \langle \alpha; S, O,\ \rho,\ \sigma\rangle[\,\widetilde{a}\,] \quad \rightarrow \quad p :: \langle \alpha; S, O,\ \rho,\ \sigma\rangle$$

Calls to completely finished processes are also systematically rejected.

$$p :: \langle \mathsf{stop}, O,\ \rho,\ \sigma\rangle[\,\widetilde{a}\,] \quad \rightarrow \quad p :: \langle \mathsf{stop}, O,\ \rho,\ \sigma\rangle$$

3.2.3 Clones

Assume that $\mathsf{dom}(f_O) = \{P_1, \cdots, P_k\}$ and $f(P_i) = (m_i, p_i)$. One defines $f_{O^\sharp}$ such that $f_{O^\sharp}(P_i) = (m_i, p'_i)$, where p'_i are fresh names. Let also σ_{C_i} be the store created by the declaration C_i. Then:

$$p :: \langle \mathsf{clone}(O, O^\sharp)\,;\, S, O'',\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi_{\widetilde{x}},\, f_O\rangle \;,\; m_1 :: \langle O',\, C_1\ S_1,\, \widetilde{a_1}\rangle \;,\; \cdots \;,\; m_k :: \langle O',\, C_k\ S_k,\, \widetilde{a_k}\rangle$$

$$\downarrow$$

$$\begin{array}{c} p :: \langle S, O'',\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi_{\widetilde{x}},\, f_O\rangle \;,\; m_1 :: \langle O',\, C_1\ S_1,\, \widetilde{a_1}\rangle \;,\; \cdots \;,\; m_k :: \langle O',\, C_k\ S_k,\, \widetilde{a_k}\rangle \\ O^\sharp :: \langle O',\, \bot_{\widetilde{x}},\, f_{O^\sharp}\rangle \;,\; p'_1 :: \langle \mathsf{stop} \triangleright S_1, O^\sharp,\, \bot_{\widetilde{a_1}},\, \sigma_{C_1}\rangle \;,\; \cdots \;,\; p'_k :: \langle \mathsf{stop} \triangleright S_k, O^\sharp,\, \bot_{\widetilde{a_k}},\, \sigma_{C_k}\rangle \end{array}$$

3.2.4 Adding a method

Let f be a function; $f[x \mapsto a]$ is the function defined by:

$$f[x \mapsto a](y) = \begin{cases} a & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}$$

Then:

$$p :: \langle \mathsf{add}(O, m, P)\,;\; S, O'',\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi,\, f\rangle \;,\; m :: \langle O',\, C\ S',\, \widetilde{a}\rangle$$

$$\downarrow$$

$$p ::\ \langle S, O'',\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi,\, f[P \mapsto (m, p')]\rangle \;,\; m :: \langle O',\, C\ S',\, \widetilde{a}\rangle \;,\; p' :: \langle \mathsf{stop} \triangleright S', O,\, \bot_{\widetilde{a}},\, \sigma_C\rangle$$

where p' is a fresh name.

3.2.5 Calls

To call a method, one simply catenate the arguments to the called method. Assume that $f(P) = (m, p')$ then:

$$p :: \langle \mathsf{send}(O, P, \widetilde{d})\,;\; S, O'',\, \rho,\, \sigma\rangle,\; O :: \langle O',\, \varphi,\, f\rangle \;,\; p' :: \langle S, O,\, \rho',\, \sigma'\rangle$$

$$\downarrow$$

$$p ::\ \langle S, O'',\, \rho,\, \sigma\rangle \;\;,\; O :: \langle O',\, \varphi,\, f\rangle \;,\; p' :: \langle S, O,\, \rho',\, \sigma'\rangle[[\![\widetilde{d}]\!]]$$

3.2.6 Values

In a method body, evaluation of expressions may depend on three domains: the method variables σ, the object's fields $\varphi_{\widetilde{x}}$, and the method's parameters $\rho_{\widetilde{y}}$. We note $[\![V]\!](\varphi_{\widetilde{x}}, \rho_{\widetilde{y}}, \sigma)$ the value of V according to these domains.

Fields.

$$p :: \langle \mathsf{field}(y) := V\,;\; S, O,\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi,\, f\rangle \;\rightarrow\; p :: \langle S, O,\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi[y \mapsto [\![V]\!](\varphi, \rho, \sigma)],\, f\rangle$$

Assignments.

$$p :: \langle z := V \,;\, S, O,\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi,\, f\rangle \;\rightarrow\;\rightarrow\; p :: \langle S, O,\, \rho,\, \sigma[z \mapsto [\![V]\!](\varphi,\rho,\sigma)]\rangle$$

Tests.

$$\begin{array}{c} p :: \langle \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \,;\, S, O,\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi,\, f\rangle \;\rightarrow\; p :: \langle S_1; S, O,\, \rho,\, \sigma\rangle \\ \mathsf{if}\ [\![B]\!](\varphi,\rho,\sigma) = \mathsf{true} \\ p :: \langle \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \,;\, S, O,\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi,\, f\rangle \;\rightarrow\; p :: \langle S_2; S, O,\, \rho,\, \sigma\rangle \\ \mathsf{if}\ [\![B]\!](\varphi,\rho,\sigma) = \mathsf{false} \end{array}$$

Loops.

$$\begin{array}{c} p :: \langle \mathsf{while}\ B\ \mathsf{do}\ S'\ \mathsf{od} \,;\, S, O,\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi,\, f\rangle \;\rightarrow\; p :: \langle S'; \mathsf{while}\ B\ \mathsf{do}\ S'\ \mathsf{od} \,;\, S, O,\, \rho,\, \sigma\rangle \\ \mathsf{if}\ [\![B]\!](\varphi,\rho,\sigma) = \mathsf{true} \\ p :: \langle \mathsf{while}\ B\ \mathsf{do}\ S'\ \mathsf{od} \,;\, S, O,\, \rho,\, \sigma\rangle \;,\; O :: \langle O',\, \varphi,\, f\rangle \;\rightarrow\; p :: \langle S, O,\, \rho,\, \sigma\rangle \\ \mathsf{if}\ [\![B]\!](\varphi,\rho,\sigma) = \mathsf{false} \end{array}$$

4 The π-Calculus Semantics

The π-calculus we use is defined in appendix A. In this section, we give the semantics of $\mathcal{L}_{ro}$ by means of a semantics function $\mathcal{S}$ which we are going to describe now.

4.1 Objects

Declarations. The semantics function $\mathcal{S}$ is defined for objects declarations in the following way:

For each declaration of an object O, three interface channels a_O, s_O, c_O are created: a_O is used to add methods to O, s_O to send method execution orders and c_O to clone O.

$$\mathcal{S}(\mathsf{object}(O,\, \widetilde{x_O}); D; B) \quad = \quad (a_O, s_O, c_O)\,(\; \mathbf{OBJ}_{\widetilde{x_O}}(a_O, s_O, c_O) \mid \mathcal{S}(D; B)\;)$$

The process **OBJ** which triggers objects is made of five parts put in parallel.

- Methods added to O are processed by **ADD**.
- The function that associates O methods to names, is implemented by **STORE**.
- Calls to O methods are processed by **SEND**.
- Clone orders are processed by **CLONE**.
- Objects fields are processed by **DATA**.

Three channels *wr*, *vl*, and *cp* are created: *wr* is used to register a new method in **STORE**; *vl* is used to get the internal name of a called method; *cp* is used to get all methods names when a cloning operation is to be performed.

$$\begin{aligned}\mathbf{OBJ}_{\widetilde{x_O}}(a_O, s_O, c_O) \quad = \quad (wr, vl, cp) \quad (\; & \mathbf{ADD}(a_O, wr) \\ & \mid \mathbf{STORE}(wr, vl, cp) \\ & \mid \mathbf{SEND}(s_O, vl) \\ & \mid \mathbf{CLONE}_{\widetilde{x_O}}(c_O, cp) \\ & \mid \mathbf{DATA}(\widetilde{x_O}) \\ ) \; & \end{aligned}$$

4.1.1 Adding a Method

The name m with its external name x are received with each add order a_O. Then, a new internal name p is created and the triple x, m, p is sent to **STORE**, to be registered. Finally, p is sent to method m, to create a new instance of method m, known internally as p.

$$\mathbf{ADD}(a_O, wr) \;\; = \;\; a_O(x, m).\, \overline{wr}[x, m, (p)].\; (\; \overline{m}[p] \mid \mathbf{ADD}(a_O, wr)\;)$$

The **STORE** process is described in appendix B, and the **DATA** process definition is left as an exercise.

4.1.2 Calls

Three parameters x, $\widetilde{d}$, and *sync* are received with each send order s_O: x is the external name of the method called, $\widetilde{d}$ are the arguments, and *sync* is a channel used for synchronization (see 4.2.2). Then, the internal name p associated to x is asked to **STORE** through channel *vl*; the reply is got from a new channel *ct*. Finally, $\widetilde{d}$ and *sync* are sent to p, for the call to be performed.

$$\mathbf{SEND}(s_O, vl) \; = \; s_O(x, \widetilde{d}, sync).\, \overline{vl}[x, (ct)].\, ct(x, m, p).\, \overline{p}[\widetilde{d}, sync].\, \mathbf{SEND}(s_O, vl)$$

4.1.3 Clones

Four parameters are associated to a cloning order c_O sent to O: $a_{O'}, s_{O'}, c_{O'}$ are interface channels for the new object O', and t is used to indicate the termination of the cloning process (when all method have been added in O'). Then, the new object O' is created (with the same fields list), and in parallel, the order is given on channel *cp*, to the **STORE** of O, to give its current methods. For that purpose, two channels *ans* and *end* are created: *ans* is used to receive the list items and *end* to indicate the end of the list.

$$\begin{aligned}\mathbf{CLONE}_{\widetilde{x_O}}(c_O, cp) \; = \; & c_O(a_{O'}, s_{O'}, c_{O'}, t).\; (\; \mathbf{OBJ}_{\widetilde{x_O}}(a_{O'}, s_{O'}, c_{O'}) \\ & \mid \; \overline{cp}[(ans), (end)].\, \mathbf{CONT}_{\widetilde{x_O}}(ans, end, t, a_{O'}, c_O, cp)\;)\end{aligned}$$

For each reply of **STORE** through *ans*, the corresponding method is added to O' through $a_{O'}$. When the list is exhausted, **STORE** send *end* and thus, the t channel can be sent, as the cloning process is terminated; then, the **CLONE** process is recursively re-executed.

$$\begin{array}{rcl}\mathbf{CONT}_{\widetilde{x_O}}(ans, end, t, a_{O'}, c_O, cp) & = & ans(x, m, p).\,\overline{a_{O'}}[x, m].\,\mathbf{CONT}_{\widetilde{x_O}}(ans, t, a_{O'}, c_O, cp) \\ & & +\ end[].\,\overline{t}[].\,\mathbf{CLONE}_{\widetilde{x_O}}(c_O, cp)\end{array}$$

4.2 Methods

For each declaration of a method m a channel of the same name is created; it is used to create new instances of the method.

$$\mathcal{S}(\mathsf{method}(O, m, \widetilde{x})\ B\ \mathsf{end}\ ;\ D\ C\ S) \quad = \quad (m)\,(\,\mathbf{MTD}_B(m, \widetilde{x}) \mid \mathcal{S}(D\ C\ S)\,)$$

4.2.1 Parameters

For each method formal parameter, one creates a cell which stores its value (initially undefined).

$$\begin{array}{rcl}\mathbf{MTD}_B(m, x_1, \cdots, x_k) & = & m(p).\,(\,\mathbf{MTD}_B(m, x_1, \cdots, x_k) \\ & & \mid (\cdots, r_{x_i}, w_{x_i}, \cdots)(\,\mathcal{S}^p_{x_1, \cdots, x_k}(B) \mid \cdots \mid \mathbf{CELL}(r_{x_i}, w_{x_i}, \perp) \mid \cdots\,)\end{array}$$

Definition of cell is standard:

$$\mathbf{CELL}(r_x, w_x, d) = \overline{r_x}[d].\,\mathbf{CELL}(r_x, w_x, d)\ +\ w_x(d').\,\mathbf{CELL}(r_x, w_x, d')$$

4.2.2 Bodies

The function $\mathcal{S}^p_{\widetilde{x}}$ is defined on method bodies in the following way:

A cell is created to hold the value of each local variable.

$$\mathcal{S}^p_{\widetilde{x}}(\mathsf{var}\ x = V\ ;\ C\ S) \quad = \quad (r_x, w_x)\,(\,\mathbf{CELL}(r_x, w_x, [\![V]\!]) \mid \mathcal{S}^p_{\widetilde{x}}(C\ S)\,)$$

We leave undefined the process $[\![V]\!]$. Informally it computes the value V in the environment consisting of the local data of the method, plus the parameters with which the method is called, and the data of the object to which the method is attached.

For the statement S, two channels *stop* and *app* are first created:

- *app* is used to get information about calls performed during execution of the method: a synchronizing channel is send on *app* each time a `send` statement is executed.

- *stop* is emitted by the method when it terminates to execute for the current instant, that is when it reaches a `stop` statement. Moreover, it is also used to control execution of the method: the method waits for a *stop* signal to start execution.

Then, two processes are put in parallel:

- $[\![S]\!]^{stop}_{app}$ is the translation of statement S, defined in 4.3.
- **SYNC** performs the synchronization for instant processing. The algorithm for distributed termination used to synchronize all called methods on the end of instant, procceeds as follows: when it terminates the current instant, each method send a first "near termination" *sync* signal to its caller; then, the method waits for the termination of all method called by it; finally, the it sends a second "true termination" *sync* to its caller.

$$\mathcal{S}^{p}_{\widetilde{x}}(S) = (stop, app)\ (\ stop().\,[\![S]\!]^{stop}_{app}\ \mid \mathbf{SYNC}_{\widetilde{x}}(p, app, stop)\ )$$

SYNC Definition. **SYNC** first waits for the method internal name p; when received, it send write orders to cells that hold parameters, and it send the *stop* signal to start execution of the method body. Then, two processes **EG** and **LIST** are put in parallel.

$$\begin{aligned}\mathbf{SYNC}_{x_1,\cdots,x_k}(p, app, stop) \ = \ & p(d_1,\cdots,d_k, sync).\,\overline{w_{x_1}}[d_1].\,\cdots.\,\overline{w_{x_k}}[d_k].\,\overline{stop}[].\\ & (eg)(\ \mathbf{EG}(p, eg) \mid (\ell,\ell')(\ \mathbf{LIST}_{x_1,\cdots,x_k}(\ell,\ell',p,app,stop,sync,eg) \mid \ell().\,\overline{\ell'}[])\ )\end{aligned}$$

EG Definition. **EG** rejects all calls to the method until the next instant; Definition of **EG** is straight forward. Notice however that two emissions of *sync* are performed to assure that **LIST** will not be blocked, as in the synchronization of `stop`, methods whose requests have been rejected are also involved.

$$\mathbf{EG}(p, eg) \ = \ eg() \ + \ p(\widetilde{d}, sync).\ (\ \overline{sync}[].\,\overline{sync}[] \mid \mathbf{EG}(p, eg)\ )$$

LIST Definition. **LIST** stores in a list all the *sync* channels of called methods. When a *stop* signal is received from the method, **LIST** begins to emit its own "near termination" *sync* signal, and waits for the "near termination" *sync*, then for the "true termination" *sync* from all channels stored in the list; finally, when all these receptions have been performed, the **EG** part is reset and the "true termination" *sync* signal is emitted. Definition of **LIST** implementing a kind of dynamic list, is as follows:

$$\begin{array}{rcl}\mathbf{LIST}_{\widetilde{x}}(\ell,\ell',p,app,stop,sync,eg) & = & app(s).\,(\ell_1,\ell_1')\,(\,\mathbf{LIST}_{\widetilde{x}}(\ell_1,\ell_1',p,app,stop,sync,eg)\\ & & \qquad\qquad\quad \mid \ell().\,(\,\overline{\ell_1}[] \mid s().\,\ell_1'().\,(\,\overline{\ell'}[] \mid s()\,)\,)\\ & & \qquad\qquad\quad)\\ & & +\ stop().\,(\,\overline{sync}[] \mid \overline{\ell}[].\,\ell'().\,(\,\overline{eg}[] \mid \overline{sync}[].\,\mathbf{SYNC}_{\widetilde{x}}(p,app,stop)\,)\,)\end{array}$$

4.2.3 Initial Body

For the initial method body, $\mathcal{S}$ definition is:

$$\mathcal{S}(\mathsf{var}\ x = V\ ;\ C\ S) \quad = \quad (r_x, w_x)\,(\,\mathbf{CELL}(r_x, w_x, [\![V]\!]) \mid \mathcal{S}(C\ S)\,)$$

$$\mathcal{S}(S) \quad = \quad (stop, app)\,(\,[\![S]\!]^{stop}_{app} \ \mid\ \mathbf{LIST}(app, stop)\,)$$

The process **LIST** used for encoding the initial method is a simplified version of $\mathbf{LIST}_{\widetilde{x}}$ (there is no caller for the initial method).

$$\mathbf{LIST}(app, stop) \ = \ (\ell,\ell')(\,\mathbf{LIST}'(\ell,\ell',app,stop) \mid \ell().\,\overline{\ell'}[]\,)$$

$$\begin{array}{rcl}\mathbf{LIST}'(\ell,\ell',app,stop) & = & app(s).\,(\ell_1,\ell_1')\,(\,\mathbf{LIST}'(\ell_1,\ell_1',app,stop)\\ & & \qquad\qquad\quad \mid \ell().\,(\,\overline{\ell_1}[] \mid s().\,\ell_1'().\,(\,\overline{\ell'}[] \mid s()\,)\,)\\ & & \qquad\qquad\quad)\\ & & +\ stop().\,\overline{\ell}[].\,\ell'().\,\overline{stop}[].\,\mathbf{LIST}(app,stop)\end{array}$$

The following proposition shows that we have implemented what was expected for **LIST** processes.

Proposition 4.1 *1.* $(app, stop)(\,\mathbf{LIST}(app, stop) \mid \overline{app}[s_1].\,\cdots.\,\overline{app}[s_k].\,\overline{stop}[]\,) \stackrel{\theta_1}{\Longrightarrow}\stackrel{\theta_2}{\Longrightarrow}$, *where* θ_1 *and* θ_2 *are permutations of* $(s_1,\cdots,s_k)$*;*

2. let $P = (\ell,\ell',app,stop)(\,\mathbf{LIST}_{\widetilde{x}}(\ell,\ell',p,app,stop,sync,eg) \mid \ell().\,\overline{\ell'}[] \mid \overline{app}[s_1].\,\cdots.\,\overline{app}[s_k].\,\overline{stop}[]\,)$. *Then* $P \stackrel{\theta_1}{\Longrightarrow}\stackrel{\theta_2}{\Longrightarrow}$, *where* θ_1 *is a permutation of* $(\overline{sync}, s_1,\cdots,s_k)$ *and* θ_2 *is a permutation of the t-uple* $(\overline{sync}, s_1,\cdots,s_k,\overline{eg})$

4.3 Statements

The format for translating a statement S is $[\![S]\!]^{stop}_{app}$, where *app* is the channel used to signal method calls, and *stop* is used to control execution and to signal the end of the method in which S appears.

Translation of a final stop (that is, followed by no statement), is simply the null process $\mathcal{O}$ of the π-calculus.

$$[\![\mathsf{stop}]\!]^{stop}_{app} \quad = \quad \mathcal{O}$$

For a stop followed by a statement S, one first send *stop* to signal the termination of the method for the current instant, then, waits for *stop* to continue at a later instant.

$$[\![\mathsf{stop}; S]\!]_{app}^{stop} \;=\; \overline{stop}[].\, stop().\; [\![S]\!]_{app}^{stop}$$

To clone an object O' from an object O, one creates three interface channel names $a_{O'}, s_{O'}, c_{O'}$ and a channel t used to synchronize, at the end of the cloning process, when all copies of O methods have been added to O'.

$$[\![\mathsf{clone}(O, O'); S]\!]_{app}^{stop} \;=\; \overline{c_O}[(a_{O'}), (s_{O'}), (c_{O'}), (t)]\,.\, t()\;.\; [\![S]\!]_{app}^{stop}$$

To add to O a method m with name P, one just send P, m on a_O.

$$[\![\mathsf{add}(O, m, P); S]\!]_{app}^{stop} \;=\; \overline{a_O}(P, m).\; [\![S]\!]_{app}^{stop}$$

To call method P of O with arguments $\widetilde{d}$, one creates a new channel *sync* and send P, the evaluation of the arguments $[\![\widetilde{d}]\!]$, and *sync* on s_O; *sync* is also sent on *app*.

$$[\![\mathsf{send}(O, P, \widetilde{d}); S]\!]_{app}^{stop} \;=\; \overline{s_O}(P, [\![\widetilde{d}]\!], (sync)).\; \overline{app}[sync].\, [\![S]\!]_{app}^{stop}$$

5 Adequacy of the π-Calculus Encoding

Let us show the adequacy of the π-calculus implementation w.r.t. the CHAM-semantics of $\mathcal{L}_{ro}$. Starting from (γ, ξ), where γ is a state of the CHAM transition system and ξ is a state of the π-calculus transition system, we shall prove that every CHAM-transition $\gamma \rightarrow \gamma'$ is mirrored by a sequence of π-calculus transitions $\xi \overset{\tau}{\longrightarrow}{}^{*} \xi'$. To this aim the CHAM-solution is too abstract to be put in correspondence with a π-calculus state. In particular the problem is due to the strategy used by the π-calculus implementation for coding the stop. Namely, exactly the processes which are active along one instant synchronize at the end of it (in the CHAM, all the molecules of the solution synchronize). Hence we consider couples made of a solution and a function μ, which associate to a process name p both the list of the process names that have called p, and the set of the process names that p has called.

Definition 5.1 *Let ξ be the proof of the* CHAM*-transition $\{\!|\mathcal{M}|\!\} \rightarrow \{\!|\mathcal{M}'|\!\}$ and let μ be a function from process names to a pair whose elements are lists of process* *names. Then* $\{\!|\mathcal{M}|\!\}\mu \rightarrow \{\!|\mathcal{M}'|\!\}\mu$ *if no instance of the rewriting rule* $\{\!|\mathsf{stop}\cdot\mathcal{S}|\!\} \rightarrow \{\!|\mathcal{S}|\!\}$ *or send rule of 3.2.5 appears in* ξ.

If the rewriting rule $\{\!|\mathsf{stop}\cdot\mathcal{S}|\!\} \rightarrow \{\!|\mathcal{S}|\!\}$ *appears in* ξ *then* $\{\!|\mathsf{stop}\cdot\mathcal{S}|\!\}\mu \rightarrow \{\!|\mathcal{S}|\!\}\emptyset$.

Otherwise, let p *be the process name of the process calling* p' *in the (unique) instance of rule 3.2.5 in* ξ *and let* μ' *be the function*

$$\mu'(x) = \begin{cases} (\mathsf{proj}_1(\mu(x)), \mathsf{proj}_2(\mu(x))\cdot p') & \text{if } x = p \\ (\mathsf{proj}_1(\mu(x))\cdot p, \mathsf{proj}_2(\mu(x))) & \text{if } x = p' \\ \mu(x) & \text{otherwise} \end{cases}$$

Then $\{\!|\mathcal{M}|\!\}\mu \rightarrow \{\!|\mathcal{M}'|\!\}\mu'$

In the definition below, if L is a list of names, we note $p \in L$ if the name p occurs as element of L.

Definition 5.2 *Let* $\mathcal{F}$ *be the function from* $\{\!|\mathcal{M}|\!\}\mu$ *to a* π*-calculus state such that*

$$\mathcal{F}(\{\!|m_1,\cdots,m_k|\!\}\mu) = (\Upsilon)(\ \mathcal{F}_\mu(m_1) \mid \cdots \mid \mathcal{F}_\mu(m_k)\ ) \qquad \Upsilon = \bigcup_{1\le i\le k}\mathsf{fn}(\mathcal{F}_\mu(m_i)$$

$$\mathcal{F}_\mu(\mathsf{stop}\cdot\{m_1,\cdots,p_{\mathrm{INIT}} :: \langle S,\mathrm{INIT},\rho,\sigma\rangle,\cdots,m_k\}) = \begin{array}{l}\mathcal{F}_\mu(m_1) \mid \cdots \\ \mid \mathcal{F}_\mu(p_{\mathrm{INIT}} :: \langle \mathsf{stop}; S,\mathrm{INIT},\rho,\sigma\rangle) \\ \mid \cdots \mid \mathcal{F}_\mu(m_k)\end{array}$$

and

$$\mathcal{F}_\mu(O :: \langle O',\varphi_{\widetilde{x_O}}, f_O\rangle) = \mathbf{OBJ}_{\widetilde{x_O}}(a_O,s_O,c_O)$$

where **DATA** *and* **STORE** *contain the same informations of* $\varphi_{\widetilde{x_O}}$ *and* f_O;

$$\mathcal{F}_\mu(m :: \langle O,C;S,\widetilde{x}\rangle) = \mathbf{MTD}_{C;S}(m,\widetilde{x})$$

$$\mathcal{F}_\mu(p_{\mathrm{INIT}} :: \langle S,\mathrm{INIT},\rho,\sigma\rangle) = (stop,app,\cdots,r_{x_i},w_{x_i},\cdots)(\ [\![S]\!]^{stop}_{app} \mid \cdots \mid \mathbf{CELL}(r_{x_i},w_{x_i},d_i) \mid \cdots \mid \\ \mid \mathbf{CALLED}(\mathsf{proj}_2(\mu(p_{\mathrm{INIT}})),app,stop) \\ )$$

where the variables x_i *are those of the store* σ *(namely the local variables of* p_{INIT}*) and* $\sigma(x_i) = d_i$*. Moreover, let* $\mathsf{proj}_2(\mu(p_{\mathrm{INIT}})) = p_1\cdot\ldots\cdot p_n$ *and* s_i *be the channel sent by* p_{INIT} *to* p_i *for the synchronization at the end of the instant.* $\mathbf{CALLED}(\mathsf{proj}_2(\mu(p_{\mathrm{INIT}})),app,stop)$ *is the process obtained by evaluating* $\mathbf{LIST}(app,stop) \mid \overline{app}[s_1].\ \cdots\ .\overline{app}[s_k]$ *(and always synchronizing over the channel app).*

Let $T = \alpha;S$ *or* $T = \mathsf{stop}$*. Then:*

$$\mathcal{F}_\mu(p :: \langle T,O,\rho,\sigma\rangle) \;=\; (stop,app,\cdots,r_{x_i},w_{x_i},\cdots)\ (\ [\![T]\!]^{stop}_{app} \mid \cdots \mid \mathbf{CELL}(r_{x_i},w_{x_i},d_i) \mid \cdots \mid \\ \mid \mathbf{CALLED}'(\mu(p),app,stop) \\ )$$

$$\mathcal{F}_\mu(p :: \langle \mathsf{stop} \triangleright S,O,\rho,\sigma\rangle) \;=\; (stop,app,\cdots,r_{x_i},w_{x_i},\cdots)\ (\ \overline{stop}[].\,stop().\,[\![S]\!]^{stop}_{app} \\ \cdots \mid \mathbf{CELL}(r_{x_i},w_{x_i},d_i)\ \cdots \\ \mid \mathbf{CALLED}'(\mu(p),app,stop) \\ )$$

where the variables x_i are those of the interface ρ and the store σ and if x_i is a variable of the interface, $\rho(x_i) = d_i$ otherwise $\sigma(x_i) = d_i$. Moreover, let $\mu(p) = (q_1 \cdot \ldots \cdot q_k,\, p_1 \cdot \ldots \cdot p_n)$ and s_i (resp. r_i) be the channel sent by q_i (resp. p) to p (resp. p_i) for the synchronization at the end of the instant. Then $\mathbf{CALLED}'(\mu(p), app, stop)$ is the process obtained by evaluating

$$\begin{array}{ll}(eg,\ell,\ell') & (\ \mathbf{LIST}_{\widetilde{x}}(\ell,\ell',p,app,stop,s_1,eg) \mid \overline{app}[r_1].\ \cdots\ .\overline{app}[r_n] \mid \\ & \mid \overline{s_2}[].\,\overline{s_2}[] \mid \cdots \mid \overline{s_k}[].\,\overline{s_k}[] \mid \mathbf{EG}(p,eg) \mid \ell().\,\overline{\ell'}[] \\ &)\end{array}$$

(and always synchronizing the communications concerning app)

$$\mathcal{F}_\mu(p :: \langle T,O,\rho,\sigma\rangle[\widetilde{d}]) = \mathcal{F}_\mu(p :: \langle T,O,\rho,\sigma\rangle) \qquad T = \alpha; S \text{ or } T = \mathsf{stop}$$

$$\begin{array}{ll}\mathcal{F}_\mu(p :: \langle \mathsf{stop} \rhd S,O,\rho,\sigma\rangle) = (stop, app, \cdots, r_{x_i}, w_{x_i}, \cdots) & (\ stop().\,[\![S]\!]^{stop}_{app} \\ & \cdots \mid \mathbf{CELL}(r_{x_i}, w_{x_i}, d_i) \cdots \\ & \mid \mathbf{SYNC}_{\widetilde{y}}(p, app, stop) \\ &)\end{array}$$

where x_i is a variable of the interface ($\widetilde{y}$) or of the local store σ. Their value is consistent with ρ and σ.

Finally $\mathcal{F}_\mu(p :: \langle \mathsf{stop} \rhd S,O,\rho,\sigma\rangle[\widetilde{d}])$ is the process obtained by evaluating $\mathcal{F}_\mu(p :: \langle \mathsf{stop} \rhd S,O,\rho,\sigma\rangle) \mid \overline{p}[\widetilde{d},s]$.

Theorem 5.3 (the adequacy theorem) *Assume that two different declarations in the solution $\{\!|\mathcal{M}|\!\}$ have different names. Then, $\{\!|\mathcal{M}|\!\}\mu \to \{\!|\mathcal{M}'|\!\}\mu'$ implies $\mathcal{F}(\{\!|\mathcal{M}|\!\}\mu) \xrightarrow{\tau}{}^{*} \mathcal{F}(\{\!|\mathcal{M}'|\!\}\mu')$.*

Proof: The theorem is proved by a case analysis on the reduction $\{\!|\mathcal{M}|\!\}\mu \to \{\!|\mathcal{M}'|\!\}\mu'$.

The cases when the reduction is due to a declaration is easy.

(add) Let $\mathcal{M}$ contain the three molecules: $p :: \langle \mathsf{add}(O,m,P);\ S,\, \rho,\, \sigma\rangle$, $O :: \langle O',\, \varphi_{\widetilde{x_O}},\, f_O\rangle$ and $m :: \langle O',\, C\ S',\, \widetilde{x'}\rangle$. Let $\{\!|\mathcal{M}|\!\}\mu \to \{\!|\mathcal{M}'|\!\}\mu'$ due to the evaluation of the add. Hence the above molecules of $\mathcal{M}$ are replaced in $\mathcal{M}'$ by

$$p :: \langle S,\, \rho,\, \sigma\rangle,\ O :: \langle O',\, \varphi_{\widetilde{x_O}},\, f_O[P \mapsto (m,p')]\rangle,\ m :: \langle O',\, C\ S',\, \widetilde{x'}\rangle,\ p' :: \langle \mathsf{stop} \rhd S', \zeta_{\widetilde{x}}, \sigma_C\rangle$$

Let us show that $\mathcal{F}(\{\!|\mathcal{M}|\!\}\mu) \xrightarrow{\tau}{}^{*} \mathcal{F}(\{\!|\mathcal{M}'|\!\}\mu)$ (remark that $\mu = \mu'$, in this case). To this aim we write the part of the term $\mathcal{F}(\{\!|\mathcal{M}|\!\}\mu)$ which is concerned by the reduction. Namely:

$$\begin{array}{l}\ldots\ a_O(x,m).\,\overline{wr}[x,m,(p')].\,(\ \overline{m}[p'] \mid \mathbf{ADD}(a_O, wr)\ ) \\ \mid \mathbf{STORE}(wr, vl, cp) \\ \mid m(p').\,(\ \mathbf{MTD}_{C;S}(m,\widetilde{y}) \mid (\cdots, r_{x_i}, w_{x_i}, \cdots)(\mathcal{S}^{p'}_{\widetilde{y}}(C\ S) \mid \cdots) \\ \mid \overline{a_O}[x,P].\,[\![S]\!]^{stop}_{app} \mid \cdots\end{array}$$

Due to the definition of **STORE**, it is clear that we obtain $\mathcal{F}(\{\!|\mathcal{M}'|\!\}\mu)$ by evaluating the above term.

(send) Let the rule $\{\!|\mathcal{M}|\!\}\mu \to \{\!|\mathcal{M}'|\!\}\mu'$ be due to the following transition

$$p :: \langle \mathsf{send}(O, P', \widetilde{d})\,;\ S,\, \rho,\, \sigma\rangle,\ O :: \langle O',\, \varphi_{\widetilde{x_O}},\, f_O\rangle\ ,\ p' :: \langle S',\, \rho_{\widetilde{x'}},\, \psi\rangle$$
$$\downarrow (f_O(P') = (m', p'))$$
$$p ::\ \langle S,\, \rho,\, \sigma\rangle\ \ ,\ O :: \langle O',\, \varphi_{\widetilde{x_O}},\, f_O\rangle\ ,\ p' :: \langle S',\, \rho_{\widetilde{x'}},\, \psi\rangle[\widetilde{d}]$$

Remark also that $\mu \neq \mu'$, since $\mu'(p) = (\mathsf{proj}_1(\mu(p)), \mathsf{proj}_2(\mu(p)) \cup \{p'\})$ and that $\mu'(p') = (\mathsf{proj}_1(\mu(p'))\cdot p, \mathsf{proj}_2(\mu(p)))$. Here is the part of the term $\mathcal{F}(\{\!|\mathcal{M}|\!\}\mu)$ concerned by the above reduction:

$$\begin{array}{l} \ldots\ s_O(P', \widetilde{a}, s).\, \overline{vl}[P, (ct)].\, ct(P', m, p').\, \overline{p'}[\widetilde{a}, (s)].\, \mathbf{SEND}(s_O, vl \\ \mid \mathbf{STORE}(wr, vl, cp) \mid \mathbf{LIST}_{\widetilde{x}}(\ell, \ell', p, app, stop, sync, eg) \\ \mid \overline{s_O}[P', [\![\widetilde{d}]\!], (s)].\, \overline{app}[s].\, [\![S]\!]^{stop}_{app} \mid \cdots \\ \mid \mathcal{F}_\mu(p' :: \langle S',\, \rho_{\widetilde{x'}},\, \psi\rangle) \end{array}$$

The output $\overline{p'}[\widetilde{a}, (s)]$ in the first line will synchronize with the dual input in $\mathcal{F}_\mu(p' :: \langle S',\, \rho_{\widetilde{x'}},\, \psi\rangle$. It depends on the body S', if the call $\overline{p'}[\widetilde{a}, (s)]$ is refused or accepted. In particular, if $S' = \alpha; S''$ (or $S' = \mathsf{stop}$), then

1. by definition of $\mathcal{F}_\mu$, $\overline{p'}[\widetilde{a}, (s)]$ will synchronize with an input in the process **EG** of $\mathcal{F}_\mu(p' :: \langle S',\, \rho_{\widetilde{x'}},\, \psi\rangle)$. This means that the call is refused and $\overline{s}[].\overline{s}[]$ is created in parallel with **EG**;

2. the output $\overline{app}[s]$ synchronizes with the dual statement in $\mathbf{LIST}_{\widetilde{x}}$, that is the number of processes with whom p synchronizes at the end of the instant is increased by p' (and the synchronizing channel will be s).

Remark that 1 and 2 are exactly the changes of μ' w.r.t. μ. We leave to the reader the check that $\mathcal{F}(\{\!|\mathcal{M}'|\!\}\mu')$ coincides with the resulting process.

If $S' = \mathsf{stop} \triangleright S''$ (and p' has not been called during the current instant, namely $\forall q.\, p' \notin \mathsf{proj}_2(\mu(q)))$ then $\overline{p'}[\widetilde{a}, (s)]$ is served by the subprocess $\mathbf{SYNC}_{\widetilde{x}}$ of $\mathcal{F}_\mu(p' :: \langle S,\, \rho_{\widetilde{x'}},\, \psi\rangle)$. This means that the call is accepted. The theorem can be proved as in the previous case.

(clone) Let $\{\!|\mathcal{M}|\!\}\mu \to \{\!|\mathcal{M}'|\!\}\mu'$ be due to the execution of a clone (remark that $\mu' = \mu$ in this case). Then $\{\!|\mathcal{M}|\!\}$ contains the molecule $p :: \langle \mathsf{clone}(O, O^\sharp); S,\, O',\, \rho,\, \sigma\rangle$, the definition of the object $O :: \langle O',\, \varphi_{\widetilde{x_O}},\, f_O\rangle$ and that of the methods $m_i :: \langle O',\, C_i; Si_i,\, \widetilde{y_i}\rangle$ used by O (and whose name is recorded in f_O). Then $\mathcal{F}(\{\!|\mathcal{M}|\!\}\mu)$ contains the process

$$\overline{c_O}[(a_{O^\sharp}), (s_{O^\sharp}), (c_{O^\sharp}), (t)].\, t().\, [\![S]\!]^{stop}_{app} \mid \mathbf{OBJ}_{\widetilde{x_O}}(a_O, s_O, c_O) \mid \cdots \mid \mathbf{MTD}_{C_i; S_i}(m_i, \widetilde{y_i}) \mid \cdots$$

The output $\overline{c_O}[(a_{O^\sharp}), (s_{O^\sharp}), (c_{O^\sharp}), (t)]$ synchronizes with the dual input in $\mathbf{CLONE}_{\widetilde{x_O}}(c_O, cp)$ in $\mathbf{OBJ}_{\widetilde{x_O}}$ which, in turn, triggers the request $\overline{cp}[(ans), (end)]$ for reading the store (namely the methods which are active for O). At the same time, the new object $\mathbf{OBJ}_{\widetilde{x_{O^\sharp}}}(a_{O^\sharp}, s_{O^\sharp}, c_{O^\sharp})$ is created (without any active method). The inheritance of the methods of O is performed as the subprocess **STORE** of $\mathbf{OBJ}_{\widetilde{x_O}}$ answers (on the channels ans and end). When the cloning terminates (i.e. communication over end), there happens the synchronization on the channel t between the process p and $\mathbf{CONT}_{\widetilde{x_O}}$. Hence, the resulting process is exactly equal to $\mathcal{F}(\{\!|\mathcal{M}'|\!\}\mu)$.

(stop) There is still one case: when $\mathcal{M} = \mathsf{stop} \cdot \mathcal{S}$. Then $\{\!|\mathsf{stop} \cdot \mathcal{S}|\!\}\mu \rightarrow \{\!|\mathcal{S}|\!\}\mu'$. According to the CHAM-semantics, the solution $\{\!|\mathsf{stop} \cdot \mathcal{S}|\!\}$ can be reached through coolings, provided that all the processes have terminated their instant (i.e. those that have been called have reached a stop). Remark that these operations of cooling do not change the term $\mathcal{F}(\mathcal{M}\mu)$. By definition of $\mathcal{F}$ and Proposition 4.1, if $\mu(p) = (q_1 \cdot \ldots \cdot q_k, p_1 \cdot \ldots \cdot p_n)$ and s_i is the synchronizing channel between p and q_i and r_i the one between p and p_i, then $\mathcal{F}(\mathcal{M}\mu)$ contains the subprocess

$$\begin{array}{lll} p = p_{\mathrm{INIT}} \;\; (\mathsf{proj}_1(\mu(p)) = \emptyset) & P & (\, P \overset{r_{\imath_1} \cdots r_{\imath_n}}{\Longrightarrow} \overset{r_{\jmath_1} \cdots r_{\jmath_n}}{\Longrightarrow} \,) \\ p \neq p_{\mathrm{INIT}} & \overline{s_2}[].\overline{s_2}[] \mid \cdots \mid \overline{s_k}[].\overline{s_k}[] \mid P & (\, P \overset{r_{\imath_1} \cdots r_{\imath_{n+1}}}{\Longrightarrow} \overset{r_{\jmath_1} \cdots r_{\jmath_{n+2}}}{\Longrightarrow} \,) \end{array}$$

where $r_{\imath_1} \cdots r_{\imath_n}$ and $r_{\jmath_1} \cdots r_{\jmath_n}$ are permutations of $(r_1, \cdots, r_n)$ and $r_{\imath_1} \cdots r_{\imath_{n+1}}$ is a permutation of the t-uple $(\overline{s_1}, r_1, \cdots, r_n)$ while $r_{\jmath_1} \cdots r_{\jmath_{n+2}}$ is a permutation of $(\overline{s_1}, r_1, \cdots, r_n, \overline{eg}_p)$.

It is a tedious check verifying that all such communications are accomplished. In particular, the output $\overline{eg}_p)$ forces the termination of the subprocess **EG** of $\mathcal{F}_\mu(p :: \langle S, O, \rho, \sigma\rangle)$. As a consequence, the state of $\mathcal{F}_\mu(p :: \langle S, O, \rho, \sigma\rangle)$ after all these communications is $\mathcal{F}_\emptyset(p :: \langle S, O, \rho, \sigma\rangle)$. This implies the theorem. ∎

6 Conclusion

We have described the kernel of a reactive objects language, and gave two semantics for it, the first one in CHAM and the second one in π-calculus. In the CHAM semantics, instants are processed in a very simple and abstract way. On the contrary, a distributed termination algorithm is used in the π-calculus semantics. Finally, we prove the adequacy of the more concrete π-calculus semantics to the more abstract CHAM semantics. The CHAM reflects the simplicity of the language, although the π-calculus semantics express how to implement it (it would be an interesting task to directly implement the π-calculus semantics, for example in PICT[6]).

References

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

[2] G. Berry and G. Boudol. The Chemical Abstract Machine. *TCS*, 96:217–248, 1992.

[3] D. Walker. π-calculus semantics for object-oriented programming languages. In Springer-Verlag, editor, *Proc. TACS'91*, volume 526 of *LNCS*, pages 452–547, 1991.

[4] R. Milner. The Polyadic π-Calculus: A Tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[5] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.

[6] B.C. Pierce. Programming in the Pi-Calculus, an Experiment in Programming Language Design. Technical report, University of Edinburgh, 1993.

A The π-Calculus

We consider the π-calculus as defined in [4], that is the polyadic π-calculus together with an axiomatization over terms. The syntax is:

$$P \ ::= \ 0 \ | \ \overline{x}[\widetilde{y}].P \ | \ x(\widetilde{y}).P \ | \ P \,|\, Q \ | \ (y)P \ | \ [x=y]\,P\,P \ | \ P+P$$

There are two forms of binding: $x(\widetilde{y})$ and (y). The variable x is *free* in $x(\widetilde{y}).P$. We use $\mathsf{fn}(P)$ for the *free names* of P, $\mathsf{bn}(P)$ for the bound names of P and $\mathsf{n}(P)$ for all the names occurring in P.

Terms are quotiented by the structural congruence $\equiv$ defined by:

- $P \equiv Q$ if P is α-convertible to Q;
- let $\diamond \in \{|,+\}$ then $P \diamond 0 \equiv P$, $P \diamond Q \equiv Q \diamond P$ and $P \diamond (Q \diamond R) \equiv (P \diamond Q) \diamond R$;
- $!P \equiv P \,|\, !P$;
- $(x)\,0 \equiv 0$, $(x)(y)P \equiv (y)(x)P$, $(x)(\,P \mid Q) \equiv P \mid (x)\,Q$ and $(x)(\,P + Q) \equiv P + (x)\,Q$, if $x \notin \mathsf{fn}(P)$.

The evaluation relation is:

$$x(\widetilde{y}).P \xrightarrow{x(\widetilde{y})} P \qquad \overline{x}[\widetilde{y}].P \xrightarrow{\overline{x}[\widetilde{y}]} P \qquad \overline{x}[\widetilde{y}].P \mid x(\widetilde{z}).Q \xrightarrow{\tau} P \mid Q[\widetilde{y}/\widetilde{z}]$$

$$\frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \frac{x=y \quad P \xrightarrow{\alpha} P'}{[x=y]\,P\,Q \xrightarrow{\alpha} P'} \qquad \frac{x \neq y \quad Q \xrightarrow{\alpha} Q'}{[x=y]\,P\,Q \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{\overline{x}[\widetilde{z}]} P'}{(y)\,P \xrightarrow{\overline{x}\vartheta_y(\widetilde{z})} P'}\ x \neq y \qquad \frac{P \xrightarrow{\alpha} P'}{(y)\,P \xrightarrow{\alpha} (y)\,P'}\ y \notin \mathsf{n}(\alpha) \qquad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$$

$$\frac{Q \equiv P \quad P \xrightarrow{\alpha} P' \quad P' \equiv Q'}{Q \xrightarrow{\alpha} Q'}$$

where $\vartheta_y(z_1, \cdots, z_k) = [\vartheta_y(z_1), \cdots, \vartheta_y(z_k)]$ and

$$\vartheta_y(z) = \begin{cases} (y) & \text{if } z = y \\ z & \text{otherwise} \end{cases}$$

We don't use the "bang" operator "!", but instead, admit recursive definitions of processes. Let $\mathbf{A}$ be a process name, then:

$$\frac{A(\widetilde{x}) = P \quad P[\widetilde{d}/\widetilde{x}] \xrightarrow{\alpha} P'}{A(\widetilde{d}) \xrightarrow{\alpha} P'}$$

B Finite and Unbounded Stores

We are going to describe in π-calculus stores which are finite but unbounded. To this purpose we shall use *lists*, encoded by means of *ephemeral cells*, as suggested in [**?**]. Remark that, in this respect, our description departs from Walkers's one (see [**?**]), since stores are bounded there. Stores interact with the environment through three channels: wr is used for *writing* a value m into a cell x, vl for asking the content of a cell x and cp allows to perform the copy of the store.

$$\mathbf{STORE}(wr, vl, cp) \;=\; (\ell, nc)(\;\mathbf{RWC}_\ell(wr, vl, cp, nc) \;|\; (m, \ell')\,\mathbf{EC}(\ell, \mathtt{nil}, m, \ell') \;|\; \mathbf{CELL}(nc))$$

Remark that ℓ is the pointer to the head of the list representing the store). The process $\mathbf{RWC}_\ell$ accepts a request for reading or writing of a value or copy the content of the store. Accordingly it serves such requests.

$$\begin{array}{rcl}\mathbf{RWC}_\ell(wr, vl, cp, nc) & = & wr(x, \widetilde{m}).\, \mathbf{W}_\ell(\ell, x, \widetilde{m}, wr, vl, cp, nc)\\ & & +\;\; vl(x, ct).\, \mathbf{R}_\ell(\ell, x, wr, vl, cp, nc, ct)\\ & & +\;\; cp(ans, end).\, \mathbf{C}_\ell(\ell, ans, end, wr, vl, cp, nc)\end{array}$$

The function **CELL** is called in order to create a new cell of the memory.

$$\mathbf{CELL}(nc) \;=\; nc(\ell', x).\,(\;\;(\ell'', m)\mathbf{EC}(\ell, x, m, \ell') \;|\; \mathbf{CELL}(nc)\;)$$

where **EC** is the encoding of an ephemeral buffer:

$$\mathbf{EC}(\ell, x, m, \ell') \;=\; \overline{\ell}[x, m, \ell'].\, \ell(y, m', \ell'').\, \mathbf{EC}(\ell, y, m', \ell'')$$

Finally, the encodings of $\mathbf{W}_\ell$ and $\mathbf{R}_\ell$ are in order:

$$\begin{array}{rcl}\mathbf{W}_\ell(\ell, x, \widetilde{m}, wr, vl, nc) & = & \ell(y, m', \ell').\quad (\;\;[y = x]\; \overline{\ell}(x, \widetilde{m}, \ell').\, \mathbf{RWC}_\ell(wr, vl, cp, nc)\\ & & \qquad\qquad (\;[y = \mathtt{nil}]\;\;(\;\;\overline{\ell}(x, \widetilde{m}, \ell').\, \mathbf{RWC}_\ell(wr, vl, cp, nc)\\ & & \qquad\qquad\qquad\qquad |\; nc(\ell', \mathtt{nil})\;\;)\\ & & \qquad\qquad\qquad\quad \mathbf{W}_\ell(\ell', x, \widetilde{m}, wr, vl, nc)\;)\\ & & \quad)\end{array}$$

$$\begin{array}{rcl}\mathbf{R}_\ell(\ell', x, wr, vl, cp, nc, ct) & = & \ell'(y, \widetilde{m}, \ell'').\, \overline{\ell'}(y, \widetilde{m}, \ell'')\quad (\;\;[y = x]\; \overline{ct}[x, \widetilde{m}].\, \mathbf{RWC}_\ell(wr, vl, cp, nc)\\ & & \qquad\qquad (\;[y = \mathtt{nil}]\;\; \mathbf{RWC}_\ell(wr, vl, cp, nc)\\ & & \qquad\qquad\qquad \mathbf{R}_\ell(\ell'', x, wr, vl, nc, ct)\;)\\ & & \quad)\end{array}$$

$$\begin{array}{rcl}\mathbf{C}_\ell(\ell', ans, end, wr, vl, cp, nc) & = & \ell'(y, \widetilde{m}, \ell'').\, \overline{\ell'}(y, \widetilde{m}, \ell'').\quad (\;[y = \mathtt{nil}]\; \overline{end}[].\, \mathbf{RWC}_\ell(wr, vl, cp, nc)\\ & & \qquad \overline{ans}[y, \widetilde{m}].\, \mathbf{C}_\ell(\ell', ans, end, wr, vl, cp, nc)\;)\end{array}$$

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399