

# ML-Act, un langage fonctionnel d'Acteurs

---

Fabien Dagnat, Marc Pantel, Patrick Sallé

*LIMA/IRIT/INPT*

*2, rue Charles Camichel*

*31071 Toulouse CEDEX, France*

*Fabien.Dagnat@enseeiht.fr*

*Marc.Pantel@enseeiht.fr*

[www.enseeiht.fr/fr/recherche/info/Logiciel/vestale/vestale.html](http://www.enseeiht.fr/fr/recherche/info/Logiciel/vestale/vestale.html)

## Résumé

Un des thèmes de recherche développés par notre équipe est la conception d'outils d'analyses statiques de programmes écrits dans les langages d'Acteurs dans le but d'éliminer différentes formes d'erreurs sémantiques. Ces analyses ont été définies et validées pour un calcul d'Acteurs primitifs (CAP). Nous présentons dans cet article, un langage de haut niveau dérivé de ce calcul et du langage ML. Son but est de servir de plate-forme pour l'écriture de programmes répartis réalistes qui permettront de valider les différentes analyses statiques proposées. L'exposé utilise une version allégée du langage ML-ACT et en présente une sémantique à base de règles de réduction. Nous évoquons également sa traduction en CaML qui exploite les processus légers. Une première approche de son typage est ensuite présentée. Elle est basée sur le typage de la partie fonctionnelle du programme et sur la construction d'un terme CAP représentant la communication entre acteurs. Nous concluons en montrant les limitations de cette approche du typage et nous évoquons une seconde approche en cours d'étude.

## 1. Introduction

Portée par la vague INTERNET, la programmation concurrente répartie connaît un essor sans précédent. Le modèle d'Acteurs proposé par Hewitt [11] et développé par Agha [1], est l'un des premiers modèles de la programmation concurrente. Il a inspiré la plupart de ses successeurs mais reste l'un des plus simples à utiliser pour le programmeur. Il propose une structure à base d'agents (les acteurs) autonomes et réactifs qui encapsulent des données et des programmes.

La coopération entre les différents acteurs d'une application suit un protocole (fixe) de communication asynchrone de type point à point. Un système peut donc être décrit (à un instant donné) comme un ensemble d'acteurs s'exécutant en parallèle et un ensemble de messages en transit. La création dynamique d'acteur et la communication des *adresses* par message rendent la topologie de communication dynamique.

Divers aspects de la flexibilité du modèle de programmation par objets sont utilisés par le modèle d'Acteurs. D'une part, les acteurs encapsulent des données auxquelles on ne peut accéder que par une requête à l'acteur. D'autre part, chaque acteur présente une interface au monde extérieur et les messages qu'il recevra devront être compatibles avec celle-ci. Toutefois, ils diffèrent des objets concurrents classiques par le fait qu'ils peuvent dynamiquement changer leur interface. Ce phénomène appelé *changement de comportement* introduit une difficulté majeure au niveau du typage des acteurs qui ne peut plus être réalisé par les techniques de typage usuelles des objets.

Un des thèmes d'étude majeur de notre équipe est le typage des langages d'acteurs. Nous avons proposé dans ce but plusieurs techniques d'analyses statiques détaillées dans [5, 6, 7, 2] qui sont appliquées sur un calcul d'Acteurs primitifs CAP décrit dans [3, 4]. Ce formalisme permet de valider nos différentes analyses, mais il ne permet pas de décrire simplement des programmes répartis réalistes. En effet, CAP n'exprime pas simplement les structures séquentielles. Nous avons donc choisi de définir un langage d'Acteurs de haut niveau dont la partie concurrente correspond au calcul CAP. Le typage des programmes est alors composé de deux étapes : le typage classique de la partie séquentielle et le typage du squelette concurrent.

Parallèlement à ce développement de la programmation distribuée, la programmation fonctionnelle typée et ses qualités sont de plus en plus appréciées. On assiste, ces dernières années, à un effort d'adaptation de ce paradigme au développement d'applications distribuées. Parmi les nombreuses propositions, on peut distinguer deux stratégies : l'utilisation de bibliothèques spécifiques (CaML et MMM [23]) ou bien l'extension du langage (CML [22], PICT [21, 20], Facile [25], Join-Calculus Language [12]). Notre proposition consiste en l'intégration du modèle fonctionnel «à la ML» et du modèle d'Acteurs. Nous avons choisi de réaliser une extension «uniforme» de CaML et donc respecté les choix syntaxiques de Objective CaML pour une construction des acteurs similaire à celle des objets. Nous suivons donc plutôt la deuxième politique et appelons notre langage : ML-ACT.

Dans cet article, nous présentons succinctement les travaux effectués et ceux en cours dans le but de fournir un langage fonctionnel d'Acteurs fortement typé par inférence ainsi qu'un traducteur de ML-ACT en

CaML. Il est composé de trois parties qui traitent respectivement de la syntaxe du langage, de sa sémantique et de son typage. Dans la première partie, nous présentons le principe de la syntaxe de ML-ACT et introduisons plus précisément un sous-ensemble de celui-ci : mini-Act. Puis, nous décrivons sa sémantique et évoquons la structure actuelle de la génération de code CaML. La troisième partie introduit le système de type implanté par le compilateur, les deux phases (typage fonctionnel et typage concurrent) sont présentées l'une après l'autre. Enfin, nous concluons en évoquant les axes de développement du projet ML-ACT envisagés.

## 2. Le langage

Afin d'introduire la syntaxe du langage, nous présentons en exemple une implantation possible d'un gestionnaire d'impression. Il s'agira d'un acteur de comportement :

```
behavior empty_spooler () =
  message put job =
    become (busy_spooler job) (* change de comportement *)
  message ready iadd =
    send {noprint,()} to iadd (* envoi de message *)
and busy_spooler jobinit =
  val mutable jobs = [jobinit]
  message put job =
    jobs <- (put_last job jobs) (* place job en queue *)
  message ready iadd =
    send {print,hd jobs} to iadd;
    jobs <- (tl jobs);
    if jobs = [] then become (empty_spooler ())
end
```

La construction d'un message est représentée syntaxiquement par  $\{e_1, e_2\}$  où  $e_1$  fournit la valeur de l'étiquette du message et  $e_2$  le tuple des paramètres du message. Pour envoyer une requête d'impression, il suffira que le client connaisse l'adresse de ce serveur (`spooler`). La forme de la requête sera alors : `send {put, job} to spooler` où `job` représente le document à imprimer. Une imprimante effectue une requête `ready` lorsqu'elle peut imprimer un document.

Pour améliorer la clarté de la présentation de la sémantique, nous utiliserons mini-Act qui simplifie la syntaxe de ML-ACT tout en conservant toutes les difficultés inhérentes à celui-ci. Le lecteur intéressé

pourra consulter [8]. Nous exposerons les expressions du langage petit à petit, afin de les commenter.

Le langage (à l'image de ML) va être composé de deux entités : les expressions et les filtres. La première partie de la grammaire va être similaire au noyau fonctionnel de ML. Nous ne nous intéressons pas aux problèmes de parenthésage et de priorité : nous supposons qu'aucun conflit de reconnaissance n'est possible durant l'évaluation d'un filtre ou d'une expression. Durant l'exposé de la syntaxe, nous utilisons la notation  $\tilde{x}$  pour représenter une suite quelconque de  $x$ . Les termes (expressions et filtres) sont construits (en partie) par la grammaire suivante :

$$\begin{array}{l}
 e ::= x \mid c \mid C(\tilde{e}) \mid \text{fun}[\tilde{p} \rightarrow \tilde{e}] \mid \text{op}(\tilde{e}) \mid e \ e \mid e;e \mid \text{let } x = e \text{ in } e \\
 \quad \mid \text{letrec}[\tilde{x} = \tilde{e}] \text{ in } e \\
 p ::= - \mid x \mid c \mid C(\tilde{p})
 \end{array}$$

On dispose d'un ensemble de variable  $\mathbb{V}$ , de constantes (les entiers, les flottants, ..., () et []), d'un certain nombre de primitives (+, fst...) et d'un ensemble de constructeurs. Les primitives et les constructeurs prennent un tuple comme unique argument. Les expressions ont la même signification qu'en ML. Nous supposons en plus, qu'une variable n'apparaît qu'une unique fois dans un filtre.

Nous ajoutons à cette grammaire les règles concernant le noyau acteur, l'ensemble de nom  $\mathbb{M}$  est utilisé pour les étiquettes de messages.

$$\begin{array}{l}
 e ::= m(\tilde{e}) \mid \text{send } e \text{ to } e \mid \{\text{val}[\tilde{x} = \tilde{e}], \text{mess}[\tilde{m} = \tilde{e}]\} \mid \text{become } e \\
 \quad \mid \text{suicide} \mid \text{letactor}[\tilde{x} = \tilde{e}] \text{ in } e \mid x \leftarrow e \mid \text{ego}
 \end{array}$$

La construction de messages et l'envoi de messages sont effectués par les deux premières dérivations. Ensuite figure le terme correspondant à la formation d'un comportement composé de deux parties : les champs privés (la mémoire) et les messages (les réactions ou programmes). Le changement de comportement est réalisé par les instructions «become» et «suicide» (qui correspond à la disparition de l'acteur). La création simultanée d'acteurs se connaissant mutuellement se fait via un «letactor». L'expression qui crée un acteur lui fournit alors son comportement initial. Enfin, la mise à jour de champs privés est similaire à la syntaxe de Objective CaML. Notons que, dans ML-ACT, la construction «letactor» n'existe pas et la création de nouveaux acteurs se fait par l'instruction «new». Mais, pour obtenir une sémantique plus simple, la liaison lexicale (let) et la création d'acteurs ont été séparées.

```

let cell = fun[v → {val[mem = v],
                    mess[get = fun[c → send set(mem) to c];
                        set = fun[v → mem ← v]]}]
in letactor[a = cell 1; b = cell 2] in
    
```

```
send get(b) to a; send set(3) to b
```

Le programme mini-Act ci-dessus met en scène deux cellules (case mémoire) et correspond au programme ML-ACT suivant :

```
behavior cell v =  
  val mutable mem = v  
  message get c = send {set,mem} to c  
  message set v = mem <- v  
end;;  
let a = new cell 1  
and b = new cell 2  
in send {get,b} to a ;  
  send {set,3} to b;;
```

### 3. Sémantique opérationnelle

#### 3.1. Notations et conventions

La sémantique opérationnelle à base de réduction («sémantique à petit pas») est bien adaptée aux langages concurrents. Elle permet une description pertinente et élégante du sens des expressions dans un tel langage. La sémantique de mini-Act sera donc présentée en utilisant ce formalisme. Les règles de réduction auront la forme  $e \rightarrow r$  si  $h$  signifiant que si les hypothèses ( $h$ ) sont vérifiées et si l'expression à réduire s'unifie avec le sujet ( $e$ ), la règle peut être appliquée. Le résultat est alors ( $r$ ) dans lequel les variables sont remplacées par leur valeur (obtenue par unification).

Les adresses vont être gérées automatiquement, leur ensemble est noté  $\mathbb{A}$ . Lors de la création d'acteurs nous utiliserons la fonction *newname* qui fournira un objet de  $\mathbb{A}$  non utilisé. Le programmeur ne peut donc pas accéder directement aux adresses, il ne peut que les référencer. Cette gestion transparente de l'adressage permet de s'acquitter du problème d'unicité de l'adresse d'un acteur qui est garantie automatiquement. Dans une forme étendue du langage autorisant la réflexivité, une analyse statique serait mise en place pour imposer cette unicité [3].

Afin de présenter la sémantique du langage de façon concise, nous utilisons la classique notion de contexte (noté  $R$ ) et de «trou». La notation  $R[e]$  signifie que le trou est comblé par l'objet  $e$ . La sémantique de l'évaluation des expressions fonctionnelles de mini-Act (à l'image de CaML) consiste en une sémantique d'appel par valeur, le trou se déplacera donc de gauche à droite et de l'extérieur vers l'intérieur.

Deux ensembles de valeurs sémantiques sont construits :  $\mathcal{V}_s$  et  $\mathcal{V}_c$ . Le premier est l'ensemble de toutes les valeurs qui peuvent être le résultat d'une évaluation. Le second correspond au sous-ensemble de valeurs communicables dans un message. En effet, l'envoi d'une fonction qui pourrait mettre à jour directement les champs privés est interdit car cela viole la propriété d'encapsulation. Plus précisément,  $e_c$  est utilisé pour dénoter les expressions communicables, qui correspondent aux expressions ne contenant ni `ego`, ni `suicide`, ni `become` ni  $x \leftarrow e$ .

$$\begin{array}{l} v_s ::= v_c \mid [\widetilde{p} \rightarrow e] \mid C(\widetilde{v}_s) \\ v_c ::= c \mid [\widetilde{p} \rightarrow e_c] \mid \langle\langle \mathcal{M}, \mathcal{R} \rangle\rangle \mid a \mid C(\widetilde{v}_c) \mid m(\widetilde{v}_c) \end{array}$$

Les valeurs sémantiques possibles sont donc les classiques constantes, abstractions fonctionnelles et termes construits (communicables ou non) auxquels sont ajoutés la notion de fermeture comportementale, d'adresse (un élément de  $\mathbb{A}$ ) et de message. La fermeture comportementale est composée de deux éléments : une mémoire initiale ( $\mathcal{M} : \mathbb{V} \rightarrow \mathcal{V}_s$ ) et un ensemble de réactions ( $\mathcal{R} : \mathbb{M} \rightarrow \mathcal{Exp}$ ). Ils associent respectivement une valeur à une variable et une expression (la réaction) à une étiquette de message.

La sémantique de mini-Act va s'exprimer par une réduction sur des *configurations*. Une configuration est la description de l'état d'une exécution, elle consiste donc en un certain nombre d'acteurs s'exécutant en parallèle et des messages transitant par le médium de communication. La définition de cette notion est inspirée du  $\pi$ -calcul et du calcul CAP. Les termes décrivant les configurations sont obtenus par la grammaire :

$$w ::= \epsilon \mid w \parallel w \mid a \triangleleft m(\widetilde{v}) \mid a \triangleright \langle\langle \mathcal{M}, \mathcal{R} \rangle\rangle \mid \llbracket e, a, \mathcal{M}, \mathcal{R} \rrbracket$$

Les configurations peuvent être vides ; elles se composent parallèlement et contiennent des messages (d'étiquette  $m$  et d'arguments  $\widetilde{x}$  destinés à  $a$ ) et des acteurs (de comportement  $\langle\langle \mathcal{M}, \mathcal{R} \rangle\rangle$ ). La dernière configuration présentée correspond à une évaluation fonctionnelle dans un acteur. Les quatre champs qu'elle contient représentent respectivement l'expression de la réaction en cours de calcul, l'adresse de l'acteur (son *ego*), sa mémoire et ses réactions (ils composent le *self*). Notons enfin que le programme est initialement traité comme un acteur anonyme sans mémoire ni réaction.

### 3.2. Évaluation

La présentation de l'évaluation va se faire en trois étapes : tout d'abord la réduction des configurations non-fonctionnelle, puis les réductions fonctionnelles correctes et enfin le traitement des erreurs. Afin de ne pas alourdir la présentation, le filtrage et le traitement des erreurs ne sera pas présenté en détail, le lecteur intéressé consultera [8].

L'opérateur de composition parallèle est associatif, commutatif et  $\epsilon$  est son élément neutre. Toutes les configurations congrues (égales modulo l'application d'une des trois propriétés de l'opérateur de composition parallèle) sont assimilées. L'opérateur de filtrage (noté  $/$ ) des valeurs sémantiques construit une substitution qui remplacera chaque variable du filtre par la valeur qui lui correspond dans la valeur filtrée ou renvoie **fail** si le filtrage échoue. Nous ne présenterons pas les règles classiques de substitution qui consistent pour une application  $\psi(e)$  (avec  $\psi = [v/x]$ ) à remplacer toutes les occurrences libres de  $x$  dans  $e$  par  $v$ .

### réduction non-fonctionnelle

1 : $w \parallel w_1$	$\rightarrow w \parallel w_2$ si $w_1 \rightarrow w_2$
2 : $a \triangleright \langle \mathcal{M}, \mathcal{R} \rangle \parallel a \triangleleft m(\tilde{v})$	$\rightarrow [([a/ego](\mathcal{R}(m)) v_1) \dots v_n], a, \mathcal{M}, \mathcal{R}]$ si $m \in \text{dom}(\mathcal{R})$
3 : $[(\cdot), a, \mathcal{M}, \mathcal{R}]$	$\rightarrow a \triangleright \langle \mathcal{M}, \mathcal{R} \rangle$

La première règle (1) exprime l'indépendance de la réduction d'une configuration par rapport aux configurations qui sont en parallèle avec elle; elle permet de simplifier les règles en se concentrant sur la partie vraiment intéressante de la configuration que l'on cherche à réduire. Les deux dernières transitions signalent respectivement le début et la fin d'une évaluation fonctionnelle. En effet, lorsqu'un acteur sait traiter un message (2), il disparaît de la configuration ainsi que le message accepté. En contrepartie, un terme «fonctionnel» (un acteur actif) est introduit dans le milieu, il consiste en l'évaluation de la réaction de l'acteur dans laquelle l'adresse a été substituée à l'*ego* et qui utilise les données transportées par le message. La terminaison d'un acteur doit être une valeur vide (unit) et, si c'est le cas (3), l'acteur (passif) est régénéré à partir du nouvel état de la mémoire et de ses réactions.

Cette réapparition de l'acteur exprime le fait que par défaut il boucle sur son comportement. Ce choix est inhabituel dans la communauté des acteurs, mais nous a semblé cohérent car les entités ne changent que rarement de comportement. Le défaut de cette sémantique est que l'acteur n'est capable de traiter un message que lorsqu'il a fini son calcul. Or, dans certain cas, une partie de ce calcul n'a pas d'influence sur son état futur, le parallélisme potentiel est donc diminué. Cependant, nous envisageons d'inclure au compilateur une étape d'analyse qui permettrait d'insérer les changements de comportement aux meilleurs moments.

### calcul fonctionnel

La deuxième série de règles de réduction consiste en un calcul fonctionnel se déroulant sans erreurs. On subdivise celles-ci en deux

familles, la première contient les règles qui ne modifient pas la configuration (elles ne modifient que le contexte), alors que les secondes la modifie.

Pour présenter la première famille, l'écriture d'une réduction du type  $\llbracket \mathbf{R}[e_1], a, \mathcal{M}, \mathcal{R} \rrbracket \rightarrow \llbracket \mathbf{R}[e_2], a, \mathcal{M}, \mathcal{R} \rrbracket$  sera dénotée  $e_1 \rightsquigarrow e_2$ . Des références à  $\mathcal{M}$  sont utilisées si nécessaire.

1: $x$	$\rightsquigarrow \mathcal{M}(x)$ <b>si</b> $x \in \text{dom}(\mathcal{M})$
2: $v; e$	$\rightsquigarrow e$
3: $\text{fun}[\widetilde{p \rightarrow e}]$	$\rightsquigarrow [\phi(\mathcal{M})(\widetilde{p \rightarrow e})]$
4: $\{\text{val}[\widetilde{x = v}], \text{mess}[\widetilde{m = e}]\}$	$\rightsquigarrow \langle \langle \mathcal{M}_1, \mathcal{R}_1 \rangle \rangle$ <b>si</b> $\mathcal{M}_1 = \{x \mapsto v\} \dots$ et $\mathcal{R}_1 = \{m \mapsto \phi(\mathcal{M})(e)\} \dots$
5: $\text{op}(\widetilde{v})$	$\rightsquigarrow v$ <b>si</b> $\text{compute}(\text{op}, \widetilde{v}) = v$
6: $[p \rightarrow e; \widetilde{p \rightarrow e}] v$	$\rightsquigarrow [\widetilde{p \rightarrow e}] v$ <b>si</b> $v/p \Rightarrow \text{fail}$
7: $[p \rightarrow e; \widetilde{p \rightarrow e}] v$	$\rightsquigarrow \psi(e)$ <b>si</b> $v/p \Rightarrow \psi$
8: $\text{let } x = v \text{ in } e$	$\rightsquigarrow \psi = [v/x](e)$
9: $\text{letrec}[\widetilde{x = v}] \text{ in } e$	$\rightsquigarrow [\psi(e_1)/x_1] \dots [\psi(e_n)/x_n](e)$
10: $\text{letactor}[\widetilde{x = e}] \text{ in } e$	$\rightsquigarrow \text{new}(a_1, \psi(e_1)); \dots; \psi(e)$ <b>si</b> $\forall i a_i = \text{newname}$ et $\psi = [a_1/x_1] \dots [a_n/x_n]$

Les quatre premières règles (1, 2, 3 et 4) sont classiques. Les règles (3 et 4) utilisent la fonction  $\phi$  définie par :  $\phi(\{\widetilde{x \mapsto v}\}) = [v_1/x_1] \dots [v_n/x_n]$ . L'appel de primitive (5) consiste en un appel de code (en librairie) au travers de la fonction *compute* ; celle-ci signale toute erreur d'exécution (elle correspond à une  $\delta$ -règle du  $\lambda$ -calcul). L'utilisation de primitives ne peut aboutir à une application partielle et nous supposons que son calcul ne peut pas boucler. L'application (6 et 7) utilise l'opérateur de filtrage, on tente de filtrer la valeur (appliquée) par les différents filtres dans l'ordre de déclaration. Les liaisons lexicales (8 et 9) consistent en la construction d'une substitution (éventuellement de façon récursive) et l'application de celle-ci au corps. Pour simplifier, nous supposons que chaque bloc de liaison simultané ne contient un nom qu'une seule fois. La création d'acteurs (10) est scindée en deux étapes : l'allocation et l'initialisation. L'allocation consiste à attribuer à chaque acteur une adresse puis à remplacer les variables du programmeur par ces noms. L'initialisation sera traitée au paragraphe suivant puisqu'elle modifie la configuration.

Les règles modifiant la configuration sont présentées in-extenso.

1 : $\llbracket \mathbb{R}[x \leftarrow v], a, \mathcal{M}, \mathcal{R} \rrbracket$ si $x \in \text{dom}(\mathcal{M})$	$\rightarrow \llbracket \mathbb{R}[\()], a, \mathcal{M} :: \{x \mapsto v\}, \mathcal{R} \rrbracket$
2 : $\llbracket \mathbb{R}[[p \rightarrow e] v], \_ , \_ , \_ \rrbracket$ si $v/p \Rightarrow \mathbf{fail}$	$\rightarrow \epsilon$
3 : $\llbracket \mathbb{R}[\text{send } m(\tilde{v}) \text{ to } a'], a, \mathcal{M}, \mathcal{R} \rrbracket$ si $\tilde{v} \in \mathcal{V}_c^*$	$\rightarrow \llbracket \mathbb{R}[\()], a, \mathcal{M}, \mathcal{R} \rrbracket \parallel a' \triangleleft m(\tilde{v})$
4 : $\llbracket \mathbb{R}[\text{become } \langle \mathcal{M}_1, \mathcal{R}_1 \rangle \rangle], a, \mathcal{M}, \mathcal{R} \rrbracket$ si $a' = \mathit{newname}$	$\rightarrow a \triangleright \langle \mathcal{M}_1, \mathcal{R}_1 \rangle \parallel \llbracket \mathbb{R}[\()], a', \mathcal{M}, \{\} \rrbracket$
5 : $\llbracket \mathbb{R}[\text{suicide}], \_ , \_ , \_ \rrbracket$	$\rightarrow \epsilon$
6 : $\llbracket \mathbb{R}[\text{new}(a', \langle \mathcal{M}_1, \mathcal{R}_1 \rangle \rangle)], a, \mathcal{M}, \mathcal{R} \rrbracket$	$\rightarrow a' \triangleright \langle \mathcal{M}_1, \mathcal{R}_1 \rangle \parallel \llbracket \mathbb{R}[\()], a, \mathcal{M}, \mathcal{R} \rrbracket$

La mise à jour (1) de la valeur d'un champ privé  $x$  se fait par l'opération « $::$ » qui remplace l'ancienne valeur associée à  $x$  par la nouvelle. Il convient de remarquer que la mémoire peut être gérée comme un tableau puisque, lors de l'exécution, aucune nouvelle variable n'est ajoutée. Si lors du filtrage (2) aucun filtre ne convient, l'évaluation disparaît. L'envoi de message (3) consiste, tout simplement, en l'introduction dans la configuration de celui-ci. Lorsqu'un acteur change de comportement (4), on crée un acteur anonyme (personne ne le connaît) qui possède la mémoire de l'acteur avant le changement de comportement et ne peut réagir à aucun message. Notre sémantique est conçue dans le contexte de langage de programmation qui dispose d'un ramasse miette. Celui-ci se chargera de détruire les acteurs que personne ne connaît ou qui ne peuvent réagir à aucun message. Le suicide d'un acteur (5) est la fin de son exécution, et donc sa disparition définitive du milieu. Enfin, la phase d'initialisation d'un acteur (6) l'introduit avec son comportement dans la configuration.

### traitement des erreurs

Enfin, nous présentons les diverses erreurs pouvant survenir à l'exécution d'un programme. Le choix a été fait de ne pas interrompre l'exécution si une erreur survenait dans un acteur; la réaction consiste donc uniquement à supprimer de la configuration l'évaluation fonctionnelle qui ne s'est pas bien déroulée. On peut éventuellement introduire un mécanisme de signalisation de ces erreurs. Toutes ces erreurs vont cependant être évitées par une phase de typage qui sera présentée dans la section suivante. Enfin, signalons que certains objets peuvent atteindre des états où ils ne pourront plus être réduits et ne disparaîtrons pas du milieu. C'est le cas par exemple, d'une évaluation fonctionnelle dans un acteur se terminant par une autre valeur que «unit». Le mécanisme de typage éliminera ces cas, ce qui justifie l'utilisation d'une approche opérationnelle.

Nous ne présentons pas les règles de sémantique traitant des erreurs, nous allons les exposer rapidement :

- L'apparition d'une variable ne figurant pas dans la mémoire.
- Un constructeur appliqué partiellement.
- La fonction «compute» (déjà vue) renvoie une erreur.
- On applique un objet qui n'est pas une fermeture fonctionnelle.
- On envoie un objet qui n'est pas un message ou qui est un message contenant des données non-communicables.
- On envoie un message à un objet qui n'est pas une adresse.
- L'installation d'un comportement (*new*) ou le changement de comportement (*become*) dispose d'un objet qui n'est pas une fermeture comportementale.

### 3.3. Le compilateur

Cette sémantique formelle a été développée conjointement à l'implantation d'un compilateur pour le langage. Actuellement, le compilateur de ML-ACT réalisé traduit un programme en un ensemble de module CaML. La simulation du parallélisme et du non-déterminisme est faite en utilisant les processus légers d'Objective CaML. Nous allons examiner les différentes entités du modèle d'acteur et, pour chacune, expliciter la réalisation que nous avons choisie.

Un *comportement* est traduit en une fonction récursive. Les arguments de celle-ci sont la valeur de *ego* et les arguments du comportement. Les champs privés sont réalisés par des références locales et le corps d'une telle fonction consiste en une boucle sans fin qui récupère un message dans la boîte aux lettres de l'acteur et filtre le message sur son étiquette. Un changement de comportement est traduit en un appel de la fonction correspondant au nouveau comportement. Ainsi, le graphe d'appel de ces fonctions mutuellement récursives simulera l'automate de tous les comportements possibles d'un acteur.

Les *adresses* des acteurs sont codées par des entiers ; ceux-ci représentent l'indice de leur boîte aux lettres dans le tableau qui les contient.

Un *acteur* est traduit en un processus léger dont le code est la fonction associée à son comportement courant. Les acteurs du milieu possèdent une mémoire commune qui contient le tableau de leurs boîtes aux lettres. Celles-ci sont codées par des quadruplets : un sémaphore d'exclusion

mutuelle (mutex), une variable condition qui active l'acteur lorsqu'un message arrive dans la boîte aux lettres et deux files. La première des deux files contient les messages qui arrivent à l'acteur et qui sont donc en attente de traitement. La seconde contient les messages qui n'ont pu être acceptés par les comportements précédents. Lors d'un changement de comportement, on cherche à exécuter le maximum de réactions possibles parmi les messages de cette seconde file avant de passer à la file des messages en attente. La création d'un acteur consiste donc à créer une boîte aux lettres (quitte à agrandir le tableau de celles-ci) ainsi qu'un processus léger qui exécute le code de l'acteur.

Enfin, le dernier objet à traduire est le *message*; celui-ci est implanté de façon immédiate par un couple composé d'une étiquette de message et d'un tuple représentant les données qu'il transporte.

Cette implantation simple nous sert de test pour vérifier certaines analyses que l'on souhaite développer sur les acteurs et permet également l'«exécution» de programmes d'acteurs.

## 4. Une première approche pour le typage

Dans cette section, nous présentons un système de type pour mini-Act. En effet, pour supprimer les programmes produisant des erreurs sémantiques lors de leur évaluation, nous simulons l'exécution de ce programme par une approximation de chaque expression. Ce travail d'analyse statique sur les programmes est scindé en deux étapes distinctes. Dans un premier temps, le but est de garantir un fonctionnement correct de la partie fonctionnelle. Pour cela, nous effectuons un typage «à la ML», dans lequel nous approximons grossièrement tous les effets de la communication. Puis, dans un deuxième temps, nous extrayons du code un terme du calcul CAP. Une analyse très précise des problèmes liés à la communication est alors effectuée sur ce terme.

### 4.1. Typage fonctionnel

Notre système de type fonctionnel est basé sur le système de type classique de Milner [14, 9, 13] dont il diffère par la stratégie choisie pour la reconstruction des types. En effet, suivant les travaux de [19] sur le typage d'un langage fonctionnel par collecte et résolution de contraintes, notre système extrait des contraintes du programme, puis les résout. Le «typeur» actuel collecte des contraintes d'égalité entre types (comme dans ML) et les résout par unification. Toutefois, nous envisageons de générer des contraintes d'inclusion et d'utiliser un algorithme de

résolution de contraintes ensemblistes afin de les résoudre.

On dispose de constantes de types (notées  $c$ ) classiques ( $Int, Float, \dots, Unit$ ) auxquelles nous ajoutons pour approximer les phénomènes concurrents :  $Beh$  (comportement),  $Addr$  (adresse),  $Mess$  (message). De plus, un ensemble de variables supposé infini est utilisé, la fonction  $new$  nous fournira une variable encore inutilisée à chaque appel. Ces variables de type sont notées  $\tau$ . Les constructeurs de la sémantique possèdent chacun un constructeur de type associé ( $C_t$ ). Leur type sera obtenu par la fonction  $Typeof$  qui typera également les constantes. Enfin, nous utiliserons le type  $mut(t)$  pour indiquer le type d'un champ privé de type  $t$ . La syntaxe des types suit la grammaire suivante :

$$t ::= \tau \mid C_t(\tilde{t}) \mid t \rightarrow t \mid t \times t \mid mut(t) \mid c$$

Pour rendre compte du polymorphisme, l'environnement qui conservera le type des variables, utilise les *schémas de type* usuels, ceux-ci sont notés :  $\forall \tilde{\alpha}. \tau$ . Cette écriture signifiant que dans  $\tau$ , les variables  $\alpha_1 \dots \alpha_n$  sont universellement quantifiées. Nous utilisons les fonctions  $Gen$  et  $Spe$  qui respectivement fournissent le schéma de type correspondant à un type (le tuple est composé des variables libres du type) et un type associé à un schéma (les variables quantifiées sont remplacées par des variables «fraîches»). La notion de généralisation est limitée aux liaisons lexicales («let» et «letrec»).

Nous ne présenterons ici que la collecte des contraintes de type. La présentation du système de type va suivre le principe de la sémantique naturelle. Le typage consistera donc en la construction d'un arbre de dérivation. Pour alléger les règles d'inférence, celle-ci ajouterons des contraintes dans un ensemble global de contraintes. Elle seront de la forme présentée ci-dessous où  $C$  est l'ensemble des contraintes issues de l'application de la règle.

$$\frac{E \vdash e_1 : \tau_1 \dots E \vdash e_n : \tau_n}{E \vdash f(e_1, \dots, e_n) : \tau} \{C\}$$

Le typage des filtres est classique et ne sera pas présenté ; il calcule pour chaque filtre son type et construit un environnement contenant ses variables (règle de la forme :  $\vdash p_i : \tau_i, E_i$ ). Il prévient la formation de filtres erronés qui ne sont pas des cas d'erreur à proprement parler, mais qui échoueront systématiquement et ne sont donc pas souhaitables. Il vérifie également le respect de la linéarité (que nous avons supposée dans la section précédente). Cependant, il ne comporte pas d'analyse précise permettant de déterminer si un filtrage est complet ou si certains cas de filtrage ne seront jamais utilisés.

La figure 1 fournit les règles qui reconstruisent le type d'une expression. Il convient de souligner quelques particularités de ce système

$$\begin{array}{c}
 \frac{}{E \vdash c : \text{typeof}(c)} \quad \frac{x \in \text{dom}(E) \quad \text{Spe}(E(x)) = \text{mut}(t)}{E \vdash x : t} \\
 \frac{x \in \text{dom}(E) \quad \forall t \text{ Spe}(E(x)) \neq \text{mut}(t)}{E \vdash x : \text{Spe}(E(x))} \quad \frac{E \vdash e_i : \tau_i}{E \vdash m(\tilde{e}) : \text{Mess}} \\
 \frac{E \vdash e_i : \tau_i \quad \tau = \text{new}}{E \vdash C(\tilde{e}) : \tau} \{ \tau_1 \times \dots \times \tau_n \rightarrow \tau = \text{typeof}(C) \} \\
 \frac{E \vdash e_i : \tau_i \quad \tau = \text{new}}{E \vdash \text{op}(\tilde{e}) : \tau} \{ \tau_1 \times \dots \times \tau_n \rightarrow \tau = \text{typeof}(\text{op}) \} \\
 \frac{\tau_{in} = \text{new} \quad \tau_{out} = \text{new}}{\vdash p_i : \tau_i, E_i \quad E::E_i \vdash e_i : \tau'_i} \bigcup_i (\{ \tau_i = \tau_{in}; \tau'_i = \tau_{out} \}) \\
 \frac{E \vdash \widetilde{[p \rightarrow e]} : \tau_{in} \rightarrow \tau_{out}}{E \vdash e_1 e_2 : \tau_3} \{ \tau_1 = \tau_2 \rightarrow \tau_3 \} \\
 \frac{E \vdash e_1 : \tau_1 \quad E \vdash e_2 : \tau_2 \quad \tau_3 = \text{new}}{E \vdash e_1 e_2 : \tau_3} \{ \tau_1 = \tau_2 \rightarrow \tau_3 \} \\
 \frac{E \vdash e_1 : \tau_1 \quad E \vdash e_2 : \tau_2}{E \vdash \text{send } e_1 \text{ to } e_2 : \text{Unit}} \{ \tau_1 = \text{Mess}; \tau_2 = \text{Addr} \} \quad \frac{\text{ego} \in \text{dom}(E)}{E \vdash \text{suicide} : \text{Unit}} \\
 \frac{E \vdash e_1 : \tau_1 \quad E \vdash e_2 : \tau_2}{E \vdash e_1; e_2 : \tau_2} \quad \frac{\text{ego} \in \text{dom}(E) \quad E \vdash e : \tau}{E \vdash \text{become } e : \text{Unit}} \{ \tau = \text{Beh} \} \\
 \frac{\text{ego} \in \text{dom}(E) \quad x \in \text{dom}(E) \quad \text{Spe}(E(x)) = \text{mut}(t) \quad E \vdash e : \tau}{E \vdash x \leftarrow e : \text{Unit}} \{ \tau = t \} \\
 \frac{\text{lin}(\tilde{x}) \quad E' = E::\bigcup_i \{x_i : \forall. \text{Addr}\} \quad E' \vdash e_i : \tau_i \quad E' \vdash e : \tau}{E \vdash \text{letactor}[\widetilde{x = e}] \text{ in } e : \tau} \bigcup_i (\{ \tau_i = \text{Beh} \}) \\
 \frac{\text{lin}(\tilde{x}) \quad \text{lin}(\tilde{m}) \quad E \vdash e_i : \tau_i \quad \tau''_i = \text{new}}{E::\bigcup_i (\{x_i : \text{mut}(\tau_i)\})::\{\text{ego} : \text{Addr}\} \vdash e'_i : \tau'_i} \{ \tau'_i = \tau''_i \rightarrow \text{Unit} \} \\
 \frac{E \vdash \{\text{val}[\widetilde{x = e}], \text{mess}[\widetilde{m = e'}]\} : \text{Beh}}{E \vdash e_1 : \tau_1, C_1 \quad E::\{x : \text{Gen}(\tau_1, C_1)\} \vdash e_2 : \tau_2, C_2} \\
 \frac{E \vdash e_1 : \tau_1, C_1 \quad E::\{x : \text{Gen}(\tau_1, C_1)\} \vdash e_2 : \tau_2, C_2}{E \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2, C_2} \\
 \frac{\text{lin}(\tilde{x}) \quad \tau_i = \text{new} \quad E::\bigcup_i \{x_i : \tau_i\} \vdash e_i : \tau'_i, C_i}{C_g = \bigcup_i (C_i \cup \{\tau'_i = \tau_i\}) \quad E::\bigcup_i \{x_i : \text{Gen}(\tau_i, C_g)\} \vdash e : \tau, C} \\
 \frac{E \vdash e_1 : \tau_1, C_1 \quad E::\bigcup_i \{x_i : \tau_i\} \vdash e_i : \tau'_i, C_i}{E \vdash \text{letrec}[\widetilde{x = e}] \text{ in } e : \tau, C}
 \end{array}$$

FIG. 1 – Le typage des expressions.

de type. Tout d'abord la vérification de l'utilisation correcte des primitives acteurs (suicide, become, ...) se fait par le test de la présence dans l'environnement de *ego*. Ensuite, les règles de typage des liaisons lexicales ne suivent pas la convention sur les contraintes. En effet, la généralisation d'un type manipule les contraintes en les résolvant, nous avons donc choisi de présenter précisément ces manipulations. Enfin, les règles de liaisons multiples utilisent un prédicat *lin* afin de vérifier leur linéarité.

## 4.2. Analyse de la communication

Du changement de comportement découle une évolution constante des interfaces des acteurs qui rend difficile l'étude des *messages orphelins*, c'est-à-dire des messages qui, reçus par un acteur, ne pourront jamais être traités. Divers travaux tentent de résoudre ce problème [24, 18, 15]. Dans une première étape, nous avons choisi de réutiliser directement les outils développés par J-L. Colaço dans le cadre de sa thèse [2]. Nous avons donc choisi d'extraire de nos programmes un terme du calcul CAP. Cette approche est inspirée des travaux de F. Nielson et H.R. Nielson ([16]) qui effectuent une extraction d'un terme du  $\pi$ -calcul à partir d'un programme CML, en utilisant un calcul d'effet durant le typage. Toutefois, notre approche sépare les deux phases, l'extraction se fait donc sur un programme décoré par les types. L'utilisation de l'unification pose un problème pour le calcul des effets car l'algèbre des termes du  $\pi$ -calcul (et donc a fortiori celle de CAP) n'est pas libre. F. Nielson et H.R. Nielson résolvent le problème en simplifiant les effets calculés (voir [17]). Nous souhaitons conserver la précision du terme CAP et nous avons donc choisi de restreindre la forme des programmes typables. Ainsi, nous rejetons des programmes corrects pour lesquels notre algorithme ne peut calculer un type ou un effet. Cette stratégie nous permet de disposer d'un type très précis pour les programmes acceptés.

La traduction intégrale de ML-ACT en CAP est possible mais la structure du type du terme obtenu ne refléterait pas du tout le type du programme ML-ACT, nous avons donc suivi une approche intermédiaire. Les limites imposées sont donc :

- Un terme construit ne contient pas de données communicables,
- Les fonctions récursives n'ont pas d'effet (elles ne peuvent contenir ni envoi de message ni création d'acteur).

Nous allons présenter uniquement les idées directrices de cette extraction qui est très technique. Pour la présentation de CAP nous

renvoyons à [3], cet article contient également certains aspects du problème de la traduction d'un langage de haut niveau en CAP. Enfin, pour une présentation complète des techniques utilisées et de l'algorithme de traduction, on peut consulter [8].

L'extraction du terme CAP correspondant à un programme se déroule en deux phases. La première étape consiste à supprimer toutes les entités fonctionnelles, alors que la seconde effectuera le calcul des *self* et dépliera les comportements (pour supprimer la récursivité).

### Phase 1

Nous allons examiner un à un les objets que peut contenir un programme et indiquer rapidement le travail effectué pour le traduire.

Tout d'abord, le calcul CAP ne va contenir aucun terme fonctionnel, donc les variables qui ne sont ni de type *Beh* ni de type *Addr* disparaissent. Les entités suivantes sont également supprimées : les constantes, les appels de primitives et les constructions de termes.

Pour traduire chaque filtrage, on ne conserve que les filtres qui sont des variables -abstraction simple du  $\lambda$ -calcul- (de type *Beh* ou *Addr*) car ce sont les seuls objets autorisés à transporter des données communicables. De plus, la traduction des différent cas utilise un opérateur (+) de choix indéterministe. Celui-ci n'existe pas en CAP mais son ajout et l'extension des analyses associées ne posent pas de problème.

La traduction des liaisons lexicales ainsi que celle de l'application consiste à substituer des noms par les valeurs. Donc, pour calculer le résultat de l'appel d'une fonction, l'appel est remplacé par le corps de la fonction dans lequel chaque paramètre est substitué par sa valeur. Cette stratégie est applicable uniquement parce que les fonctions récursives n'ont pas d'effet.

Le traitement de la séquence est plus complexe car elle introduit un ordre d'évaluation qui n'existe pas en CAP. Le problème est résoluble parce que cet ordre ne porte que sur les accès ou les mises à jour de champs privés (il n'y a plus de calcul fonctionnel). Donc, pour chaque réaction de chaque comportement, nous construisons une séquence de ces accès en lecture ou en écriture. Après cette collecte, les accès en écriture disparaissent ainsi du code et les expressions en séquence se traduisent donc directement en une composition parallèle.

La formation de comportement, la construction de messages, l'envoi de messages, le changement de comportement ainsi que la création d'acteurs ne sont pas modifiés durant cette phase. Enfin, le suicide consiste en l'«oubli» de toutes les opérations qui le suivent (au sens de l'ordre induit par la séquence).

**Phase 2**

A l'issu du premier traitement, le terme obtenu est un terme CAP dans lequel les comportements sont (encore) récursifs et dans lequel les captures de *ego* et *self* n'ont pas encore été insérées. La deuxième étape de la traduction réalise donc deux tâches qui sont étroitement liées. Elle effectue le dépliage des comportements récursifs et le calcul des valeurs des *ego* et *self*.

C'est la capture de chaque comportement par une variable (le *self*) permet de garantir la terminaison du dépliage. En effet, lors du parcours en profondeur de l'arbre des comportements que peut prendre l'acteur, on atteindra forcément un comportement déjà capturé (car ils sont en nombre fini). Notons que pour que cet argument soit valable, les comportements ne doivent pas avoir d'argument. Les éventuels paramètres sont donc transformés en champs privés pour vérifier cette contrainte.

Pour calculer les valeurs des *self*, on va parcourir (dans l'ordre) les séquences de lecture/écriture construites à l'étape précédente. Initialement, une variable dont le nom doit être unique (pour éviter tout problème de portée) est utilisée. Dans l'implantation actuelle, elle consiste en un "s" concaténé avec la valeur d'un compteur. A chaque écriture  $x \leftarrow v$ , le contenu de la variable *self* est remplacé par (*self*. $x \leftarrow v$ ) (où *self* correspond à la valeur courante de l'état de l'acteur), et une lecture  $x$  est remplacée par *self*. $x$ .

**4.3. Un exemple**

Nous concluons la présentation du système de type par l'exemple d'une case mémoire. Après avoir déposé une valeur dans cette case, toute lecture provoquera la perte de son contenu.

```
behavior empty_buf () =
  message put v =
    become (full_buf v)
and full_buf v =
  message get c =
    send {reply,v} to c ;
    become (empty_buf ())
end;;
let a = new (empty_buf ());;
```

Le typage «à la ML» fournit :

```
empty_buf :unit -> beh
full_buf  :'a   -> beh
a : addr
```

Le squelette concurrent extrait de l'exemple est le terme CAP suivant :

$$\nu a(a \triangleright [put\ v = \zeta(e_1, s_1)(e_1 \triangleright [get\ c = \zeta(e_2, s_2)(c \triangleleft reply(v) \parallel e_2 \triangleright s_1)])])$$

Les outils développés par J-L. Colaço permettent alors sur cette expression de vérifier l'arité des messages envoyé à un acteur ; de

vérifier la linéarité -c'est-à-dire, d'interdire l'installation de plusieurs comportements sur un même adresse-; de détecter les messages orphelins -les messages qui ne pourront jamais être traités par un acteur.

## 5. Conclusion et Perspectives

La sémantique formelle ainsi que le compilateur de ML-ACT nous offrent une plate-forme de validation. Actuellement nous implantons diverses applications réparties sous forme d'acteurs afin de valider les divers analyseurs conçus par l'équipe.

D'autre part, le système de type actuel fournit de bons résultats sur les programmes fonctionnels ainsi que sur les programmes uniquement concurrents. Le transport de données du premier ordre dans les messages est également traité. Toutefois, des évolutions sont nécessaires afin de parvenir à un système de type intégrant le transport de données d'ordre supérieur et la gestion de fonctions récursives contenant des effets. En effet, l'approximation fonctionnelle d'une part, et l'approximation concurrente d'autre part, sont toutes deux trop grossières et ne se communiquent pas assez d'informations pour autoriser le typage des termes lorsque le fonctionnel et le concurrent sont fortement entrelacés.

Pour surmonter ces limitations, nous envisageons deux techniques possibles. La première consiste en l'extension du calcul CAP par des valeurs correspondant aux types obtenus lors de l'approximation fonctionnelle. Ainsi, en augmentant la communication entre les deux systèmes de type, on pourrait parvenir à une résolution partielle du problème. Cependant, dans le cas des fonctions récursives ayant un effet, les travaux de F. Nielson et H.R. Nielson dans [16] montrent que nous serons confrontés aux problèmes d'unification des termes CAP. Cette remarque nous amène à la deuxième solution envisagée. Suivant les idées de M. Fähndrich et A. Aiken présentées dans [10], nous étudions l'intégration d'une analyse grossière de la partie fonctionnelle (par unification) et d'une analyse précise de la partie concurrente (sous forme de contraintes ensemblistes) dans un seul système. Cette stratégie conduit donc à un travail d'intégration du système de type actuel avec les analyseurs de CAP développés par l'équipe.

Parallèlement, nous envisageons de développer le langage dans deux directions. Tout d'abord, l'intégration d'un système de compilation séparée afin de pouvoir utiliser les bibliothèques CaML en ML-ACT. Et, principalement, l'ajout au langage d'une notion de répartition qui permette de manipuler des sites. Le but de cet ajout étant d'obtenir un langage qui puisse servir de banc d'essai pour l'étude des pannes et de leurs conséquences. En effet, nous souhaitons définir des techniques

---

et des outils de validation de comportements dégradés de programmes repartis dans le contexte du modèle de programmation par acteurs.

## Références

- [1] G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, Cambridge, Mass., 1986.
- [2] J-L. Colaço. Analyses statiques d'un calcul d'Acteurs par typage. Thèse, Institut National Polytechnique de Toulouse, Octobre 1997.
- [3] J-L. Colaço, M. Pantel, P. Sallé, and A. Senteni. Un calcul d'acteurs primitifs (CAP). Dans *Actes des JFLA*, Janvier 1996.
- [4] J-L. Colaço, M. Pantel, and P. Sallé. CAP: An actor dedicated process calculus. In *ECOOP'96 Workshop on PTCOOP*, May 1996.
- [5] J-L. Colaço and M. Pantel and P. Sallé. A set-constraint-based analysis of Actors. In *Proc. of the 1997 IFIP International conference on FMOODS*, Chapman & Hall, July 1997.
- [6] J-L. Colaço and M. Pantel and P. Sallé. Analyse de linéarité par typage dans un calcul d'Acteurs. Dans *Actes des JFLA*, Janvier 1997.
- [7] J-L. Colaço and M. Pantel and P. Sallé. Static Analysis of Behavior Changes in Actor languages. In *Proc. of the 2<sup>nd</sup> France Japan Workshop on OBPDC*, October 1997.
- [8] F. Dagnat. Conception d'un langage Fonctionnel d'Acteurs et réalisation de son compilateur. Rapport de DEA, ENSEEIHT, Septembre 1997.
- [9] L. Damas and R. Milner. Principal type-schemes for fonctionnal programs. In *POPL'82*, ACM Press 1982.
- [10] M. Fähndrich and A. Aiken. Program Analysis Using Mixed Term and Set Constraints. In *SAS*, 1997.
- [11] C. Hewit, P. Bishop, and R. Steiger. An universal modular actor formalism for artificial intelligence. In *Proc. of the IJCAI'73*, 1973.
- [12] <http://pauillac.inria.fr/join/index.html>, documentation électronique.
- [13] X. Leroy. Typage polymorphe d'un langage algorithmique. Thèse, Université Paris VII, Juin 1992.

- 
- [14] R. Milner. A theory of type polymorphism in programming. In *Journal of Computer and System Sciences*,3(17) 1978.
  - [15] E. Najm and A. Nimour. A Calculus of Object Binding. In *Proc. of the 1997 IFIP International conference on FMOODS*, Chapman &Hall, July 1997.
  - [16] F. Nielson and H.R. Nielson. From CML to process algebras. In *CONCUR'93*, LNCS 715, 1993.
  - [17] H.R. Nielson., F. Nielson and T. Amtoft. Polymorphic Subtyping for Effect Analysis : the Integration ; the Semantics ; the Algorithm. 1996.
  - [18] Oscar Nierstrasz. Regular Types for Active Objects. In *Proceedings OOPSLA '93*, ACM SIGPLAN Notices, October 1993.
  - [19] M. Pantel. Représentation et Transformation : Un modèle de la réutilisabilité dans les langages fonctionnels à objets. Thèse, Institut National Polytechnique de Toulouse, Février 1994.
  - [20] Benjamin C. Pierce and David N. Turner. Concurrent Objects in a Process Calculus. In *TPPP, Sendai, Japan*, November 1994.
  - [21] Benjamin C. Pierce. Programming in the Pi-Calculus: An Experiment in Programming Language Design. Tutorial notes on the PICT language. Available electronically, 1994.
  - [22] J.H. Reppy. High-Order Concurrency. Ph.D thesis, Departement of Computer Science, Cornell University Ithaca, New York, June 1992.
  - [23] F. Rouaix. A web navigator with applets in Caml. In *Proc. of the Fifth International World-wide Web Conference*, Paris, France, Elsevier Science B. V., May 1996.
  - [24] António Ravara and Vasco T. Vasconcelos. Behavioural Types for a Calculus of Concurrent Objects. Euro-Par'97, LNCS, Springer-Verlag, 1997.
  - [25] B. Thansen, L. Leth, S. Prasad, T.M. Kuo, A. Kramer, F. Knahe and A. Giacalone. Facile antigua release programmation guide. Technical report ECRC 93-20, 1993.