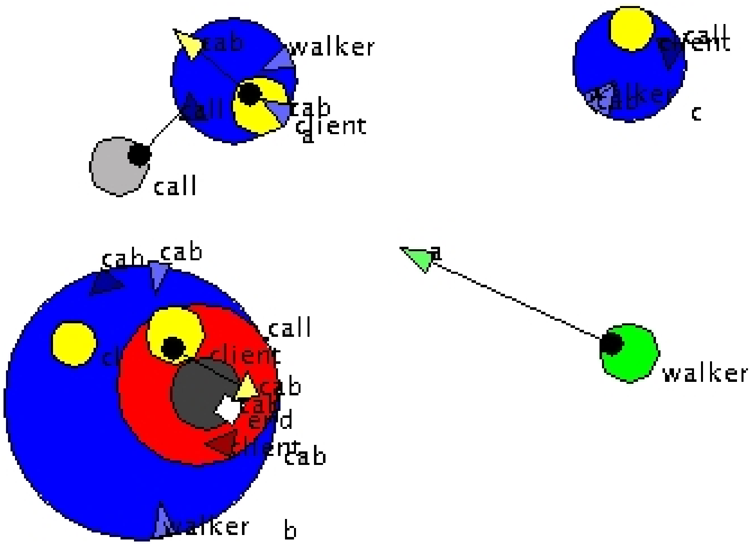
Les Mobile Ambients en Icobj

Damien Pous, **[dpous@ens-lyon.fr]**



Stage de mim1, Juin et Juillet 2002

Equipe MIMOSA
INRIA-Sophia Antipolis
Encadré par Frederic Boussinot et Pascal Zimmer

Table des matières

Introduction

Dans l'équipe MIMOSA (Migration Mobilité, Sémantique et Applications), on cherche à établir des langages, des principes facilitant la programmation répartie et la migration de code : comment faire pour qu'un *agent* puisse aller travailler sur différents *sites*, sans en tenir une copie sur chaque site ? Viennent ensuite les problèmes de sécurité associés : comment un site délivre-t-il la permission de rentrer à l'agent ? Il est donc interressant de pouvoir démontrer la *sécurité* d'un programme écrit dans les langages étudiés, d'où la nécessité d'une sémantique formelle bien précise.

Faire migrer un morceau de code à n'importe quel instant relève de l'utopie. Il nous faut s'assurer que le code est dans un état *cohérent*, *conciliant* avant de pouvoir le déplacer. C'est pourquoi la *Programmation Réactive* [BH96] introduit la notion d'*instants* : tous les processus s'exécutant partagent les mêmes instants, et assurent leur cohérence à la fin de ceux-ci.

De ce modèle, un langage de programmation graphique (et ludique) a été créé : les *Icobjs* [Bou96]. En associant une icône à un comportement on obtient un icobj, qui réagit dynamiquement avec tous les autres icobjs de la *Machine Réactive* .

Le calcul des *Ambients* [CG98] est un modèle de calcul distribué permettant de modéliser les réseaux (*internet...*). En effet, leur structure d'arbres dynamiques correspond bien à la structure administrative des réseaux et sous-réseaux : un ordinateur portable, branché sur *club-internet.fr* à son domicile, sera déplacé sous *monbureau.fr* au bureau... De plus, ce modèle fournit des outils permettant l'étude de la sécurité d'un système [TZH02, CGG99, DCS00].

Afin de mettre des images sur les mots, mon travail au sein de l'équipe était donc de produire un outil de visualisation des termes du calcul des Ambients, à partir des Icobjs.

1 De la Programmation Réactive aux Icobjs

1.1 La Programmation Réactive

La Programmation Réactive est une méthode de programmation distribuée, centrée sur la diffusion instantanée de messages : le temps est découpé en *instants* durant lesquels des messages peuvent êtres générés ou détectés par les différents processus réagissant en parallèle. On y trouve aussi un aspect *dynamique* : des processus peuvent être ajoutés ou retirés *presque* à tout moment.

Décrire toutes les ficelles de la programmation réactive n'est pas le but de ce rapport. J'essaie donc, en m'appuyant sur une implémentation : *les Scripts Réactifs*, d'en introduire les notions majeures qui nous serviront par la suite. Pour une description plus détaillée, voir [BH96].

Messages, Séquence, Parallèle

Les deux primitives permettant de diffuser et de détecter des messages sont : `generate E` et `await E`. Un message n'est émis que pendant un instant : il disparaît à l'instant suivant, à moins d'y être réémis. Si ces deux termes sont exécutés au même instant, il y aura *toujours* détection. On notera qu'un message ne peut être émis qu'une fois par instant ; si plusieurs processus tentent d'émettre le même message, seul le *premier* l'émettra réellement...

Deux termes peuvent être mis en *séquence* : `P ; Q`. La séquence est immédiate, elle ne fait pas intervenir de nouvel instant. Le terme `generate E ; await E` sera donc évalué en un seul instant, et `E` sera détecté.

L'opération duale de la séquence est la mise en *parallèle* : `P || Q`, cette expression se terminant une fois `P` *et* `Q` terminés. Aucune prédiction ne peut être faite sur l'ordre d'évaluation des deux expressions au sein de l'instant, d'où un certain indéterminisme :

```
generate E ; (await E ; print a || await E ; print b)
```

peut afficher “ab” comme “ba”...

La dernière commande de base est `stop`, elle attend l'instant suivant. De cette façon, on peut écrire un terme qui attend deux occurences distinctes d'un message : `await E ; stop ; await E`

Configurations

Plus que des messages, on peut détecter des *configurations* : les opérateurs `or`, `and` et `not` permettent de tester, par exemple la présence simultannée de `A` et `B` : `await (A and B)`, à différencier de `await A || await B`, qui attend bien d'avoir détecté `A` et `B`, mais pas nécessairement lors du même instant.

L'opérateur `not` est particulier : il détecte l'absence d'un message. Pour ce faire il doit donc être certain que le message ne sera plus généré dès lors qu'il a *répondu*. D'où le paradoxe : `await not A ; emit A`, où `A` ne peut-être ni présent ni absent durant l'instant considéré. La décision de l'absence d'un message doit donc être prise à la fin de l'instant, c'est à dire, en pratique juste avant l'instant suivant.

Boucles

Pour que le langage soit complet, il lui faut nécessairement des boucles, infinies comme finies. Celles-ci s'écrivent `loop P end` et `loop n times P end`. Tout comme la séquence, les boucles ne consomment pas d'instant : `loop 3 times print a end` affichera trois "a" lors d'un seul instant, à la différence de `loop 3 times print a ; stop end` qui affichera trois "a" mais en trois instants.

De ce fait, la boucle infinie peut poser problème : `loop print a end` tenterait d'afficher "a" une infinité de fois pendant un seul instant, et bloquerait donc la machine dans cet instant. La solution adoptée dans les scripts réactifs consiste à vérifier à l'exécution que l'on ne passe pas deux fois dans une boucle infinie au sein du même instant, et, le cas échéant, à rajouter le `stop` manquant.

Objets

Les scripts réactifs contiennent aussi la notion d'objet, que je simplifie ici au maximum : un objet contient un processus, qui s'exécute durant toute la durée de vie de l'objet, ainsi que des méthodes (asynchrones). Ils peuvent êtres *détruits* : `destroy x` ou *gelés* : `freeze x`. Ces deux actions n'agissent qu'à la fin de l'instant ; la première retire purement et simplement l'objet tandis que la seconde le retire, mais calcule aussi l'objet *résidu* correspondant à *ce qu'il reste à faire*. Par exemple, le résidu de l'objet

```
   loop 5 times print a ; stop ; end
|| loop generate O ; stop ; end
|| await A ; print b ; await B ; generate C
```

peut être, selon l'instant du gel, l'objet

```
   loop 3 times print a ; stop ; end
|| loop generate O ; stop ; end
|| await B ; generate C
```

Ainsi l'objet gelé pourra être réintégré plus tard (ou ailleurs), sans qu'il n'ait rien vu. C'est ainsi qu'est réalisée la *migration* !

1.2 Junior

Junior est une implémentation, en Java, de ce langage.

À chaque élément de la syntaxe (`||`, `;`, `await`...) correspond une classe (*Par*, *Seq*, *Await*...) héritant de la classe abstraite *Program*.

Le terme `await A || loop generate B ; stop end` s'écrira donc

```
new Par(new Await('A'),
        new Loop(new Seq(new Generate('B'),
                         new Stop())))
```

Un objet *Machine* permet alors d'enregistrer des programmes (méthode *register*), puis de les faire réagir (méthode *react*), instant par instant.

Exécution de code *externe*

La classe *Atom*, fille de *Program*, est particulière : elle permet d'exécuter une *action*, c'est à dire du code Java, de manière atomique. L'affichage à l'écran peut se faire de cette manière : on définit une action *Print* (classe qui implémente l'interface *Action*) qui consiste à afficher du texte (*System.out.println(txt);*) que l'on peut alors utiliser comme suit :

new Par(new Atom(new Print('aa')), new Atom(new Print('bb')))

Les seuls affichages possibles seront *'aabb'* et *'bbaa'*.

Scanner

Junior étend légèrement le modèle présenté ci dessus en permettant de lier un objet (au sens Java) à l'émission d'un message. Lorsque le même message est émis plusieurs fois au sein d'un même instant, la machine garde une liste de tous les objets qui ont été associés aux émissions, ce qui permet une nouvelle instruction : le *Scanner*. Celle-ci attend un message E, et, pour chaque objet associé à ce message durant l'instant, exécute une action donnée.

Dans l'exemple suivant, l'affichage à l'écran se fait par l'émission du message *'PRINT'*, associé à la chaîne de caractères à imprimer :

La classe *ScanPrint* (qui implémente l'interface *ScanAction*) affiche l'objet associé au message (*System.out.println(obj);*),

$$
\begin{aligned}
&new\ Par(new\ Scanner('PRINT',\ new\ ScanPrint()),\\
&new\ Par(new\ Seq(new\ Await('E'),\\
&\qquad\qquad\qquad new\ Generate('PRINT',\ 'World')),\quad {}^{1}\\
&\qquad\quad new\ Seq(new\ Generate('PRINT',\ 'Hello\ '),\\
&\qquad\qquad\qquad new\ Generate('E'))))
\end{aligned}
$$

Avantage

L'avantage majeur de cette technique de programmation *concurrente* provient du fait qu'elle ne fait intervenir qu'un seul *thread*. En effet, lors d'une réaction de la machine, celle-ci effectue tout ce qu'il y a à faire durant l'instant (émissions de messages, détections, actions atomiques...), jusqu'à ce qu'il n'y ait plus rien à faire (et ce, de manière séquentielle). Ainsi, deux processus mis en *parallèle* ne risquent pas d'entrer en conflit au niveau des données qu'ils partagent, contrairement au modèle habituel des threads. On évite de la sorte tous les problèmes de verrous et de *dead-lock*, si pénibles à mettre en place, puis à déboguer...

Consulter [BS00, Bou00, HSB99] pour une description détaillée de cette implémentation.

1.3 Les Icobjs

C'est de l'association d'une icône à un comportement, c'est à dire un programme réactif, que sont nés les *Icobjs*. Placés ensembles (en parallèle) au sein d'une machine réactive, ils interéagissent tout étant affichés à chaque instant dans une fenêtre graphique.

Des comportements

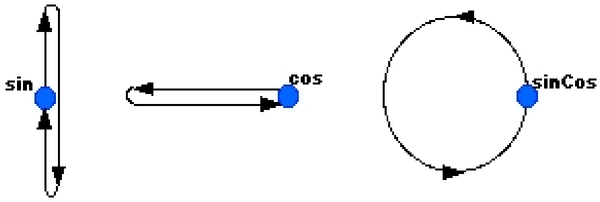
Un comportement peut par exemple consister à se déplacer vingt fois d'un pixel vers la droite. L'icobj obtenu se déplacera donc vers la droite, puis s'arrêtera. Mis en séquence avec le comportement *opposé* (vers la gauche), l'icobj fera un aller-retour vers la droite, avant de s'arrêter. Enfin, si ce comportement est placé dans une boucle infinie, l'icobj ne cessera de faire des aller-retour.

Un autre exemple *standard* est celui du cercle. On définit deux comportements :

- SIN : déplacement en sinus t selon l'axe des ordonnées,
- COS : déplacement en cosinus t selon l'axe des abscisses,

[1] L'affichage 'WorldHello ' est ici possible, bien que $Generate('PRINT', 'Hello\ ')$ soit toujours exécuté avant $Generate('PRINT', 'World')$. En effet, il s'agit de deux émissions d'un même message $('PRINT')$ au cours du même instant.

La mise en parralèle de ces deux comportements donnera comme on s'y attend, un icobj décrivant une trajectoire circulaire. [2]



D'autres comportements basiques sont le changement d'icône, le redimensionnement, la réaction au contrôle de la souris (déplacement et redimensionnement *manuels*), l'émission d'un message...

Généralement, on dote aussi les icobjs d'une masse et d'une vitesse, ce qui permet de nouveaux comportements :
- inertie (conservation de la vitesse)
- rebond sur les bords
- collision avec les autres icobjs (boules de billard)
- prédateur (accélération vers tout émétteur d'un message *proie*)
- ...

Enfin, des icobjs plus complexes peuvent être construits, comme par exemple, les élastiques. Attaché à deux icobjs, un élastique excerce une force à chacune de ses extrêmités, selon les lois de la physique. De la même manière, on construit des tiges rigides...

Un constructeur

Mais comment programme l'utilisateur (qui se refuse à rentrer la moindre ligne de code...) ?

Ce dernier a à sa disposition un icobj très particulier : le constructeur. C'est cet icobj qui permet la création d'icobjs *nouveaux* par l'utilisateur :

Le constructeur permet tout d'abord de cloner le comportement d'un icobj (sans son icône) : il suffit de déposer le constructeur sur l'icobj à cloner, de cliquer dessus, de déplacer le constructeur dans une zone *vide*, et de recliquer dessus pour créer un nouvel icobj.

[2] Il est impossible de réaliser ceci avec deux *thread*, sans qu'ils soient synchronisés ; en effet, les temps accordés par le système aux deux threads n'étant pas *controlés*, ils se décaleraient...

Il permet ensuite de mettre les comportements de plusieurs icobjs en séquence : on clique avec le constructeur sur chaque icobj de la séquence, puis sur une zone *vide*. Un icobj particulier permet de mettre en boucle la séquence ainsi construite.

Enfin, il permet la mise en parrallèle des comportements de plusieurs icobjs, en le plaçant sur tous les icobjs à *parralléliser* (il peut être utile de le redimensionner)

Une démonstration !

Ce sont surtout l'aspect ludique, et l'impact visuel qui importent dans les icobjs... Un schéma vaut mieux qu'un long discours, je vous conseille donc le tutoriel et les exemples présentés sur

http ://www-sop.inria.fr/mimosa/rp/Icobjs/IntroToIcobjs/index.html.

2 Le calcul des Ambients

Parallèlement aux développements du matériel et des logiciels, des théories on été formulées pour comprendre et pour utiliser au mieux les nouveaux moyens offerts par la technologie. Le formalisme permet alors de dire ce qu'il est possible ou impossible de faire en utilisant une certaine technologie. Il permet aussi d'établir de nouveaux algorithmes, d'en établir leur complexité, et de démontrer leur validité.

Ainsi, le *Lambda Calcul* permet de formaliser le calcul séquentiel, calcul qui est utilisé à l'intérieur de nos ordinateurs. Le Lambda Calcul est utile par exemple, pour montrer qu'un problème particulier ne sera jamais résolu par un algorithme.

Avec les avancées de la technologie actuelle en matière de réseau, il est nécessaire d'introduire de nouvelles théories. Le modèle de référence pour le calcul distribué est le *Pi-Calcul*, introduit par Robin Milner [Mil00]. Ce modèle est basé sur l'émission de messages à travers des *canaux nommés*. Le calcul des Ambients un peu plus récent, fut introduit par Luca Cardelli et Andrew D. Gordon [CG98].

2.1 Les Mobile Ambients

Les Mobiles Ambients constituent l'essence du calcul des Ambients, toutes les autres variantes en dérivent. C'est un calcul basé sur la notion de *localité* et de *mobilité*.

Chaque ambient, noté $n[P]$, est une localité portant un nom (n), qui contient des *sous-localités* et des *capacités*, qui réagissent en parallèle (P). Ce sont ces capacités qui induisent la mobilité au sein de l'arbre ainsi contruit. Elles correspondent chacune à une volonté de mouvement particulier, et maintiennent un *futur bloqué* derrière elles :

- *in* m : l'ambient veut devenir une sous-localité de m, lorsqu'il en est le frère.
- *out* m : l'ambient veut sortir de la localité m, lorsqu'il en est le fils.
- *open* m : veut ouvrir la localité m, lorsqu'il en est le père. Tout le contenu de m (ambients *et* capacités) sera alors libéré dans l'ambient père, et l'ambient m disparaît.

Lorsqu'une capacité est *consommée*, c'est à dire que le mouvement a eu lieu, elle est détruite, et son futur est *libéré* dans l'ambient concerné. Dans les figures suivantes, le futur est représenté par un carré jaune.

La syntaxe usuelle est décrite par les règles suivantes :

$$
\begin{array}{lcll}
P & ::= & n[P] & //\ \textit{un ambient} \\
 & & P|P & //\ \textit{mise en parallèle} \\
 & & 0 & //\ \textit{feuilles de l'arbre} \\
 & & M.P & //\ \textit{une capacité} \\
 & & (P) & //\ \textit{parenthésage} \\
 & & & \\
M & ::= & \textit{in } n & \\
 & & \textit{out } n & \\
 & & \textit{open } n &
\end{array}
$$

La mise en séquence $(M.P)$ a priorité sur la mise en parallèle ; on s'autorise les racourcis $n[]$ et M pour désigner $n[0]$ et $M.0$.

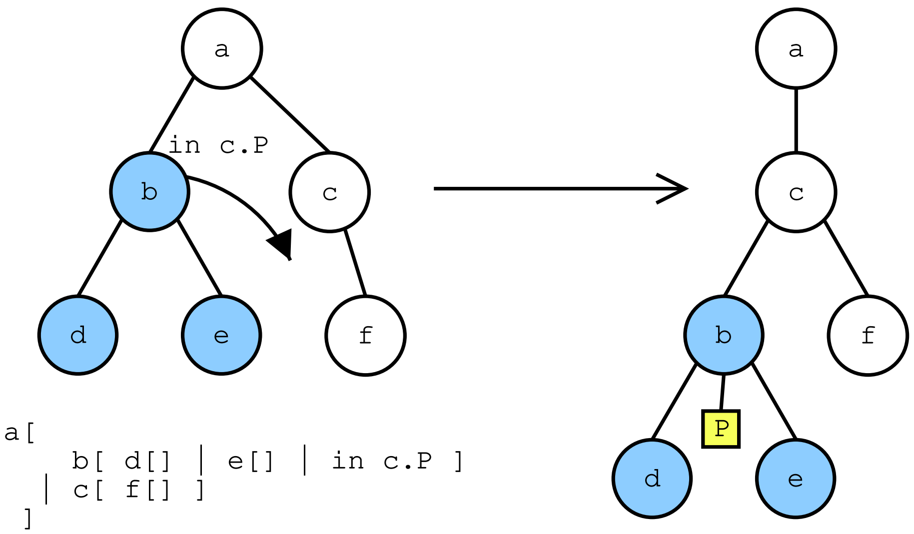


Fig. 1 – consommation d'une capacité de type IN

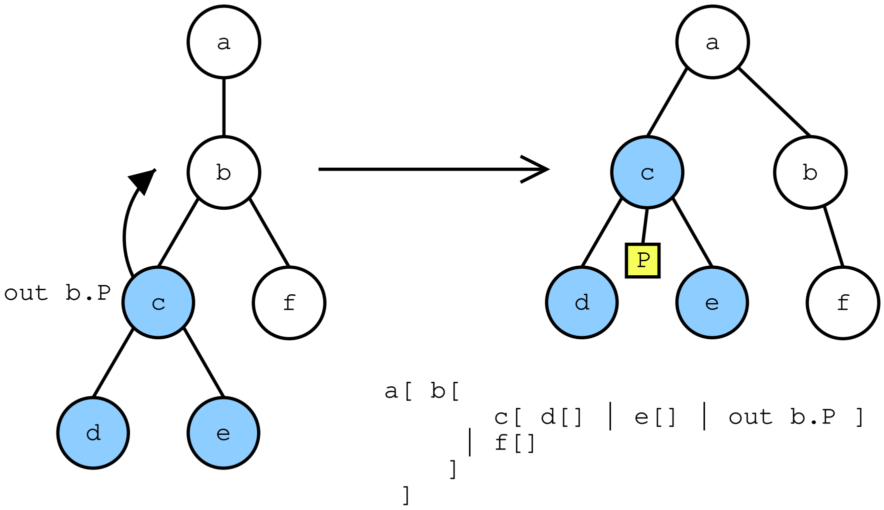


FIG. 2 – consommation d'une capacité de type OUT

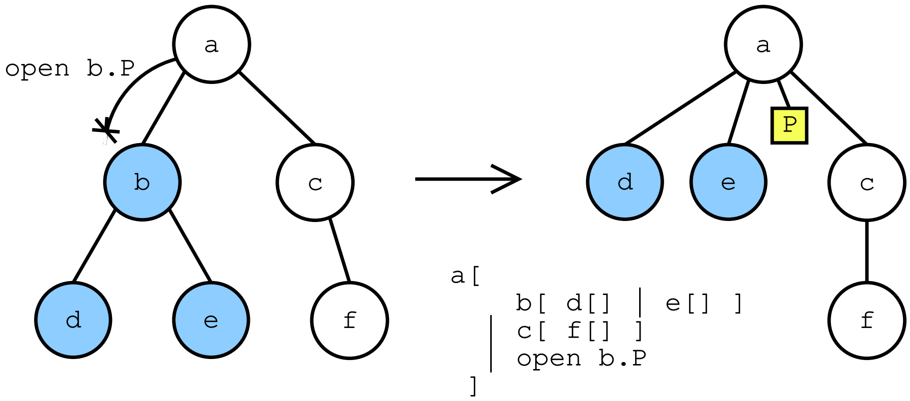


FIG. 3 – consommation d'une capacité de type OPEN

Il est important de remarquer que l'action des capacités est *locale*. Ainsi les mouvements dans deux branches *distinctes* de l'arbre sont totalement indépendants, ce qui confère au modèle son aspect *distribué*...

Comme indiqué dans les règles de la syntaxe, le futur d'une capacité peut être soit un ambient, soit une autre capacité, ce qui permet à un ambient de suivre un *chemin* dans l'arbre, comme le montre la figure 4.

De plus, l'ordre des consommations (réductions) a son importance, d'où un calcul *non déterministe* : Considérons l'ambient $p[a[in\ b.P] \mid b[in\ a.Q]]$. Selon la règle appliquée en premier, il se réduira en $p[a[b[Q] \mid in\ b.P]]$ ou $p[b[a[P] \mid in\ a.Q]]$ et dans les deux cas, la seconde capacité sera bloquée. On a donc construit un ambient qui réalise un ou exclusif, non déterministe.

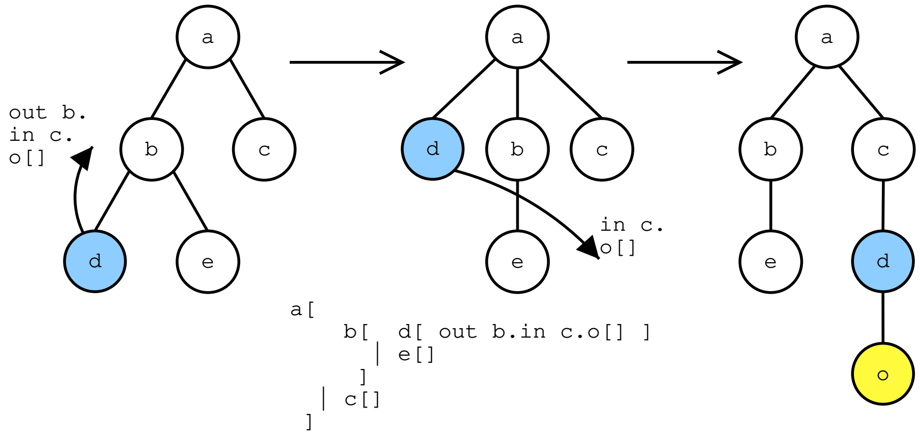


FIG. 4 – séquence de capacités

Rien n'interdit à deux ambients de porter le même nom, on peut ainsi écrire cette opération comme suit :

$$\begin{array}{ll} p[& open\ o.u[] \\ & \mid o[open\ u.P] \\ & \mid o[open\ u.Q] \\ ] & \end{array}$$

2.2 Puissance du modèle

Les Mobile Ambients étant un modèle de calcul, la moindre des choses est qu'il soit *Turing-puissant*. Tel que présenté ci-dessus, toute suite de réductions sur un terme de taille finie est finie, et ne peut augmenter la taille du terme. Le calcul s'arrête donc sur tout terme sur un terme de taille inférieure ou égale... Il est donc peu probable que l'on ait la Turing-puissance ! Deux éléments sont donc rajoutés au modèle :

– Un *opérateur de récursion*, $Rec\ X.P$
  Toute occurence de la variable X dans le terme P peut se réduire en P : $Rec\ X.a[X]$ fournit un arbre peigne étiqueté a, aussi long que voulu. De même, L'ambient $a[Rec\ X.in\ b.X]$ pourra rentrer dans tout ambient nommé b, et ce autant de fois qu'il en rencontrera.

– Un *créateur* de noms *frais*, $(\nu n)P$
  Toute occurence du nom n dans P est remplacée par un nom n'ayant jamais été mentionné. Ainsi, dans le terme $p[(\nu o)(o[P] \mid M.open\ o) \mid Q]$, seul l'ambient p peut ouvrir l'ambient o et libérer P (une fois M terminé), puisque le terme Q ne connaît pas ce nom.
  Conjugué avec l'opérateur de récursion, il permet la création d'une infinité de noms : $rec\ X.(X \mid (\nu a)a[P])$ fournit autant d'ambients que

l'on souhaite, portant tous des noms distincts.

Ces deux opérations suffisent à rendre le modèle Turing-puissant : on encode les entiers par des arbres peignes

$$\begin{array}{lcl} Zero & := & zero[Rec\ X.in\ succ.X] \\ Succ & := & succ[Rec\ X.in\ succ.X] \\ Dec & := & open\ succ \end{array}$$

Le test d'égalité à zéro s'écrit

$$\begin{array}{lcl} ifZero\ P\ Q & := & (\nu u, o)(\\ & & \quad u[in\ zero.out\ zero.o[out\ u.P]] \\ & & \mid u[in\ succ.out\ succ.o[out\ u.Q]] \\ & & \mid open\ o \\ & &) \end{array}$$

Avec l'opérateur de récursion, on peut donc *calculer* toute fonction Turing-calculable.

On peut aussi montrer que ce modèle est aussi expressif que le modèle du Pi-Calcul, en utilisant un ambient pour encoder un canal nommé. Pour plus de détails, consulter [Zim]

2.3 Modèles dérivés

Le problème de ce modèle est la difficulté d'y modéliser certains problèmes courant des réseaux : sécurité, restriction d'accès et contrôle des ressources.

Considérons l'exemple standard du *firewall* : on désire limiter l'accès à un réseau R aux seules personnes connaissant le mot de passe *titi*. Une solution simple consiste à cacher le réseau dans un ambient nommé *titi* : $titi[R]$. [3]

Mais comment faire pour que seules les personnes nommées *toto* puissent entrer ? La solution nécessite déjà l'utilisation du ν : le réseau est placé dans un ambient dont le nom est maintenu secret ($(\nu system)system[P\ ...]$), ainsi personne ne peut rentrer directement, mais le système libère des ambients dont le nom est le mot de passe, et qui souhaitent rentrer chez toute personne autorisée à pénétrer dans le réseau : $titi[out\ system.in\ toto\ ...]$. Un tel ambient pourra alors être ouvert par cette personne, libérant ainsi la clef du

[3] *titi* étant considéré comme un *bon* mot de passe...

système : $in\ system$.

$$
\begin{array}{lll}
Client & ::= & toto[open\ titi] \\
System\ R & ::= & (\nu system)system[\\
 & & \quad R \\
 & & \quad \mid Rec\ X.(X \mid titi[out\ system.in\ toto.in\ system]) \\
 & &]
\end{array}
$$

Bien que ce soit plus compliqué, on arrive encore à nos fins. Par contre il nous est impossible de limiter le nombre d'accès simultanés au réseau : lorsque l'on donne au client la capacité de sortir, le système est incapable de déterminer quand a lieu l'éventuelle sortie.

2.3.1 Les Robust Ambients

Dans le modèle des *Robust Ambients*, on donne à un ambient la possibilité de refuser des mouvements. En effet, ceux ci doivent donner explicitement leur accord à l'aide de trois *co-capacités* :

- $\overline{in}\ m$: l'ambient autorise m à entrer,
- $\overline{out}\ m$: l'ambient autorise m à sortir,
- $\overline{open}\ m$: l'ambient autorise m à l'ouvrir.

Comme le montrent les figures 5, 6 et 7, ces co-capacités sont utilisées et consommées comme les capacités *standard*.

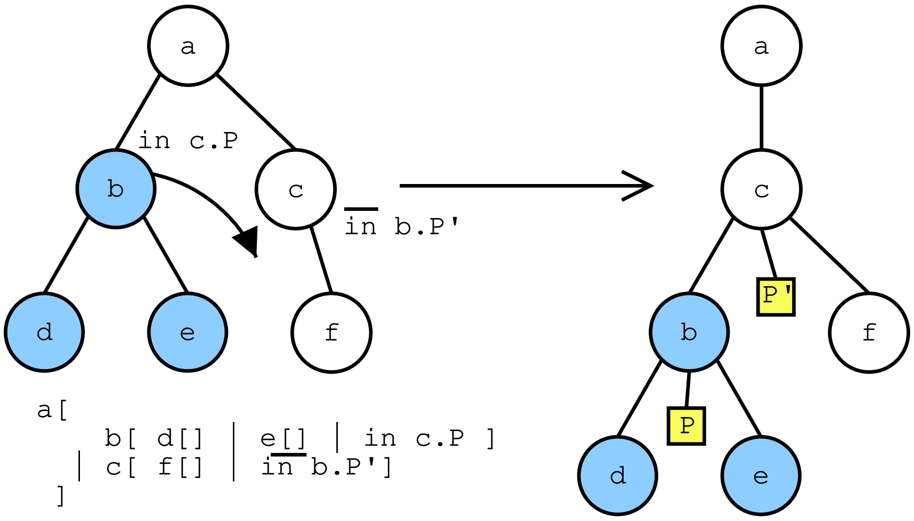


FIG. 5 – Robust Ambients : réduction IN

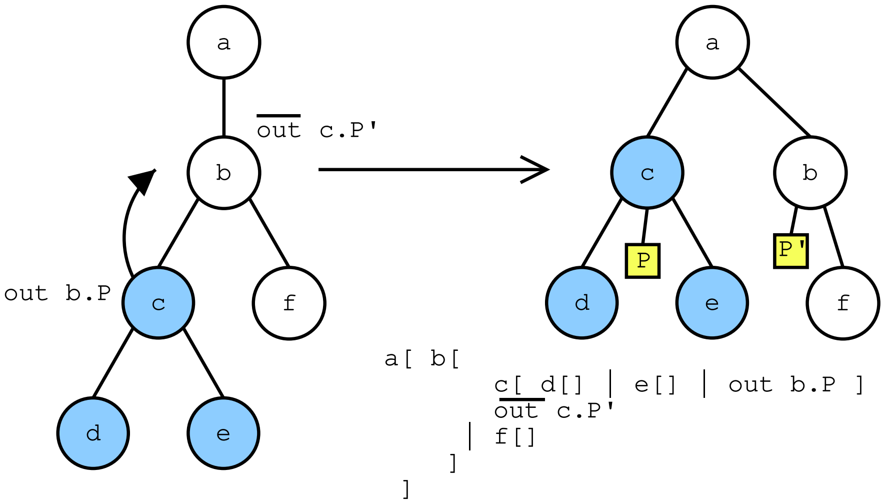
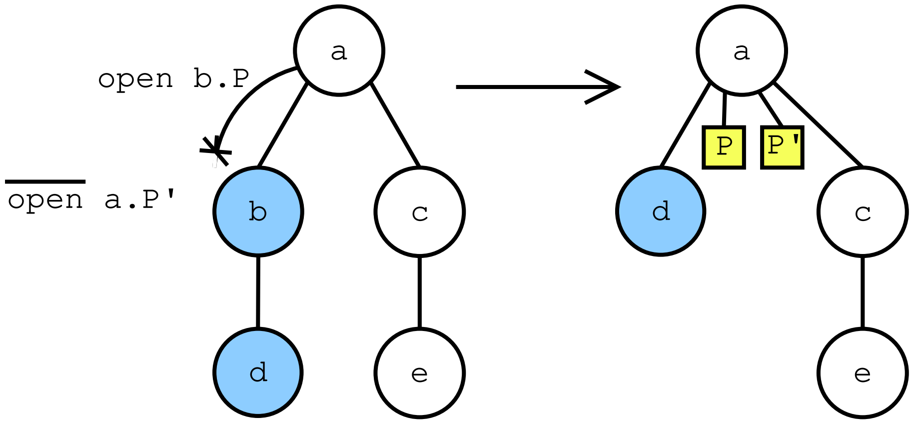


FIG. 6 – Robust Ambients : réduction OUT

On peut ainsi réécrire le firewall plus simplement (et en autorisant la sortie...) :

$$
\begin{array}{lcl}
\mathit{Client} & ::= & \mathit{toto}[\mathit{in}\ \mathit{titi}] \\
\mathit{System}\ R & ::= & \mathit{titi}[\mathit{rec}\ X.\overline{\mathit{in}}\ \mathit{toto}.X \mid \mathit{rec}\ X.\overline{\mathit{out}}\ \mathit{toto}.X\ ]
\end{array}
$$



$$\begin{array}{l}
\texttt{a[} \\
\quad \texttt{c[e[]]} \\
\quad | \ \texttt{open b.P} \\
\quad | \ \texttt{b[d[] |} \ \overline{\texttt{open}}\ \texttt{a.P']} \\
\texttt{]}
\end{array}$$

FIG. 7 – Robust Ambients : réduction OPEN

Pour limiter le nombre de clients simultanés, Il suffit alors de lier l'autorisation d'entrée à l'autorisation de sortie :

$$\begin{array}{lcl}
Client & ::= & toto[in\ titi.out\ titi] \\
System\ R & ::= & titi[\\
& & \quad\ \ rec\ X.\overline{in}\ toto.\overline{out}\ toto.X \\
& & \quad \mid rec\ X.\overline{in}\ toto.\overline{out}\ toto.X \\
& & \quad \mid rec\ X.\overline{in}\ toto.\overline{out}\ toto.X \\
& &]
\end{array}$$

de la sorte, le système ne peut pas contenir plus de trois clients *toto* simultanément.

2.3.2 Les Controlled Ambients

Le modèle précédent permet à chaque ambient de connaître la taille de sa descendance. En effet, cette taille ne peut changer sans qu'une (co-)capacité ait été consommée. Cependant, on n'y contrôle pas le nombre de fils directs d'un ambient ; par exemple, avec $a[b[in\ c] \mid c[\overline{in}\ b]]$, l'ambient a n'est pas averti de la perte de son fils b.

Le modèle des *Controlled Ambients*, en renforçant encore plus les autorisations à réunir, palie à ce manque, tout en augmentant la sécurité du système.

Les trois co-capacités des Robust Ambients sont remplacées par cinq co-capacités :

- $\overline{in_{\downarrow}}\ m$: l'ambient autorise m à rentrer *d'en haut*, c'est l'équivalent du $\overline{in}$ des Robust Ambients.
- $\overline{out_{\downarrow}}\ m$: l'ambient autorise m à partir *vers le bas*,

- $\overline{out_{\uparrow}}\ m$: l'ambient autorise m à sortir *vers le haut*, c'est l'équivalent du $\overline{out}$ des Robust Ambients.
- $\overline{in_{\uparrow}}\ m$: l'ambient autorise m à rentrer *d'en bas*,

- $\overline{open}$ (m,n) : l'ambient, s'il est nommé m, autorise n à l'ouvrir.

L'opération IN nécessite ainsi deux autorisations : celle du père ($\overline{out_{\downarrow}}\ m$) et celle du fils ($\overline{in_{\downarrow}}\ m$).

De même l'opération OUT nécessite l'autorisation du père ($\overline{out_{\uparrow}}\ m$) et celle du grand-père ($\overline{in_{\uparrow}}\ m$).

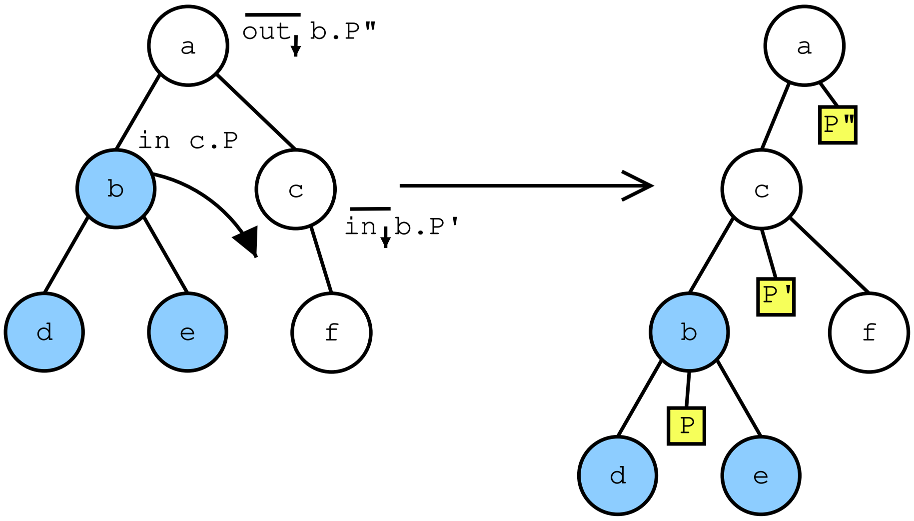


Fig. 8 – Controlled Ambients : réduction IN

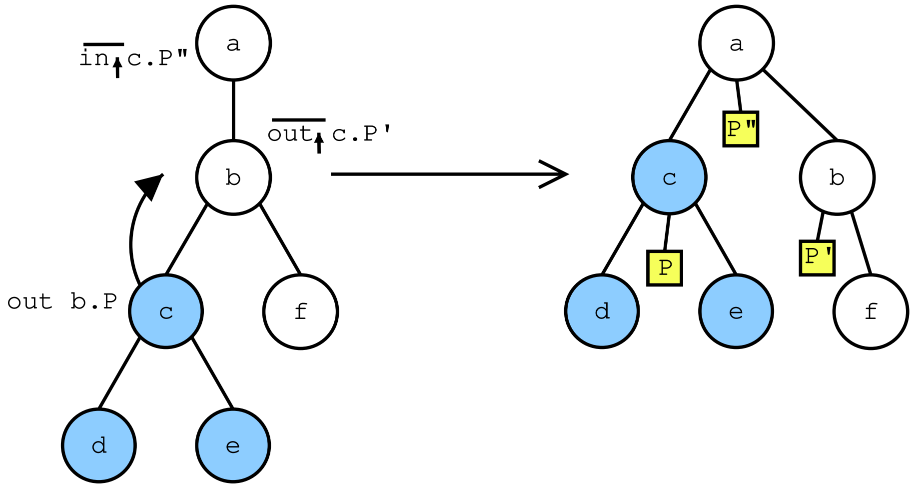


FIG. 9 – Controlled Ambients : réduction OUT

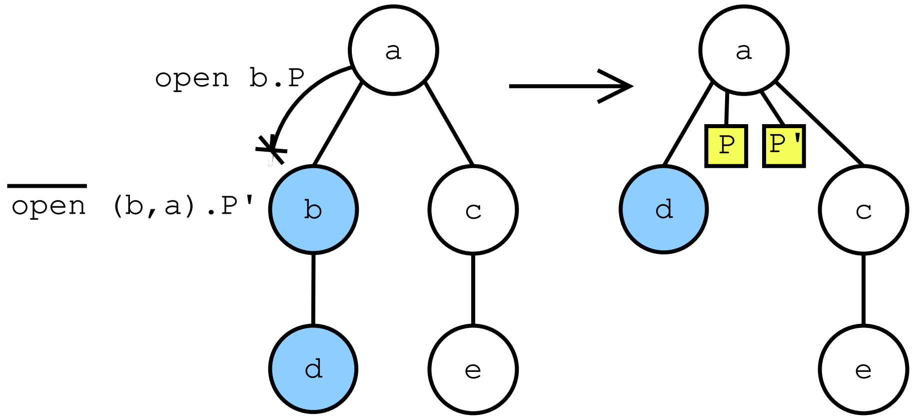


FIG. 10 – Controlled Ambients : réduction OPEN

Ces règles permettent la connaissance du nombre de fils directs d'un ambient, *à l'exécution*. Il est alors possible d'établir des algorithmes de typage *statique*, permettant d'affecter à chaque ambient d'un terme une *capacité*, c'est à dire un nombre maximal d'ambients qu'il pourra contenir. Par exemple, l'ambient $cell[Rec\ X.\overline{in_\downarrow}\ m.\overline{out_\uparrow}\ m.X]$ a une capacité égale à un : seul un ambient nommé m peut résider dans *cell*. Par contre, la capacité de $big[rec\ X.(a[]|X)]$ est infinie !

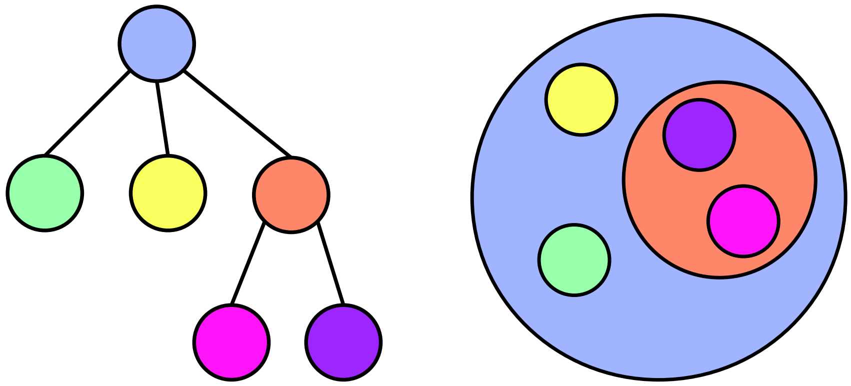
De tels types sont évidemment bien utiles pour l'étude des propriétés des programmes, cependant la détermination d'un type *minimal* dans le cas général s'avère être difficile ; à moins de fixer initialement le type des variables de récursion. [TZH02]

3 Mon implémentation

130 fichiers, 8000 lignes de code, j'éviterai donc une description ligne à ligne du code source... Si toutefois quelqu'un souhaitait le réutiliser, qu'il n'hésite pas à me contacter pour de plus amples explications...

3.1 Structure arborescente : les ambients

Le tracé *à l'informaticienne* d'un arbre n'est pas adapté aux icobjs. Je représente donc les ambients par des disques, pouvant en contenir d'autres. Ainsi, l'ambient $a[\ b[] \mid c[] \mid d[\ e[] \mid f[]\ ]\ ]$, tracé de manière habituelle à gauche, sera représenté comme sur la figure de droite



À un ambient correspond donc un icobj, dont l'icône est un disque de couleur donnée. [4]

Cet icobj se redimensionne automatiquement lorsque des fils lui sont retirés, ajoutés, ou que leur taille change... La formule donnant le rayon en fonction du contenu dépend du nombre de fils, ainsi que de la somme de leurs surfaces.

Il possède aussi une masse ; afin que les opérations IN et OUT soient *conservatives*, celle-ci est prise égale à $2n + 1$ où n est le nombre de descendants.

Voici les différents comportements qui les animent :
- inertie,
- rebond sur les bords,
- rebond à l'intérieur du père (collision élastique),
- collision avec ses frères (collision élastique, sphérique),
- réponse au contrôle de la souris.

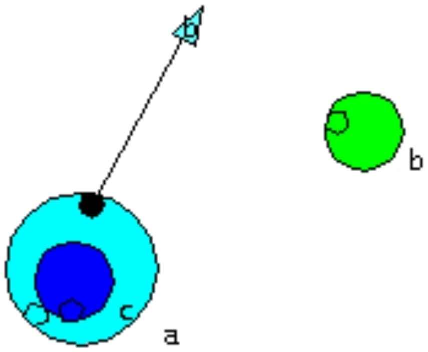
[4] on peut préciser cette couleur par la syntaxe $a\%couleur[]$

Ces icobjs sont particuliers car ils ne respectent pas la procédure habituelle d'affichage : En effet, la machine affiche à chaque instant, l'ensemble des icobjs enregistrés, via la méthode $render()$. L'ordre d'affichage n'est donc pas contrôlé, et des ambients pourraient être cachés par leur père ! En pratique, l'affichage d'un ambient est réalisé de manière récursive, en partant des ambients *libres*, c'est à dire sans père.

3.2 Dynamicité : les capacités

Les Mobile Ambients

La capacité IN est représentée par une flèche constituée de trois icobjs :
- une tête chercheuse, qui se déplace dans l'ambient père, en recherchant tout ambient frère portant le bon nom,
- une ancre, qui reste au sein de l'ambient,
- un élastique, qui relie les deux icobjs précédents.



Capacité IN : $a[\ in\ b \mid c[]\ ] \mid b[]$

Les petits pentagones que l'on aperçoit sur chaque ambient de la figure ci-dessus sont en fait des icobjs *portes*. Ces icobjs émettent à chaque instant un message permettant aux têtes chercheuses de les repérer.

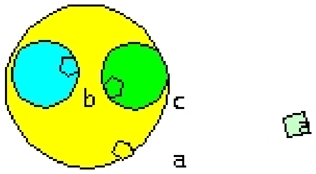
Une fois la porte atteinte, l'élastique se *renforce* pour raprocher les deux ambients. Lorsque l'ancre et la porte sont suffisament proches, le IN est effectué.

Si le IN est effectué de manière atomique, au sein d'un seul instant, grâce au modèle mono-thread des icobjs, l'opération dans sa globalité ne l'est pas : elle s'étend sur plusieurs instants. Il est donc important de s'assurer à chaque instant que la porte *harponnée* correspond toujours à un frère portant le bon nom. En effet, sur l'exemple $a[\ in\ b \mid in\ c\ ] \mid b[] \mid c[]$, les deux capacités peuvent être accrochées aux portes de a et b, durant les mêmes instants, mais en ne doivent en aucun cas être toutes deux consommées...

L'opération OUT s'effectue de manière analogue, à la forme de la tête près...

Capacité OUT : $a[\ b[out\ a] \mid c[]\ ]$

L'opération OPEN est légèrement plus simple : la capacité est représentée par un icobj carré qui se déplace dans l'ambient, à la recherche d'une porte *valide*. Lorsqu'il la touche, l'opération est effectuée.



Capacité OPEN : $a[\ b[] \mid c[]\ ] \mid open\ a\ ]$

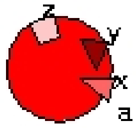
Seul l'usage des capacités distinguant les différents modèles de calcul, celles-ci sont mises à part dans le code. Par le jeu des classes abstraites Java, pour obtenir un modèle, il suffit de définir :
- un moyen de *parser* des chaînes de caractère, c'est à dire, en extraire un arbre représentant l'expression (type *Expr*),
- les méthodes d'action des capacités.

Les Robust Ambients

Dans le modèle précédent, les portes sont *universelles*, et sont crées automatiquement par le programme. De plus, lorsqu'une porte est utilisée, une nouvelle est automatiquement créée, au cas où d'autres mouvements seraient possibles.

Le modèle des Mobile Ambients peut donc être vu dans le monde des Robust Ambients comme un modèle où tout ambient accorde toutes les permissions : $a[P] \equiv a[Rec\ X.\overline{in}\ \star.X \mid Rec\ X.\overline{out}\ \star.X \mid Rec\ X.\overline{open}\ \star.X \mid P]$

Pour obtenir les Robust Ambients, il suffit donc de spécialiser les portes précédentes, et ne les mettre que sur demande.

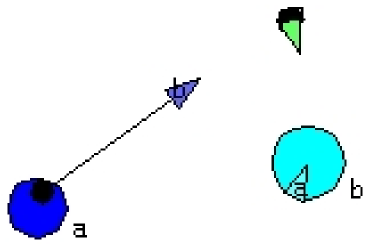


Les trois co-capacités des Robust Ambients : $a[\ \overline{in}\ x \mid \overline{out}\ y \mid \overline{open}\ z\ ]$

Les Controlled Ambients

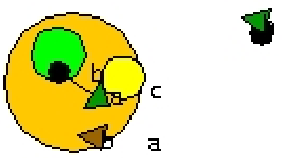
Les opérations IN et OUT des Controlled Ambients sont plus techniques. En effet, elles nécessitent deux autorisations. Or, obtenir la collision simultannée de trois icobjs n'est pas simple... Il faut donc garder l'idée de la tête chercheuse, mais qui doit passer successivement par les deux autorisations, tout en sachant que tant que la seconde autorisation n'a pas été passée, la première ne doit pas être bloquée. Considérons par exemple l'ambient $a[\overline{in_\downarrow}\ b] \mid \overline{out_\downarrow}\ b \mid b[\ in\ a \mid in\ c\ ]$ si la capacité *in c* passait et bloquait l'autorisation $\overline{out_\downarrow}\ b$ avant la capacité *in a*, cette dernière ne pourrait jamais être consommée, alors que c'est la seule à devoir l'être !

Au niveau de la représentation, le $\overline{in_\downarrow}$ (qui correspond au $\overline{in}$) est toujours représenté par une porte *entrante*, mais la tête chercheuse doit passer au paravant par l'autorisation $\overline{out_\downarrow}$, représentée par un *trou*. Ce trou peut être passé par plusieurs capacités, mais dès que l'une d'elle concrétise son mouvement (IN), le trou est désactivé, et les autres capacités doivent en retrouver un...



Opération IN : $a[\ in\ b\ ] \mid \overline{out_\downarrow}\ a \mid b[\ \overline{in_\downarrow}\ a\ ]$

Pour l'opération OUT, c'est la même chose, mais le rôle de la porte et du trou sont inversés : la porte ($\overline{out_\uparrow}$) doit être passée en premier, et c'est le trou $\overline{in_\uparrow}$ qui est accroché.



Opération OUT : $a[\ \overline{out_\uparrow}\ b \mid b[\ out\ a\ ] \mid c[]\ ] \mid \overline{in_\uparrow}\ b$

3.3 Extensions : les pseudo-capacités

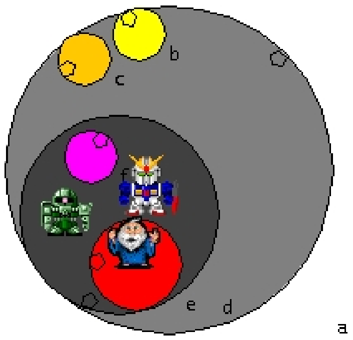
Pour faciliter l'étude du système, et permettre une meilleure interaction avec les icobjs *non-ambients*, j'ai rajouté des *pseudo-capacités* :

- $Slip\ t.P$: attente de t instants, puis déclenchement de P, c'est utile par exemple pour ralentir l'*explosion* d'un terme ($Rec\,X.b[\ X\ ]$, $Rec\,X.Slip\ 51.b[\ X\ ]$) ;

- $Emit\ 's'.P$: émission du signal s de manière locale (interne à l'ambient) ;
- $Wait\ 's'.P$: attente du signal s (local) ;

- $Emit\#\ 's'.P$: émission du message s de manière globale : tous les icobjs de la machine détectent ce message ;
- $Wait\#\ 's'.P$: attente du message global s (qui peut être émis par tout icobj...).

Les deux dernières permettent de créer deux nouveaux comportements : le *chargement* et le *déchargement* d'un icobj. L'icobj doté du comportement de chargement attend l'émission d'un message donné, si celui-ci a été émis par un ambient, il en devient le fils et se déplace donc à l'intérieur de cet ambient. Le comportement de déchargement lui permet alors de quitter cet ambient.



4 Conclusions

La programmation réactive, via les icobjs est bien adaptée au modèle des ambients, et en permet une visualisation très attractive qui pourra être utilisée lors de conférences, ou sur internet.

C'est un mode de programmation très agréable, car les comportements qu'il faut implémenter, et qui seront mis en parallèle par la suite, peuvent réellement être pensés indépendamment. Ainsi, le programmeur peut se concentrer sur de petites parties du code, sans avoir à se préoccuper de sa cohérence globale (exclusions mutuelles...).

Deux extensions sont envisageables :

- utiliser les possibilités de migration d'objets fournies par la programmation réactive, afin de distribuer la visualisation sur plusieurs machines.
- rajouter un algorithme de typage (pour les Controlled Ambients), qui permette d'afficher les capacités de chaque ambient mis en jeu.

Références

[BH96] Frederic Boussinot and Laurent Hazard. Reactive scripts. Technical Report RR-2868, 1996.

[Bou96] Frederic Boussinot. Icobj programming. Technical Report RR-3028, 1996.

[Bou00] Frédéric Boussinot. Junior automata. 2000.

[BS00] Frédéric Boussinot and Jean-Ferdy Susini. Junior rewrite semantics, 2000.

[Car99] Luca Cardelli. Mobility and security. *NATO Science Series*, 1999.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures : First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[CGG99] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. *Lecture Notes in Computer Science*, 1999.

[DCS00] Mariangiola Dezani-Ciancaglini and Ivano Salvo. Security types for mobile safe ambients. In *Asian Computing Science Conference*, 2000.

[FLS00] Cedric Fournet, Jean-Jacques Levy, and Alan Schmitt. An asynchronous, distributed implementation of mobile ambients. In *IFIP TCS*, 2000.

[HSB99] Laurent Hazard, Jean-Ferdy Susini, and Frédéric Boussinot. The junior reactive kernel. *RR-3732 INRIA*, 1999.

[Mil00] Robin Milner. Communicating and mobile systems : the pi-calculus. *Cambridge University Press*, 2000.

[PG02] Damien Pous and Thomas Gazagnaire. Une implémentation des controlled ambients en jocaml, 2002.

[TZH02] David Teller, Pascal Zimmer, and Daniel Hirschkoff. Using ambients to control resources. *RR-2002-16 ENS Lyon*, 2002.

[Zim] Pascal Zimmer. On the expressiveness of pure mobile ambients.

Annexe : syntaxe

Comment taper $\overline{in_\downarrow}$ au clavier ? Facile me direz-vous :

```
<\latexmode>$\overline{in_downarrow}$<\edomxetal>
```

Plus sérieusement, la syntaxe *réelle* est la suivante :

n désigne un nom d'ambient : alpha (alpha | ALPHA | num | _) *
N désigne un nom de variable : ALPHA (alpha | ALPHA | num | _) *
L'opérateur séquence (.) a priorité sur la mise en parallèle (|)

```
/* processus */
P    ::=
     | 0            // processus vide
     | n[P]         // ambient
     | M.P          // séquence
     | P | P        // mise en parallèle
     | rec X.P      // opérateur de récursion
     | X            // variable de récursion
     | nu L.P       // création d'un nom 'frais'
     | (P)          // parenthésage

/* liste de noms */
L    ::=
     | n
     | n, L

/* variables ("fonctions") */
X    ::=
     | N
     | N{L}

/* capacités */
M    ::=
     | in n         // opération standard IN
     | out n        // opération standard OUT
     | open n       // opération standard OPEN
     | CO??         // co-capacités dépendantes du modèle

     | slip t       // pseudo-capacités
     | wait STR
     | emit STR
     | wait# STR
     | emit# STR
```

```
/* chaîne de caractères */
STR      : :=
         | ¨ ( alpha | ALPHA | num | _ ) * ¨

/* Mobile Ambients */
COMA     : :=            // pas de co-capacités dans ce modèle

/* Safe Ambients */
COSA     : :=            // une co-capacité par capacité
         | in_ n
         | out_ n
         | open_ n

/* Robust Ambients */
CORA     : := COSA

/* Controlled Ambients */
COCA     : :=
         | in_d n        // co-capacités du IN
         | out_d n

         | out_u n       // co-capacités du OUT
         | in_u n

         | open_(m,n)    // co-capacité du OPEN
```