Aspects dynamiques dans les langages synchrones : le cas des SugarCubes

Version provisoire

Louis Mandel

Rapport de stage du D.E.A. de Programmation : Sémantique, Preuves et Langages

Encadrant : Marc Pouzet

Avril-Septembre 2002

Laboratoire d'Informatique de Paris 6

Table des matières

Introduction

Contexte

Ce document retrace le travail que j'ai effectué au cours de mon stage au LIP6 d'avril à septembre 2002. Ce stage a été effectué durant le D.E.A. de Programmation : Sémantique, Preuves et Langages. Il clôt également ma dernière année du Magistère d'Informatique Appliquée de l'Ile-de-France (MIAIF).

Ce stage s'est déroulé au LIP6 dans le pôle MFI (Méthodes Formelles et Implantations) composé des thèmes SPI (Sémantique Preuve et Implantation) et CalFor (Calcul Formel). Les principaux projets de ce pôle sont le projet FOC (Formel + Ocaml + Coq) : un atelier de calcul formel certifié, et le projet Lucid Synchrone : un langage synchrone d'ordre supérieur.

Le but de mon stage est de comprendre certains aspects dynamiques dans les langages synchrones en s'intéressant plus particulièrement au cas des SugarCubes.

Résumé

Au cours de mon stage je me suis intéressé aux langages synchrones et au aspects dynamiques dans ces langages. Je me suis concentré plus particulièrement sur l'approche réactive de Frédéric Boussinot. Cette approche qui dérive de l'approche synchrone repose sur un découpage logique de l'exécution d'un programme en succession d'instants. La notion d'instant permet de définir un parallélisme synchrone et un mécanisme de communication par diffusion instantanée d'événements. L'approche réactive permet de décrire des systèmes réactifs pouvant être enrichis dynamiquement par l'ajout de composants s'exécutant en parallèle.

Les SugarCubes sont une implantation de l'approche réactive qui se presente sous forme de librairie Java. Une sémantique formelle de Junior, un noyau des SugarCubes, a déjà été proposé par F. Boussinot et J.-F. Susini. Nous avons enrichi ce noyau pour qu'il prenne en compte la création dynamique. Nous lui avons donné une sémantique comportementale. En s'interessant à l'exécution efficace d'une réaction, nous avons proposé une sémantique "gloutonne" pour ce langage. Nous proposons une implantation nouvelle de Junior basée sur cette sémantique.

Chapitre 1

L'approche réactive de Boussinot

1.1 L'approche réactive

L'approche réactive [Bou96] dérive de *l'approche synchrone*. Elle se caractérise par :
- l'absence d'hypothèses sur la durée des instants ;
- la réaction instantanée à l'absence de signaux est impossible ;
- la possibilité d'exprimer des systèmes dynamiquement reconfigurables.

Frédéric Boussinot et son équipe de l'INRIA Sophia-Antipolis ont proposé plusieurs implantations de l'approche réactive. La principale est SUGARCUBES [BS98] qui est un langage embarqué dans Java. Ils ont également proposé un sous-ensemble de ce langage à la sémantique plus étudiée qui s'appelle JUNIOR [BHS99].

1.1.1 La notion d'instant

L'approche réactive garde la notion d'instant de *l'approche synchrone*. Une horloge découpe l'exécution d'un programme en instants logiques. Lors de l'activation d'un programme réactif, tous les composants parallèles constituant ce programme évoluent au même rythme. La notion d'instant permet d'exprimer le parallèle synchrone et un puissant mécanisme de communication entre les composants parallèles. La composition parallèle synchrone est définie ainsi, $P||Q$ avance d'un pas quand P et Q avancent d'un pas. Dans ces approches le temps est considéré comme un temps logique, ce ne sont pas des systèmes "temps-réels".

Contrairement à *l'approche synchrone*, dans *l'approche réactive* les instants ne sont pas de durée nulle. L'exécution des composants parallèles prend du temps. On peut donc parler d'évolution au cours d'un instant. On découpe un instant en une succession de micro-étapes qui vont converger vers un état stable. L'état stable est obtenu lorsque tous les composants ont été activés et qu'ils ont soit terminé leur exécution, soit qu'ils sont bloqués sur l'attente d'événements et que l'environnement ne peut plus être modifié. La durée d'une réaction peut varier d'un instant à l'autre. Donc aucune hypothèse ne peut être faite sur la durée d'un instant.

Il est important de ne pas confondre instants et micro-étapes. Les instants gardent les propriétés de *l'approche synchrone* (parallélisme, cohérence dans la vision des événements), alors que les micro-étapes ne les ont pas. En particulier, l'environnement peut évoluer au cours d'une micro-étape et donc des composants parallèles peuvent avoir des visions différentes de l'environnement.

1.1.2 Les communications

La communication dans *l'approche réactive* se fait par des événements diffusés instantanément à tous les composants. Ce système de communication a été introduit par le langage ESTEREL [BCG87]. Les événements peuvent venir de l'environnement externe (entrées) ou être générés par le programme réactif. Tous les composants ont la même vision des événements. Et les événements ne sont pas des données persistantes. C'est-à-dire qu'un événement n'est présent que durant l'instant où il est généré. Cela a pour avantage d'éviter des problèmes de stockage de l'historique des événements.

Pour communiquer, deux composants n'ont pas besoin de se connaître. Grâce aux propriétés de diffusion des événements il n'y a pas de problèmes de déclaration d'émetteur et de récepteur.

La notion d'instant permet de définir clairement les notions de présence ou absence d'un événement et de simultanéité. En effet, deux événements serons simultanés s'ils sont générés au cours du même instant. Et un événement est dit présent s'il est émis durant l'instant, sinon il est absent.

Un des principes de base de *l'approche réactive* est qu'il n'est pas possible de réagir instantanément à l'absence d'un signal. Cette impossibilité est due à la décomposition en micro-étapes des instants. En effet, si on teste l'absence d'un événement dans une micro-étape, on ne sait pas encore s'il va être généré plus tard dans l'instant. C'est uniquement une fois l'instant terminé que l'on est sûr que l'événement ne sera plus généré, mais il est alors trop tard pour réagir dans l'instant. C'est pour cela que la réaction à l'absence d'un événement est toujours retardée d'un instant.

Ce principe a été étudié dans le langage SL [BdS96]. Le langage SL est un sous-ensemble d'ESTEREL qui respecte ce principe. Dans le cadre de l'étude de ce langage il a été montré que cette contrainte forte permet d'éviter les problèmes de causalité d'ESTEREL. Par exemple le programme :

```
present S then
  nothing
else
  emit S
end
```

pause des problèmes si la réaction instantanée à l'absence est possible. Car si on suppose que `S` est absent, alors on exécute la branche `else` qui émet `S` dans le même instant. Ceci contre dit l'hypothèse d'absence du signal. Et si on suppose que `S` est présent on doit vérifier que `S` est émis lors de l'activation du programme. Or avec cette hypothèse on passe par la branche `then` de l'instruction `present` qui ne fait rien donc n'émet pas `S`. Donc les deux hypothèses `S` est présent et `S` est absent sont fausses donc le programme n'est pas causal.

Mais avec une réaction à l'absence retardée d'un instant il n'y a plus de paradoxe. En effet, si `S` est absent pendant l'instant du test, comme l'émission ne se produit qu'à l'instant d'après, alors l'hypothèse n'est pas contredite. Si on veut avoir un comportement équivalent en ESTEREL, il faut écrire le programme suivant :

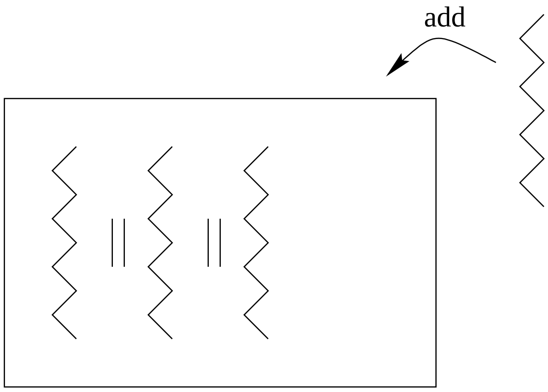
```
present S then
  nothing
else
  pause;
  emit S
end
```

1.1.3 L'aspect dynamique

La suppression des contraintes sur la durée des instants facilite la possibilité d'ajout et de suppression de composants parallèles au cours de l'exécution. Les modifications du programme à exécuter se font au moment où le programme est dans un état stable, c'est-à-dire entre deux instants.

Nous avons vu dans le paragraphe précédant que les restrictions faites dans *l'approche réactive* empêche l'écriture de programmes non causaux. Donc l'ajout de composants parallèles ne peut pas poser de problèmes de causalité.

Entre deux instants on peut donc ajouter ou supprimer des composants dans un programme. Les composants supprimés peuvent être récupérés par l'environnement Java et réintroduits ultérieurement. Les ajouts et suppressions de composants parallèles permettent la reconfiguration d'un système réactif entre deux instants.



Ajout dynamique de composant

1.2 Les SugarCubes

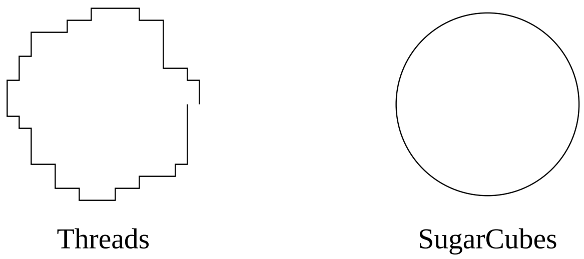
Les SUGARCUBES [BS98] sont le prolongement de travaux plus anciens sur sur *l'approche réactive*. Une des premières implantations de cette approche était le langage REACTIVE-C [Bou91]. Ce langage qui ajoute des instructions réactives au langage C a servi d'assembleur pour d'autres formalismes réactifs.

Les SUGARCUBES ont pour objectifs de fournir un langage de programmation où l'expression du parallélisme et des communications est simple et puissante. Ils veulent aussi permettre la définition de systèmes dynamiquement reconfigurables. Ils ont également pour but d'avoir un caractère modulaire et cohérent avec un modèle de programmation objets.

Les SugarCubes sont une implantation de *l'approche réactive*. SugarCubes est un langage interprété qui est embarqué dans Java. Il y a donc des interactions entre ces deux langages. Les SugarCubes permettent l'utilisation d'événements diffusés et également de configurations événementielles. Ces configurations sont des expressions booléennes qui sont construites à partir de la présence ou de l'absence de certains signaux.

Il existe déjà en Java la possibilité d'écrire des programmes parallèles avec l'utilisation de *threads*. Mais les SugarCubes ont une approche différente. Ils facilitent l'écriture des programmes fortement communicant avec le mécanisme de diffusion instantanée d'événement. Et ils ont du parallélisme synchrone.

Un exemple classique de programme que l'on peut écrire en SugarCubes mais qui n'a pas le comportement désiré avec des *threads* est l'exemple du cercle. Le principe de cet exemple est de déplacer un point de façon circulaire par la composition parallèle d'un mouvement cosinus et d'un mouvement sinus. Avec des *threads* l'ordonnancement n'étant pas déterministe les deux composants n'évolent pas nécessairement à la même vitesse et le programme n'affiche pas un cercle bien formé. En revanche les SugarCubes utilise la composition parallèle synchrone donc quand *cos* avance d'un pas, *sin* avance aussi d'un pas.



Nous allons voir maintenant les principales instructions de ce langage.

1.2.1 Le langage

Une première approche pour un langage peut être d'écrire des programmes. Voici un programme qui va afficher "Hello" puis "Bye".

```
Program p = Merge(
  Seq(
      Seq(Atom(System.out.print("Hello")),
          Stop()),
      Generate("e")),
  Seq(Await("e"), Atom(System.out.print("Bye"))));
```

Ce programme a deux composants qui s'exécutent en parallèle.
- Le premier composant affiche la chaîne de caractères "Hello", attend un instant puis génère l'événement e.
- Le second composant attend que l'événement e soit généré puis affiche par une action Java la chaîne de caractères "Bye".

L'activation de `p` produit la succession de micro-étapes suivante. Le premier composant est activé, donc l'action Java `System.out.print("Hello")` est exécutée, puis la rencontre de l'instruction `Stop` arrête l'activation de la première branche du parallèle. C'est alors au tour de la seconde branche d'être activée. L'instruction `Await("e")` constate que `e` n'est pas dans l'environnement, et donc stoppe son exécution. Il n'y a alors plus rien à exécuter, on peut donc passer à l'instant suivant.

On recommence par activer la branche gauche de l'instruction `Merge`. Cette activation génère le signal `e` et termine complètement l'exécution du premier composant. L'événement `e` étant généré, l'activation du second composant peut passer l'instruction `Await` et afficher le message `"Bye"`.

Finalement l'exécution du programme réactif `p` donne la sortie suivante :

```
instant1 : Hello
instant2 : Bye
```

Le programme `p` est équivalent à :

```
Atom(System.out.print("Hello"));
Stop();
Generate("e");
Await("e");
Atom(System.out.print("Bye"));
```

Nous donnons maintenant un exemple de programme avec création dynamique.

```
Program p =
  Merge(
    Shell(a,anim),
    loop(Seq(Seq(await(lbp),
                 add(a,nouv_anim)),
             Stop())));
```

La branche gauche de l'instruction `Merge` est le programme `Shell(a,anim)`. L'instruction `Shell` nomme le sous-programme `anim`. Nous supposons ici que `anim` est un programme SugarCubes qui affiche une animation à l'écran.

La branche droite de l'instruction `Merge` est une boucle infinie. Le corps de cette boucle attend l'événement `lbp` et quand il est généré ajoute le programme `nouv_anim` en parallèle au programme `anim` dans le `Shell` de nom `a`. L'instruction `Stop` à la fin de la boucle est là pour éviter d'avoir une boucle instantanèe.

Si l'événement `lbp` est généré lors que l'on click sur le bouton de la souris, alors le comportement du programme `p` est d'afficher l'animation `anim`, et à chaque click ajouter une nouvelle animation.

Après avoir présenté intuitivement le langage nous donnons sa syntaxe. Nous nous sommes restreint à Junior, un noyau des SugarCubes auquel on a ajouté les instructions `shell`, `add` et `machine`.

$$
\begin{array}{rcl}
prog & ::= & \texttt{nothing} \\
 & | & \texttt{stop} \\
 & | & \texttt{atom}(j) \\
 & | & \texttt{generate}(evt) \\
 & | & \texttt{await}(config) \\
 & | & \texttt{add}(id, prog) \\
 & | & prog || prog \\
 & | & prog; prog \\
 & | & \texttt{loop}(prog) \\
 & | & \texttt{repeat}(n, prog) \\
 & | & \texttt{until}(config, prog, prog) \\
 & | & \texttt{when}(config, prog, prog) \\
 & | & \texttt{control}(evt, prog) \\
 & | & \texttt{local}(evt, prog) \\
 & | & \texttt{link}(obj, prog) \\
 & | & \texttt{shell}(id, prog) \\
 & | & M \\
 & & \\
M & ::= & \texttt{machine}(id, obj, prog)
\end{array}
$$

Ici evt correspond à un événement. j représente du code Java à exécuter et obj est un objet Java. n est un entier et id un identificateur.

Nous regroupons ces instructions en quatre familles :

- les instructions atomiques ;
- les instructions de compositions d'instructions ;
- les instructions de contrôle d'exécution ;
- et les instructions de structuration du code.

Nous donnons maintenant une description des instructions en commençant par les instructions atomiques.

Nothing : qui ne fait rien et termine instantanément.

Stop : est une instruction qui attend un instant puis termine.

Atom(j) : est une instruction qui a en paramètre une action Java. Du point de vue réactif `atom` se comporte comme `nothing` mais en plus `atom` va exécuter l'action Java qu'elle prend en argument et donc produire un effet de bord.

Generate(s,v) : génère un événement qui peut être valué.

Await(C) : attend qu'une configuration événementielle soit satisfaite.

Add(a,P) : est l'instruction qui permet de faire des demandes d'ajout dynamique de code en parallèle au sous-programme nommé a.

Pour composer les instructions on dispose de :

Merge : qui est la composition parallèle.

Seq : est la composition séquentielle. Elle active sa première branche, et si celle-ci termine dans le même instant alors Seq active sa seconde branche.

Repeat et Loop : sont les boucles respectivement infinies et finies.

On peut contrôler l'exécution d'un programme avec les instructions suivantes :

Until(C,P,Q) **:** qui permet de faire de la préemption faible sur le programme P lorsque la configuration C est satisfaite. Et exécuter la continuation Q après la préemption.

When(C,P,Q) **:** est une instruction qui permet de choisir entre deux programmes P et Q lequel exécuter suivant la valeur de configuration événementielle C.

Control(s,P) **:** exécute son corps que lorsque l'événement s est présent.

Enfin nous allons voir les instructions qui permettent de donner des propriétés à un sous-programme d'un programme :

Local(s,P) **:** déclare localement l'événement s au sous-programme P.

Link(obj,P) associe un objet Java à un programme réactif.

Shell(a,P) nomme un sous-programme pour pouvoir permettre l'ajout dynamique en parallèle de ce sous-programme.

Les machines réactives

En plus des instructions précédentes, il y a l'instruction `machine`. Cette instruction particulière englobe un programme et lui fournit un environnement pour s'exécuter. Depuis l'environnement Java, on peut appeler la méthode `react` d'une machine pour exécuter un instant. De plus comme `machine` est une instruction, on peut l'utiliser à l'intérieur d'un programme réactif, dans ce cas la `machine` forme une coque imperméable autour de son programme.

Ce sont les machines qui implantent l'algorithme de détection de fin d'instant. Et comme pour les `shell` on peut ajouter dynamiquement des programmes dans une machine.

Pour mieux comprendre les SugarCubes nous commençons par donner une sémantique comportementale de ce langage.

1.3 Sémantique comportementale

Nous ne présentons pas une sémantique complète des SugarCubes mais une sémantique de Junior auquel on a ajouté les instructions `add`,`shell` et `machine`. Plusieurs sémantiques ont déjà été données à Junior. On peut les trouver dans [Sus01] et nous en présenterons une dans la section 1.4.

La sémantique comportementale est une sémantique à grand pas. Elle est grand pas dans le sens où une règle représente l'exécution d'un instant. Sa présentation s'inspire de la sémantique comportementale d'Esterel qui a été donnée dans [BG92]. Avec cette sémantique, nous avons souhaité avoir une formulation plus simple de la sémantique de Junior. Elle décrit la réaction du programme, mais ne définit pas comment faire une implantation exécutable. Nous verrons au moment de la définition de la composition parallèle les problèmes de réalisation de cette sémantique.

Les transitions sont de la forme :

$$E,\ A \vdash P \xrightarrow{E',\ \alpha,\ A'} P'$$

où
- E' est l'ensemble des signaux émis par P ;
- $\alpha \in \{STOP,\ TERM\}$ est le statut rendu par l'activation de P ;
- A' est l'ensemble des requêtes d'ajout faites par P ;
- E est l'environnement des signaux dans lequel P doit être évalué ;
- A est l'ensemble des requêtes d'ajout avant l'activation de P ;

Ce qui signifie que dans l'environnement (E, A) le programme P se réécrit en P' en produisant les signaux E' et les requêtes d'ajout A'. Si le statut α retourné par la réaction est $STOP$, cela veut dire qu'il faudra exécuter le programme P' à l'instant suivant. Si le statut est $TERM$, le programme a complètement fini son exécution.

Une réaction $E,\ A \vdash P \xrightarrow{E',\ \alpha,\ A'} P'$ correspond à l'exécution d'un instant. L'exécution d'un programme est une suite de réactions qui renvoie le statut $STOP$ et qui se finit par une réaction qui renvoie le statut $TERM$.

Dans cette sémantique nous ne précisons pas l'environnement Java. Ainsi pour les instructions `atom` et `link` nous ne voyons que leur comportement réactif.

1.3.1 Définitions de l'environnement et des configurations événementielles

L'ensemble des événements

E est un ensemble d'événements défini ainsi :

$$\begin{array}{lcl} E & ::= & \emptyset \\ & | & \{\ evt\{, evt\}^*\ \} \end{array}$$

evt est un événement que l'on note en général s ou s_i.

L'ajout d'un signal s dans l'ensemble se note $E[s]$ et la suppression se note $E\backslash s$. L'union de deux ensembles d'événements est l'union classique sur les ensembles.

L'ensemble des requêtes d'ajout

Les requêtes d'ajout A sont représentées par un ensemble de paires (a, P) où a est un nom et P un programme.

$$\begin{array}{lcl} A & ::= & \emptyset \\ & | & \{\ (id, prog)\{, (id, prog)\}^*\ \} \end{array}$$

id est un identificateur qui est utilisé pour nommer les sous-programmes. On prend en général les lettres a, b ou c pour les représenter.

L'obtention du programme associé à un identificateur a (noté $A(a)$) est défini par :

$$\begin{array}{lcll} A(a) & = & P & \text{si } (a, P) \in A \\ & = & \texttt{nothing} & \text{sinon} \end{array}$$

L'ajout de l'élément (a, P_1) dans A qui se note $A[a\backslash P_1]$, est défini par :

$$\begin{array}{lcll} A[a\backslash P_1](a) & = & P_1 & \text{si } a \notin Dom(A) \\ & = & P_1 || A(a) & \text{sinon} \end{array}$$

et

$$A[a \backslash P_1](b) \quad = \quad A(b) \quad \text{si } a \neq b$$

L'union sur les ensembles de requêtes d'ajout (notée $A_1|A_2$), consiste à mettre en parallèle les programmes de A_1 et A_2 lorsqu'ils sont associés au même nom :

$$(A_1|A_2)(a) = A_1(a)||A_2(a)$$

Par exemple, l'union de $\{(a, P_1); (b, P_2)\}$ et $\{(a, P_3), (c, P_4)\}$ est :

$$\{(a, P_1); (b, P_2)\}|\{(a, P_3), (c, P_4)\} = \{(a, P_1||P_3); (b, P_2)(c, P_4)\}.$$

Les configurations événementielles

Dans les programmes nous utilisons des configurations événementielles qui sont définies ainsi :

$$\begin{array}{rcl} config & ::= & present(evt) \\ & | & not\ config \\ & | & config\ and\ config \\ & | & config\ or\ config \end{array}$$

On dispose de deux fonctions *eoi* (*End Of Instant*) et *eval* qui permettent respectivement de savoir si une configuration est fixée pendant l'instant ou après la fin d'instant et de connaître sa valeur.

La fonction *eoi* retourne la valeur vrai si on ne peut connaître la valeur de la configuration qu'à la fin d'un instant.

$$\begin{array}{rcl} eoi(present(s), E) & = & (s \notin E) \\ eoi(not\ C, E) & = & eoi(C, E) \\ eoi(C_1\ and\ C_2, E) & = & \left\{ \begin{array}{ll} & (eoi(C_1, E) \wedge eoi(C_2, E)) \\ \vee & (\neg eoi(C_1, E) \wedge eval(C_1, E) \wedge eoi(C_2, E)) \\ \vee & (eoi(C_1, E) \wedge \neg eoi(C_2, E) \wedge eval(C_2, E)) \end{array} \right. \\ eoi(C_1\ or\ C_2, E) & = & \left\{ \begin{array}{ll} & (eoi(C_1, E) \wedge eoi(C_2, E)) \\ \vee & (\neg eoi(C_1, E) \wedge \neg eval(C_1, E) \wedge eoi(C_2, E)) \\ \vee & (eoi(C_1, E) \wedge \neg eoi(C_2, E) \wedge \neg eval(C_2, E)) \end{array} \right. \end{array}$$

$eoi(present(s), E)$ rend la valeur vrai quand s est absent car dans le cadre de l'approche réactive la réaction à l'absence est retardée (voir 1.1.2). Lors du test d'une configuration *and* ou *or*, la configuration peut être fixée dès qu'une branche retourne la valeur $false$ dans le cas *and* ou $true$ dans le cas *or*. Il est donc nécessaire de traiter ces cas dans la définition de *eoi*.

On définit maintenant la fonction *eval* :

$$\begin{array}{rcl} eval(present(s), E) & = & s \in E \\ eval(not\ C, E) & = & \neg eval(C, E) \\ eval(C_1\ and\ C_2, E) & = & eval(C_1) \wedge eval(C_2) \\ eval(C_1\ or\ C_2, E) & = & eval(C_1) \vee eval(C_2) \end{array}$$

On définit la fonction qui teste si une configuration est satisfaite :

$$sat(C, E) = eval(C, E)$$

et la fonction qui teste si une configuration est insatisfaite :

$$unsat(C, E) = \neg eval(C, E)$$

1.3.2 Les instructions

Nothing

L'instruction `nothing` ne fait rien et termine dans l'instant (ce qui est indiqué dans la règle par le statut *TERM*).

$$E,\ A \vdash \texttt{nothing} \xrightarrow{\emptyset,\ TERM,\ \emptyset} \texttt{nothing}$$

Stop

L'instruction `stop` ne fait rien mais termine à l'instant suivant. c'est pour cela que l'instruction se réécrit en `nothing` et renvoie le statut *STOP*.

$$E,\ A \vdash \texttt{stop} \xrightarrow{\emptyset,\ STOP,\ \emptyset} \texttt{nothing}$$

Atom

`atom` est une action atomique qui modifie l'environnement Java et termine immédiatement.

$$E,\ A \vdash \texttt{atom}(j) \xrightarrow{\emptyset,\ TERM,\ \emptyset} \texttt{nothing}$$

Generate

La génération d'événement se fait par l'instruction `generate`.

$$E,\ A \vdash \texttt{generate}(s) \xrightarrow{\{s\},\ TERM,\ \emptyset} \texttt{nothing}$$

Await

L'instruction `await`(C) est bloquante tant que la configuration C n'est pas satisfaite. Comme nous donnons une sémantique comportementale, nous sommes sûrs d'avoir toutes les informations dans E pour savoir quelle règle il faut appliquer.

Dans le cas où la configuration C est satisfaite et que la fin d'instant n'a pas encore été décidée alors l'instruction termine.

$$\frac{sat(C, E) \quad \neg eoi(C, E)}{E,\ A \vdash \texttt{await}(C) \xrightarrow{\emptyset,\ TERM,\ \emptyset} \texttt{nothing}}$$

Maintenant, si la configuration C est satisfaite après la fin d'instant alors `await` se comporte comme `stop`.

$$\frac{sat(C,E) \quad eoi(C,E)}{E,\ A \vdash \texttt{await}(C) \xrightarrow{\emptyset,\ STOP,\ \emptyset} \texttt{nothing}}$$

Si la configuration C n'est pas satisfaite, l'instruction se réécrit en elle même et devra être réexécutée à l'instant suivant.

$$\frac{unsat(C,E)}{E,\ A \vdash \texttt{await}(C) \xrightarrow{\emptyset,\ STOP,\ \emptyset} \texttt{await}(C)}$$

When

L'instruction `when` est une instruction non bloquante qui permet d'exécuter un programme ou un autre en fonction de la valeur d'une configuration. Cette instruction se comporte comme un `if` sur une configuration événementielle. Pour des problèmes de causalité, il se peut que la configuration soit fixée uniquement lorsque l'instant est terminé. Dans ce cas la réaction du programme ne peut se faire qu'à l'instant suivant.

$$\frac{sat(C,E) \quad \neg eoi(C,E) \quad E,\ A \vdash P \xrightarrow{E',\ \alpha,\ A'} P'}{E,\ A \vdash \texttt{when}(C,P,Q) \xrightarrow{E',\ \alpha,\ A'} P'}$$

$$\frac{sat(C,E) \quad eoi(C,E)}{E,\ A \vdash \texttt{when}(C,P,Q) \xrightarrow{\emptyset,\ STOP,\ \emptyset} P}$$

$$\frac{unsat(C,E) \quad \neg eoi(C,E) \quad E,\ A \vdash P \xrightarrow{E',\ \alpha,\ A'} Q'}{E,\ A \vdash \texttt{when}(C,P,Q) \xrightarrow{E',\ \alpha,\ A'} Q'}$$

$$\frac{unsat(C,E) \quad eoi(C,E)}{E,\ A \vdash \texttt{when}(C,P,Q) \xrightarrow{\emptyset,\ STOP,\ \emptyset} Q}$$

Control

Cette instruction active son corps uniquement si l'événement s est présent.

$$\frac{s \in E \quad E,\ A \vdash P \xrightarrow{E',\ \alpha,\ A'} P'}{E,\ A \vdash \texttt{control}(s,P) \xrightarrow{E',\ \alpha,\ A'} \texttt{control}(s,P')}$$

$$\frac{s \notin E}{E,\ A \vdash \texttt{control}(s,P) \xrightarrow{\emptyset,\ STOP,\ \emptyset} \texttt{control}(s,P)}$$

Add

L'instruction `add` permet de faire une demande d'ajout d'un programme en parallèle dans un shell. L'exécution de cette instruction enregistre la requête pour que l'ajout effectif puisse se faire à la fin de l'instant.

$$E,\ A \vdash \texttt{add}(a, P) \xrightarrow{\emptyset,\ TERM,\ \{(a,P)\}} \texttt{nothing}$$

Shell

L'instruction `shell` nomme un sous-programme. Il faut que le nom du shell soit unique dans une machine. L'activation de cette instruction lance l'activation de son corps et si cette activation retourne le statut $STOP$, alors l'instruction `shell` ajoute en parallèle à son corps les programmes qui dans A sont associés au nom a.

$$\frac{E,\ A \vdash P \xrightarrow{E',\ STOP,\ A'} P' \quad P'' = A(a)||A'(a)}{E,\ A \vdash \texttt{shell}(a, P) \xrightarrow{E',\ STOP,\ A'} \texttt{shell}(a, P'||P'')}$$

Dans le cas où le corps de l'instruction `shell` rend le statut $TERM$, les requêtes d'ajout ne sont pas exécutées car le shell a déjà complètement fini son exécution.

$$\frac{E,\ A \vdash P \xrightarrow{E',\ TERM,\ A'} P'}{E,\ A \vdash \texttt{shell}(a, P) \xrightarrow{E',\ TERM,\ A'} \texttt{shell}(a, P')}$$

Link

L'instruction `link` permet d'associer un objet Java à un programme réactif. Ceci permet aux instructions `atom` de P d'appeler des méthodes de cet objet. Mais ici, d'un point de vue réactif, l'instruction `link` ne fait qu'activer son corps.

$$\frac{E,\ A \vdash P \xrightarrow{E',\ \alpha,\ A'} P'}{E,\ A \vdash \texttt{link}(obj, P) \xrightarrow{E',\ \alpha,\ A'} \texttt{link}(obj, P')}$$

Seq

La mise en séquence d'instructions se décompose en deux cas. Dans le premier, l'activation de la branche gauche de la séquence retourne le statut $STOP$. Alors, l'exécution de la séquence ne peut pas continuer dans l'instant, et donc retourne le statut $STOP$.

$$\frac{E,\ A \vdash P_1 \xrightarrow{E_1,\ STOP,\ A_1} P_1'}{E,\ A \vdash P_1; P_2 \xrightarrow{E_1,\ STOP,\ A_1} P_1'; P_2}$$

Dans le second cas, l'activation de la branche gauche termine, il faut donc lancer l'activation de la branche droite.

$$\frac{E,\ A \vdash P_1 \xrightarrow{E_1,\ TERM,\ A_1} P_1' \quad E \cup E_1,\ A|A_1 \vdash P_2 \xrightarrow{E_2,\ \alpha,\ A_2} P_2'}{E,\ A \vdash P_1; P_2 \xrightarrow{E_1 \cup E_2,\ \alpha,\ A_1|A_2} P_2'}$$

Si on active la séquence dans l'environnement E, A alors on active P_1 dans le même environnement. Lors de cette activation P_1 génère de nouveaux événements et requêtes d'ajout. Ces nouveaux éléments doivent être ajoutés à l'environnement dans lequel P_2 est activé. Le statut final est celui de P_2, mais les événements générés et requêtes d'ajout effectuées par la séquence sont l'union de ceux de P_1 et P_2.

Loop

Une fois l'opérateur de mise en séquence défini, il est simple d'exprimer la boucle infinie.

$$\frac{E,\ A \vdash P; \texttt{loop}(P) \xrightarrow{E',\ \alpha,\ A'} Q}{E,\ A \vdash \texttt{loop}(P) \xrightarrow{E',\ \alpha,\ A'} Q}$$

Cette règle peut éventuellement ne jamais terminer si P ne renvoie pas le statut *STOP*. En SugarCubes, il y a un contrôle à l'exécution pour éviter ce cas.

Repeat

`repeat` est une boucle fini qui active n fois son corps, où n lui est donné en paramètre. Comme pour `loop`, on définit `repeat` de façon récursive.

$$\frac{n > 0 \quad E,\ A \vdash P; \texttt{repeat}(n-1, P) \xrightarrow{E',\ \alpha,\ A'} Q}{E,\ A \vdash \texttt{repeat}(n, P) \xrightarrow{E',\ \alpha,\ A'} Q}$$

$$\frac{n \leq 0}{E,\ A \vdash \texttt{repeat}(n, P) \xrightarrow{\emptyset,\ TERM,\ \emptyset} \texttt{nothing}}$$

Par

Pour activer $P_1||P_2$, on doit activer P_1 dans un environnement qui dépend de l'activation de P_2 et l'activation de P_2 doit se faire dans un environnement qui dépend de l'activation de P_1.

$$\frac{E \cup E_2,\ A|A_2 \vdash P_1 \xrightarrow{E_1,\ \alpha,\ A_1} P_1' \quad E \cup E_1,\ A|A_1 \vdash P_2 \xrightarrow{E_2,\ \beta,\ A_2} P_2'}{E,\ A \vdash P_1||P_2 \xrightarrow{E_1 \cup E_2,\ \gamma(\alpha,\beta),\ A_1|A_2} P_1'||P_2'}$$

$$\gamma(\alpha, \beta) = \begin{cases} TERM & \text{si } \alpha = TERM \text{ et } \beta = TERM \\ STOP & \text{sinon} \end{cases}$$

Le programme :

$$\texttt{when}(present(A), \texttt{generate}(B))||\texttt{when}(present(B), \texttt{generate}(A))$$

a deux sens dans la sémantique. Dans la première on suppose que A est présent dans E_2.

$$\frac{\dfrac{s_1 \in \{s_1\} \quad \{s_1\}, \emptyset \vdash \texttt{generate}(s_2) \xrightarrow{\{s_2\},\ TERM,\ \emptyset} \texttt{nothing}}{\{s_1\}, \emptyset \vdash \texttt{when}(s_1, \texttt{generate}(s_2)) \xrightarrow{\{s_2\},\ TERM,\ \emptyset} \texttt{nothing}} \quad \dfrac{s_2 \in \{s_2\} \quad \{s_2\}, \emptyset \vdash \texttt{generate}(s_1) \xrightarrow{\{s_1\},\ TERM,\ \emptyset} \texttt{nothing}}{\{s_2\}, \emptyset \vdash \texttt{when}(s_2, \texttt{generate}(s_1)) \xrightarrow{\{s_1\},\ TERM,\ \emptyset} \texttt{nothing}}}{\emptyset, \emptyset \vdash \texttt{when}(s_1, \texttt{generate}(s_2))||\texttt{when}(s_2, \texttt{generate}(s_1)) \xrightarrow{\{s_1, s_2\},\ TERM,\ \emptyset} \texttt{nothing}}$$

Dans la seconde sémantique, on suppose que s_1 n'est pas dans E_2.

Cette règle de mise en parallèle des instructions rend cette sémantique non exécutable.

$$\frac{\dfrac{s_1 \notin \emptyset}{\emptyset, \emptyset \vdash \texttt{when}(s_1, \texttt{generate}(s_2)) \xrightarrow{\emptyset,\ STOP,\ \emptyset} \texttt{nothing}} \quad \dfrac{s_2 \notin \emptyset}{\emptyset, \emptyset \vdash \texttt{when}(s_2, \texttt{generate}(s_1)) \xrightarrow{\emptyset,\ STOP,\ \emptyset} \texttt{nothing}}}{\emptyset, \emptyset \vdash (\texttt{when}(s_1, \texttt{generate}(s_2)))||(\texttt{when}(s_2, \texttt{generate}(s_1))) \xrightarrow{\emptyset,\ STOP,\ \emptyset} \texttt{nothing}}$$

Dans les SugarCubes c'est la seconde sémantique qui est donnée.

Until

L'opérateur `until` est un opérateur de préemption faible, c'est-à-dire que si la configuration C est satisfaite, le programme P n'est pas stoppé immédiatement ; il est arrêté lorsqu'il a fini de réagir pour l'instant courant.

Si le corps (P) termine, que la configuration soit satisfaite ou pas, l'instruction termine.

$$\frac{E,\ A \vdash P \xrightarrow{E',\ TERM,\ A'} P'}{E,\ A \vdash \texttt{until}(C, P, Q) \xrightarrow{E',\ TERM,\ A'} \texttt{until}(C, P', Q)}$$

Si le corps rend le statut $STOP$, alors si C est satisfaite avant la fin de l'instant, on lance l'exécution de Q et P ne reprendra pas son exécution.

$$\frac{E,\ A \vdash P \xrightarrow{E',\ STOP,\ A'} P' \quad sat(C, E \cup E') \quad \neg eoi(C, E \cup E') \quad E \cup E',\ A|A' \vdash Q \xrightarrow{E'',\ \alpha,\ A''} Q'}{E,\ A \vdash \texttt{until}(C, P, Q) \xrightarrow{E' \cup E'',\ \alpha,\ A'|A''} Q'}$$

Si le corps rend le statut $STOP$ et C est satisfaite après la fin de l'instant, alors à l'instant suivant il faudra exécuter Q.

$$\frac{E,\ A \vdash P \xrightarrow{E',\ STOP,\ A'} P' \quad sat(C, E \cup E') \quad eoi(C, E \cup E')}{E,\ A \vdash \texttt{until}(C, P, Q) \xrightarrow{E',\ STOP,\ A'} Q}$$

Si P retourne $STOP$ et que C n'a pas été satisfaite, alors P reprend son exécution à l'instant suivant toujours sous le contrôle du `until`.

$$\frac{E,\ A \vdash P \xrightarrow{E',\ STOP,\ A'} P' \quad unsat(C, E \cup E')}{E,\ A \vdash \texttt{until}(C, P, Q) \xrightarrow{E',\ STOP,\ A'} \texttt{until}(C, P', Q)}$$

Local

$\texttt{local}(s, P)$ permet de déclarer que le signal s est local à P. Ce qui signifie que si s est généré dans P alors le reste du programme ne reçoit pas s et si est généré à l'extérieur de P alors s n'est pas propagé dans P.

$$\frac{E\backslash s,\ A \vdash P \xrightarrow{E',\ \alpha,\ A'} P'}{E,\ A \vdash \texttt{local}(s, P) \xrightarrow{E'\backslash s,\ \alpha,\ A'} \texttt{local}(s, P')}$$

La notation $E\backslash s$ désigne l'ensemble E auquel on a retiré les occurrences de s.

Machine

La construction `machine` encapsule un programme, quelle nomme et lui associe un objet Java. On peut également préempter une machine avec l'événement $Kill_m$ où m est le nom de la machine.

La réaction d'une machine fait réagir son corps dans un nouvel environnement. Les signaux et les requêtes d'ajout restent locaux à la machine.

$$\frac{\emptyset, \emptyset \vdash \texttt{until}(Kill_m, \texttt{link}(obj, \texttt{shell}(m, P)), \texttt{nothing}) \xrightarrow{E',\ TERM,\ A'} Q}{E,\ A \vdash \texttt{machine}(m, obj, P) \xrightarrow{\emptyset,\ TERM,\ \emptyset} \texttt{machine}(m, obj, \texttt{nothing})}$$

Si la machine retourne le statut $STOP$, il faut ajouter à son corps les programmes qui sont dans A.

$$\frac{\begin{array}{c}\vdash \texttt{until}(Kill_m, \texttt{link}(obj, \texttt{shell}(m, P)), \texttt{nothing}) \\ \xrightarrow{E',\ STOP,\ A'} \\ \texttt{until}(Kill_m, \texttt{link}(obj, \texttt{shell}(m, P')), \texttt{nothing})\end{array} \qquad Kill_m \notin E \cup E'}{E,\ A \vdash \texttt{machine}(m, obj, P) \xrightarrow{\emptyset,\ STOP,\ \emptyset} \texttt{machine}(m, obj, P' || A(m))}$$

Une `machine` peut être tuée par un événement $Kill_m$ qui se trouve dans E. C'est-à-dire que l'on peut détruire une machine depuis l'extérieur.

$$\frac{\vdash \texttt{until}(Kill_m, \texttt{link}(obj, \texttt{shell}(m, P)), \texttt{nothing}) \xrightarrow{E',\ STOP,\ A'} Q \quad Kill_m \in E}{E,\ A \vdash \texttt{machine}(m, obj, P) \xrightarrow{\emptyset,\ TERM,\ \emptyset} \texttt{nothing}}$$

1.4 La sémantique Rewrite

La sémantique comportementale que nous venons de présenter n'est pas exécutable et donc mène difficilement à une implantation. La sémantique que nous présentons maintenant est exécutable et a été implanté. Elle a été réalisée par F. Boussiot et J.-F. Susini [BS00].

Plusieurs sémantiques de JUNIOR ont été proposées dans [Sus01]. Les trois principales se nomment *Rewrite*, *Replace* et *Storm*.

- *Rewrite* est une sémantique par réécriture. Elle calcule la réaction d'un instant par itérations successives de son système de règle. L'exécution d'un instant est terminée lors de l'obtention d'un point fixe.
- *Replace* se base sur le même principe que *Rewrite*. *Replace* ne travaille pas par réécriture mais par déplacement d'un point de contrôle.
- *Storm* est une extension de *Replace* auquel est ajouté un système de remontée d'information qui permet de minimiser les activations inutiles.

Nous présentons ici la sémantique *Rewrite* à laquelle nous avons ajouté les instructions `add`, `shell` et `machine` qui n'étaient pas présentes.

Dans cette sémantique les règles décrivent une micro-réaction. L'activation du programme P dans les environnements E_m^{eoi}, A produit un nouveau programme P' avec les environnements $E_m^{eoi'}$ et A'. Le résultat de la réaction étant un nouveau micro-état.

$$P, E_m^{eoi}, A \xrightarrow{\alpha} P', E_m^{eoi'}, A'$$

P est un programme réactif. E_m^{eoi} l'environnement des signaux auquel nous avons ajouté deux annotations eoi et m (pour $move$) que nous allons définir plus tard. A est l'ensemble des requêtes d'ajout et α est le statut retourné par l'activation.

Dans cette sémantique α peut prendre les valeurs $STOP$ et $TERM$ ainsi que la nouvelle valeur $SUSP$. $SUSP$ indiquant que le programme doit être réactivé avant la fin d'instant.

Les environnements

Dans cette sémantique E représente toujours l'ensemble des signaux émis. On lui ajoute deux valeurs booléennes eoi et $move$ qui indiquent respectivement si la fin d'instant a été décidée et si un changement dans l'environnement s'est produit pendant l'activation. Plus précisément $move$ est mis à la valeur vrai lorsqu'une micro-réaction génère un événement. Nous verrons comment ce booléen nous aide à déterminer la fin d'instant. On utilise la notation E_m^{eoi}.

L'environnement A n'était pas présent dans la sémantique d'origine mais a dû être ajouté pour traiter les instructions `add` et `shell`. Il garde la même définition que dans la sémantique comportementale.

Les configurations

Le fait d'avoir une sémantique exécutable et l'ajout du booléen eoi impliquent des modifications sur les fonctions qui testent les configurations. La fonction eoi devient simplement le test du booléen qui annote E.

La différence principale avec la sémantique comportementale réside dans le fait que dans E n'a pas nécessairement toutes les informations qui permettent de déterminer si la valeur d'une configuration est fixée. C'est pour cela que nous définissons la fonction $fixed$. La fonction $fixed$ retourne la valeur vrai si la configuration peut être évaluée. Elle est donc définie ainsi :

$$\begin{array}{rcl}
fixed(present(s), E_m^{eoi}) & = & (s \in E) \vee eoi \\
fixed(not\ C, E_m^{eoi}) & = & fixed(C, E_m^{eoi}) \\
fixed(C_1\ and\ C_2, E_m^{eoi}) & = & \left\{ \begin{array}{ll} & (fixed(C_1, E_m^{eoi}) \wedge fixed(C_2, E_m^{eoi})) \\ \vee & (fixed(C_1, E_m^{eoi}) \wedge \neg eval(C_2, E_m^{eoi})) \\ \vee & (\neg eval(C_1, E_m^{eoi}) \wedge fixed(C_2, E_m^{eoi})) \end{array} \right. \\
fixed(C_1\ or\ C_2, E_m^{eoi}) & = & \left\{ \begin{array}{ll} & (fixed(C_1, E_m^{eoi}) \wedge fixed(C_2, E_m^{eoi})) \\ \vee & (fixed(C_1, E_m^{eoi}) \wedge eval(C_2, E_m^{eoi})) \\ \vee & (eval(C_1, E_m^{eoi}) \wedge fixed(C_2, E_m^{eoi})) \end{array} \right.
\end{array}$$

La définition de $eval$ reste inchangée. Les définitions de sat et $unsat$ dépendent maintenant de la valeur retournée par la fonction $fixed$.

$$sat(C, E_m^{eoi}) = fixed(C, E_m^{eoi}) \wedge eval(C, E_m^{eoi})$$

$$unsat(C, E_m^{eoi}) = fixed(C, E_m^{eoi}) \wedge \neg eval(C, E_m^{eoi})$$

On ajoute une nouvelle fonction $unknown$ qui rend la valeur vrai quand la configuration n'est pas encore fixée.

$$unknown(C, E_m^{eoi}) = \neg fixed(C, E_m^{eoi})$$

Les instructions

Il faut également changer certains opérateurs pour introduire un codage de leur état. L'opérateur de parallélisme `par` doit mémoriser le statut de ses deux branches, on a donc :

$$\texttt{par}_{\alpha,\beta}(P, Q)$$

On ajoute également la forme `until`* à l'instruction `until`. Cette nouvelle forme est utilisée lorsque le corps de cette instruction a terminé son exécution pour l'instant courant. Donc l'opérateur de préemption a deux formes :

$$\texttt{until}(C, P, Q)$$

$$\texttt{until}^*(C, P, Q)$$

Pour les mêmes raisons l'instruction `shell` a deux formes :

$$\texttt{shell}(a, P)$$

$$\texttt{shell}^*(a, P)$$

L'instruction `local` garde en mémoire l'état de l'événement. C'est-à-dire que si l'événement *evt* n'a pas été généré dans le programme P alors l'instruction a la forme :

$$\texttt{local} - (s, P)$$

Mais si l'événement *evt* a été généré alors l'instruction a la forme :

$$\texttt{local} + (s, P)$$

Enfin on fait deux modifications à l'instruction `machine`. On annote cette instruction par les environnements E_m^{eoi} et A. Cette annotation permet de sauvegarder les environnements entre deux micro-réactions. L'autre modification est l'ajout d'une forme étoilée qui comme pour `until` permet à l'instruction `machine` qui a terminé sa réaction pour l'instant courant de savoir comment réagir en attendant la fin d'instant.

$$\texttt{machine}_{E_m^{eoi},A}(P)$$

$$\texttt{machine}^*_{E_m^{eoi},A}(P)$$

1.4.1 L'exécution d'un instant

L'exécution d'un instant consiste à faire réagir le programme jusqu'à ce que les statuts retournés soit *STOP* ou *TERM* ou qu'il n'y ait plus de modifications dans l'environnement. On active le programme jusqu'à obtention d'un point fixe.

Le cas de base est le suivant :

$$\frac{P, E_m^{eoi}, A \xrightarrow{\alpha} P', E_m^{eoi\prime}, A' \quad \alpha \neq SUSP}{P, E_m^{eoi}, A \overset{\alpha}{\Rightarrow} P', E_m^{eoi\prime}, A'}$$

Dans le cas où une micro-réaction ne suffit pas il faut faire des appels récursifs à $\Rightarrow$. Donc tant que *move* est vrai on active le corps.

$$\frac{P, E_m^{eoi}, A \xrightarrow{SUSP} P', E_{true}^{eoi}{}', A' \quad P, E_{false}^{eoi}{}' \overset{\alpha}{\Rightarrow} Q, E_m^{eoi\prime\prime}, A''}{P, E_m^{eoi}, A \overset{\alpha}{\Rightarrow} Q, E_m^{eoi\prime\prime}, A''}$$

Si *move* est faux, alors la fin d'instant est déclarée et on fait réagir le programme jusqu'à ce qu'il se stabilise.

$$\frac{P, E_m^{eoi}, A \xrightarrow{SUSP} P', E_{false}^{eoi}{}', A' \quad P, E_f^{true}alse' \overset{\alpha}{\Rightarrow} Q, E_m^{eoi\prime\prime}, A''}{P, E_m^{eoi}, A \xrightarrow{\alpha} Q, E_m^{eoi\prime\prime}, A''}$$

A chaque nouvel instant on fait réagir le programme dans un environnement réinitialisé.

1.4.2 Les règles de réécriture des instructions

Nothing, Stop et Atom

Les instructions atomiques de base s'expriment ainsi.

$$\texttt{nothing}, E_m^{eoi}, A \xrightarrow{TERM} \texttt{nothing}, E_m^{eoi}, A$$

$$\texttt{stop}, E_m^{eoi}, A \xrightarrow{STOP} \texttt{nothing}, E_m^{eoi}, A$$

$$\texttt{atom}(j), E_m^{eoi}, A \xrightarrow{TERM} \texttt{nothing}, E_m^{eoi}, A$$

Add

L'instruction `add` ajoute seulement une requête d'ajout dans A.

$$\texttt{add}(a, P), E_m^{eoi}, A \xrightarrow{TERM} \texttt{nothing}, E_m^{eoi}, A[a\backslash P]$$

Generate

L'instruction `generate` ajoute un événement dans l'environnement et elle modifie *move* pour spécifier que l'environnement a été modifié.

$$\texttt{generate}(s), E_m^{eoi}, A \xrightarrow{TERM} \texttt{nothing}, E_{true}^{eoi}[s], A$$

Await

Pour cette instruction il faut traiter en plus le cas où la configuration n'est pas encore fixée par rapport à la sémantique comportementale.

$$\frac{sat(C, E_m^{false})}{\texttt{await}(C), E_m^{false}, A \xrightarrow{TERM} \texttt{nothing}, E_m^{false}, A}$$

$$\frac{sat(C, E_m^{true})}{\texttt{await}(C), E_m^{true}, A \xrightarrow{STOP} \texttt{nothing}, E_m^{true}, A}$$

$$\frac{unsat(C, E_m^{eoi})}{\texttt{await}(C), E_m^{eoi}, A \xrightarrow{STOP} \texttt{await}(C), E_m^{eoi}, A}$$

$$\frac{unknown(C, E_m^{eoi})}{\texttt{await}(C), E_m^{eoi}, A \xrightarrow{SUSP} \texttt{await}(C), E_m^{eoi}, A}$$

Seq

Pour l'opérateur de séquence, si le statut retourné par l'activation de sa branche gauche est $SUSP$ ou $STOP$ alors il ne faut pas encore activer la branche droite.

$$\frac{P, E_m^{eoi}, A \xrightarrow{\alpha} P', E_m^{eoi'}, A' \quad \alpha \neq TERM}{P;Q, E_m^{eoi}, A \xrightarrow{\alpha} P';Q, E_m^{eoi'}, A'}$$

sinon on passe en séquence.

$$\frac{P, E_m^{eoi}, A \xrightarrow{TERM} P', E_m^{eoi'}, A' \quad Q, E_m^{eoi'}, A' \xrightarrow{\alpha} Q', E_m^{eoi''}, A''}{P;Q, E_m^{eoi}, A \xrightarrow{\alpha} Q', E_m^{eoi''}, A''}$$

Par

La composition parallèle peut être non déterministe mais ici nous ne présentons que la forme qui correspond à l'opérateur $Merge$ des SugarCubes.

Dans le cas où les deux branches sont dans l'état $SUSP$, on active la branche gauche puis la droite dans la même micro-réaction.

$$\frac{P, E_m^{eoi}, A \xrightarrow{\alpha} P', E_m^{eoi'}, A' \quad Q, E_m^{eoi'}, A' \xrightarrow{\beta} Q', E_m^{eoi''}, A''}{\texttt{par}_{SUSP,SUSP}(P,Q), E_m^{eoi}, A \xrightarrow{\gamma(\alpha,\beta)} \texttt{par}_{\delta_1(\alpha),\delta_2(\beta)}(P',Q'), E_m^{eoi''}, A''}$$

La valeur de $\gamma(\alpha, \beta)$ se trouve grâce au tableau suivant :

α \ β	$TERM$	$STOP$	$SUSP$
$TERM$	$TERM$	$STOP$	$SUSP$
$STOP$	$STOP$	$STOP$	$SUSP$
$SUSP$	$SUSP$	$SUSP$	$SUSP$

et δ_1 et δ_2 sont définis ainsi :

- $\delta_1 = \begin{cases} SUSP & \text{si } \alpha = SUSP \text{ et } \beta = STOP \text{ ou } TERM \\ \alpha & \text{sinon} \end{cases}$
- $\delta_2 = \begin{cases} SUSP & \text{si } \beta = SUSP \text{ et } \alpha = STOP \text{ ou } TERM \\ \beta & \text{sinon} \end{cases}$

S'il n'y a qu'une branche qui est dans l'état $SUSP$ alors on l'active sans toucher à l'autre.

$$\frac{P, E_m^{eoi}, A \xrightarrow{\alpha} P', E_m^{eoi'}, A' \quad \beta \neq SUSP}{\texttt{par}_{SUSP,\beta}(P,Q), E_m^{eoi}, A \xrightarrow{\gamma(\alpha,\beta)} \texttt{par}_{\delta_1(\alpha),\delta_2(\beta)}(P',Q), E_m^{eoi'}, A'}$$

$$\frac{Q, E_m^{eoi}, A \xrightarrow{\alpha} Q', E_m^{eoi'}, A' \quad \alpha \neq SUSP}{\texttt{par}_{\alpha,SUSP}(P,Q), E_m^{eoi}, A \xrightarrow{\gamma(\alpha,\beta)} \texttt{par}_{\delta_1(\alpha),\delta_2(\beta)}(P,Q'), E_m^{eoi'}, A'}$$

Les boucles

On peut exprimer les boucles comme dans la sémantique comportementale à partir de la séquence.

$$\frac{P;\texttt{loop}(P), E_m^{eoi}, A \xrightarrow{\alpha} Q, E_m^{eoi'}, A'}{\texttt{loop}(P), E_m^{eoi}, A \xrightarrow{\alpha} Q, E_m^{eoi'}, A'}$$

$$\frac{n > 0 \quad P;\texttt{repeat}(n-1, P), E_m^{eoi}, A \xrightarrow{\alpha} Q, E_m^{eoi'}, A'}{\texttt{repeat}(n, P), E_m^{eoi}, A \xrightarrow{\alpha} Q, E_m^{eoi'}, A'}$$

$$\frac{n \leq 0}{\texttt{repeat}(n, P), E_m^{eoi}, A \xrightarrow{\alpha} \texttt{nothing}, E_m^{eoi}, A}$$

Until

Nous avons défini deux formes pour l'instruction `until`. La première est utilisée lorsque l'on doit activer le corps. Et on reste dans cette forme si le corps ne retourne pas $STOP$.

$$\frac{P, E_m^{eoi}, A \xrightarrow{\alpha} P', E_m^{eoi'}, A' \quad \alpha \neq STOP}{\texttt{until}(C, P, Q), E_m^{eoi}, A \xrightarrow{\alpha} \texttt{until}(C, P', Q), E_m^{eoi'}, A'}$$

Si le corps retourne $STOP$ on doit activer la forme `until`* de l'instruction.

$$\frac{P, E_m^{eoi}, A \xrightarrow{STOP} P', E_m^{eoi'}, A' \quad \texttt{until}^*(C, P', Q), E_m^{eoi'}, A' \xrightarrow{\alpha} R, E_m^{eoi''}, A''}{\texttt{until}(C, P, Q), E_m^{eoi}, A \xrightarrow{\alpha} R, E_m^{eoi''}, A''}$$

La forme `until`* teste la configuration et fait le traitement qui convient.

$$\frac{sat(C, E_m^{false}) \quad Q, E_m^{false}, A \xrightarrow{Q'}, E_m^{eoi'}, A'}{\texttt{until}^*(C, P, Q), E_m^{false}, A \xrightarrow{\alpha} Q', E_m^{eoi'}, A'}$$

$$\frac{sat(C, E_m^{true})}{\texttt{until}^*(C, P, Q), E_m^{true}, A \xrightarrow{STOP} Q, E_m^{true}, A}$$

$$\frac{unsat(C, E_m^{eoi})}{\texttt{until}^*(C, P, Q), E_m^{eoi}, A \xrightarrow{STOP} \texttt{until}(C, P, Q), E_m^{eoi}, A}$$

$$\frac{unknown(C, E_m^{eoi})}{\texttt{until}^*(C, P, Q), E_m^{eoi}, A \xrightarrow{SUSP} \texttt{until}^*(C, P, Q), E_m^{eoi}, A}$$

When

La branche *then* de l'instruction `when` est exécutée lorsque la configuration est satisfaite.

$$\frac{sat(C, E_m^{false}) \quad P, E_m^{false}, A \stackrel{P'}{\rightarrow}, E_m^{eoi'}, A'}{\texttt{when}(C, P, Q), E_m^{false}, A \stackrel{\alpha}{\rightarrow} P', E_m^{eoi'}, A'}$$

$$\frac{sat(C, E_m^{true})}{\texttt{when}(C, P, Q), E_m^{true}, A \xrightarrow{STOP} P, E_m^{true}, A}$$

Si la configuration est insatisfaite c'est la branche *else* qui est exécutée.

$$\frac{unsat(C, E_m^{false}) \quad Q, E_m^{false}, A \stackrel{Q'}{\rightarrow}, E_m^{eoi'}, A'}{\texttt{when}(C, P, Q), E_m^{false}, A \stackrel{\alpha}{\rightarrow} Q', E_m^{eoi'}, A'}$$

$$\frac{unsat(C, E_m^{true})}{\texttt{when}(C, P, Q), E_m^{true}, A \xrightarrow{STOP} Q, E_m^{true}, A}$$

Enfin si la configuration n'est pas fixée on attend.

$$\frac{unknown(C, E_m^{eoi})}{\texttt{when}(C, P, Q), E_m^{eoi}, A \xrightarrow{SUSP} \texttt{when}(C, P, Q), E_m^{eoi}, A}$$

Control

Pour l'instruction `control`, il faut tester la configuration $present(s)$. On n'a pas besoin de faire des cas en fonction de *eoi* car la configuration $present(s)$ est nécessairement satisfaite avant la fin de l'instant ou insatisfaite après la fin de l'instant.

$$\frac{sat(present(s, E_m^{false})) \quad P, E_m^{false}, A \stackrel{\alpha}{\rightarrow} P', E_m^{eoi'}, A'}{\texttt{control}(s, P), E_m^{false}, A \stackrel{\alpha}{\rightarrow} \texttt{control}(s, P'), E_m^{eoi'}, A'}$$

$$\frac{unsat(present(s, E_m^{true}))}{\texttt{control}(s, P), E_m^{true}, A \xrightarrow{STOP} \texttt{control}(s, P), E_m^{true}, A}$$

$$\frac{unknown(present(s, E_m^{eoi}))}{\texttt{control}(s, P), E_m^{eoi}, A \xrightarrow{SUSP} \texttt{control}(s, P), E_m^{eoi}, A}$$

Local

Les deux formes de l'instruction `local` servent à mémoriser l'état du signal dans le sous-programme. Si le corps rend *SUSP* alors l'état doit être sauvegardé.

$$\frac{P, E_m^{eoi} \backslash s, A \xrightarrow{SUSP} P', E_m^{eoi'}, A' \quad s \notin E_m^{eoi'}}{\texttt{local} - (s, P), E_m^{eoi}, A \xrightarrow{SUSP} \texttt{local} - (s, P), E_m^{eoi'}[E_m^{eoi}(s)], A'}$$

$$\frac{P, E_m^{eoi} \backslash s, A \xrightarrow{SUSP} P', E_m^{eoi'}, A' \quad s \in E_m^{eoi'}}{\texttt{local} - (s, P), E_m^{eoi}, A \xrightarrow{SUSP} \texttt{local} + (s, P), (E_m^{eoi'} \backslash s)[E_m^{eoi}(s)], A'}$$

$$\frac{P, (E_m^{eoi} \backslash s)[s], A \xrightarrow{SUSP} P', E_m^{eoi'}, A' \quad s \in E_m^{eoi'}}{\texttt{local} + (s, P), E_m^{eoi}, A \xrightarrow{SUSP} \texttt{local} + (s, P), (E_m^{eoi'} \backslash s)[E_m^{eoi}(s)], A'}$$

Si l'activation retourne *STOP* ou *TERM* alors l'instruction `local` doit être réinitialisée à sa forme `local`$-$.

$$\frac{P, E_m^{eoi} \backslash s, A \xrightarrow{\alpha} P', E_m^{eoi'}, A' \quad \alpha \neq SUSP}{\texttt{local} - (s, P), E_m^{eoi}, A \xrightarrow{\alpha} \texttt{local} - (s, P), (E_m^{eoi'} \backslash s)[E_m^{eoi}(s)], A'}$$

$$\frac{P, (E_m^{eoi} \backslash s)[s], A \xrightarrow{\alpha} P', E_m^{eoi'}, A' \quad \alpha \neq SUSP}{\texttt{local} + (s, P), E_m^{eoi}, A \xrightarrow{\alpha} \texttt{local} - (s, P), (E_m^{eoi'} \backslash s)[E_m^{eoi}(s)], A'}$$

Link

Comme on ne détaille pas les interactions avec Java l'instruction `link` ne fait qu'activer son corps.

$$\frac{P, E_m^{eoi}, A \xrightarrow{\alpha} P', E_m^{eoi'}, A'}{\texttt{link}(obj, P), E_m^{eoi}, A \xrightarrow{\alpha} \texttt{link}(obj, P'), E_m^{eoi'}, A'}$$

Shell

Comme pour la construction `until` la forme `shell`* est utilisée une fois que le corps du shell a terminé son exécution pour l'instant courant et n'a pas rendu le statut *TERM*. Donc l'instruction `shell` active son corps tant qu'il ne retourne pas le statut *STOP*.

$$\frac{P, E_m^{eoi}, A \xrightarrow{\alpha} P', E_m^{eoi'}, A' \quad \alpha \in \{SUSP, TERM\}}{\texttt{shell}(a, P), E_m^{eoi}, A \xrightarrow{\alpha} \texttt{shell}(a, P'), E_m^{eoi'}, A'}$$

$$\frac{P, E_m^{eoi}, A \xrightarrow{STOP} P', E_m^{eoi'}, A' \quad \texttt{shell}^*(a, P'), E_m^{eoi'}, A' \xrightarrow{\alpha} P'', E_m^{eoi''}, A''}{\texttt{shell}(a, P), E_m^{eoi}, A \xrightarrow{\alpha} P'', E_m^{eoi''}, A''}$$

Une fois que l'on est dans la forme `shell`* on attend la fin de l'instant pour pouvoir faire l'ajout.

$$\texttt{shell}^*(a, P), E_m^{false}, A \xrightarrow{SUSP} \texttt{shell}^*(a, P), E_m^{false}, A$$

$$\texttt{shell}^*(a,P), E_m^{true}, A \xrightarrow{STOP} \texttt{shell}(a, P||A(a)), E_m^{true}, A$$

La machine

L'instruction machine change l'environnement dans lequel elle exécute son corps puis sauvegarde le nouvel environnement.

Si le corps de la machine termine son excution, alors la machine retourne le statut *TERM*.

$$\frac{\texttt{until}(Kill_m, \texttt{link}(obj, \texttt{shell}(a,P)), \texttt{nothing}), E_m^{eoi\prime}, A' \xrightarrow{TERM} Q, E_m^{eoi\prime\prime}, A''}{\texttt{machine}_{E_m^{eoi\prime},A'}(a, obj, P), E_m^{eoi}, A \xrightarrow{TERM} \texttt{machine}_{E_m^{eoi\prime\prime},A''}(a, obj, \texttt{nothing}), E_m^{eoi}, A}$$

Lorsque l'activation du corps retourne le statut *SUSP* et produit un nouvel environnement, alors l'instruction machine sauvegarde cette environnement pour une prochaine activation.

$$\frac{\texttt{until}(Kill_m, \texttt{link}(obj, \texttt{shell}(a,P)), \texttt{nothing}), E_m^{eoi\prime}, A' \xrightarrow{SUSP} \texttt{until}(Kill_m, \texttt{link}(obj, \texttt{shell}(a,P')), \texttt{nothing}), E_m^{eoi\prime\prime}, A''}{\texttt{machine}_{E_m^{eoi\prime},A'}(a, obj, P), E_m^{eoi}, A \xrightarrow{SUSP} \texttt{machine}_{E_m^{eoi\prime\prime},A''}(a, obj, P'), E_m^{eoi}, A}$$

Dans le cas où le corps rend *STOP*, l'instruction machine se réécrit en machine*.

$$\frac{\texttt{until}(Kill_m, \texttt{link}(obj, \texttt{shell}(a,P)), \texttt{nothing}), E_m^{eoi\prime}, A' \xrightarrow{STOP} \texttt{until}(Kill_m, \texttt{link}(obj, \texttt{shell}(a,P')), \texttt{nothing}), E_m^{eoi\prime\prime}, A''}{\texttt{machine}_{E_m^{eoi\prime},A'}(a, obj, P), E_m^{eoi}, A \xrightarrow{SUSP} \texttt{machine}^*_{E_m^{eoi\prime\prime},A''}(a, obj, P'), E_m^{eoi}, A}$$

machine* attend la fin d'instant en renvoyant *SUSP* sans activer son corps.

$$\texttt{machine}^*_{E_m^{eoi\prime},A'}(a, obj, P), E_m^{false}, A \xrightarrow{SUSP} \texttt{machine}^*_{E_m^{eoi\prime},A'}(a, obj, P), E_m^{false}, A$$

Une fois la fin d'instant déclarée, il faut vérifier que la machine n'a pas été tuée de l'extérieur.

$$\frac{Kill_m \in E_m^{eoi}}{\texttt{machine}^*_{E_m^{eoi\prime},A'}(a, obj, P), E_m^{true}, A \xrightarrow{TERM} \texttt{nothing}, E_m^{true}, A}$$

Si la machine n'est pas tuée, il faut exécuter les requêtes d'ajout et réinitialiser l'environnement.

$$\frac{Kill_m \notin E_m^{true}}{\texttt{machine}^*_{E_m^{eoi\prime},A'}(a, obj, P), E_m^{true}, A \xrightarrow{STOP} \texttt{machine}^*_{\emptyset_{false}^{false},\emptyset}(a, obj, P||A(a)), E_m^{true}, A}$$

Pour l'instant suivant on supprime tous les évéments et toutes les requêtes d'ajout de l'environnement de la machine. Les booléens *eoi* et *move* sont mis à la valeur *false*.

1.5 Sémantique grand pas exécutable

La sémantique *Rewrite* est une sémantique exécutable qui a l'avantage d'être simple. Mais sa mise en œuvre pose des problèmes d'efficacité. C'est pour cela que nous donnons une nouvelle sémantique qui a pour but d'être exécutable et de suggérer une implantation efficace. Contrairement aux sémantiques *Rewrite* ou *Replace* qui avancent par itération sur le terme, cette sémantique va toujours en avant. Cette sémantique parcours le terme en exécutant ce qu'elle peut exécuter et en mettant en attente dans $\mathcal{W}$ les instructions qui dépendent d'un événement qui n'a pas encore été généré. L'émission d'un événement s débloque les programmes dans $\mathcal{W}$ qui dépendent de lui.

Dans cette sémantique les réactions sont de la forme :

$$\mathcal{W} \vdash_{E,A} P \longrightarrow \mathcal{W}' \vdash_{E',A'} Q \text{ ou } \mathcal{W} \vdash_{E,A} P \longrightarrow \mathcal{W}' \vdash_{E',A'}$$

où $\mathcal{W}$ est l'ensemble des programmes en attente sur un signal, E est l'ensemble des signaux déjà émis et P est le programme à exécuter. La réaction produit un nouvel ensemble de programmes en attente sur des signaux $\mathcal{W}'$, un nouvel ensemble de signaux émis E' et un programme à exécuter à l'instant suivant Q. A et A' sont les ensembles de requêtes d'ajout. Si la réaction ne donne pas de programme Q à exécuter à l'instant suivant, c'est que le programme a terminé son exécution. L'absence de programme en partie droite de règle est équivalente au statut *TERM* dans les sémantiques précédente.

Nothing

Dans cette sémantique le traitement de l'instruction `nothing` consiste juste à supprimer l'instruction.

$$\mathcal{W} \vdash_{E,A} \texttt{nothing} \longrightarrow \mathcal{W} \vdash_{E,A}$$

Stop

L'instruction `stop` produit en partie droite le programme `nothing`, car cette instruction sera à exécuter à l'instant suivant.

$$\mathcal{W} \vdash_{E,A} \texttt{stop} \longrightarrow \mathcal{W} \vdash_{E,A} \texttt{nothing}$$

Await

Avec l'instruction `await` nous allons voir que cette sémantique va toujours en avant quelque soit l'état de la configuration. Dans les cas où la configuration est satisfaite il suffit de se comporter comme `nothing` ou `stop` suivant la valeur de *eoi*.

$$\frac{sat(C,E) \quad \neg eoi(E)}{\mathcal{W} \vdash_{E,A} \texttt{await}(C) \longrightarrow \mathcal{W} \vdash_{E,A}}$$

$$\frac{sat(C,E) \quad eoi(E)}{\mathcal{W} \vdash_{E,A} \texttt{await}(C) \longrightarrow \mathcal{W} \vdash_{E,A} \texttt{nothing}}$$

Quand la configuration est insatisfaite il faut à l'instant suivant exécuter de nouveau l'instruction $\texttt{await}(C)$.

$$\frac{unsat(C,E)}{\mathcal{W} \vdash_{E,A} \texttt{await}(C) \longrightarrow \mathcal{W} \vdash_{E,A} \texttt{await}(C)}$$

Si la configuration n'est pas fixée alors l'instruction `await`(C) passe en attente.

$$\frac{unknown(C,E)}{\mathcal{W} \vdash_{E,A} \texttt{await}(s) \longrightarrow \mathcal{W}, \texttt{await}(s) \vdash_{E,A}}$$

Generate

La génération d'événement débloque les programmes qui sont en attente dans $\mathcal{W}$. On note $\mathcal{P}_i(s)$ les programmes qui dépendent de l'événement s. Donc la génération de s ajoute s à l'environnement et supprime de l'attente les programmes qui dépendent de ce signal.

$$\frac{\mathcal{W} \vdash_{E[s],A} \mathcal{P}_1(s)||...||\mathcal{P}_n(s) \longrightarrow \mathcal{W}' \vdash_{E',A'} Q}{\mathcal{W}, \mathcal{P}_1(s), ..., \mathcal{P}_n(s) \vdash_{E,A} \texttt{generate}(s) \longrightarrow \mathcal{W}' \vdash_{E',A'} Q}$$

Par

Pour la composition parallèle, il suffit d'activer une branche puis l'autre. Cette règle s'explique par le fait que l'activation de la première branche met ses sous-programmes en attente d'un événement dans l'ensemble $\mathcal{W}$. Puis l'activation de la seconde branche va pouvoir réactiver ces sous-programmes si nécessaire lors de la génération d'événements.

$$\frac{\mathcal{W} \vdash_{E,A} P_1 \longrightarrow \mathcal{W}' \vdash_{E',A'} Q \quad \mathcal{W}' \vdash_{E',A'} P_2 \longrightarrow \mathcal{W}'' \vdash_{E'',A''} R}{\mathcal{W} \vdash_{E,A} P_1||P_2 \longrightarrow \mathcal{W}'' \vdash_{E'',A''} Q||R}$$

Seq

L'expression de la séquence dans ce système de règles n'est pas possible. Le problème est dû à la perte de la structure du programme lors de la réaction. Par exemple le programme $\texttt{await}(s)||\texttt{stop}$ réagit de la façon suivante :

$$\frac{\vdash_{\emptyset,\emptyset} \texttt{await}(s) \longrightarrow \texttt{await}(s) \vdash_{\emptyset,\emptyset} \quad \texttt{await}(s) \vdash_{\emptyset,\emptyset} \texttt{stop} \longrightarrow \texttt{await}(s) \vdash_{\emptyset,\emptyset} \texttt{nothing}}{\vdash_{\emptyset,\emptyset} \texttt{await}(s)||\texttt{stop} \longrightarrow \texttt{await}(s) \vdash_{\emptyset,\emptyset} \texttt{nothing}}$$

Donc si on veut écrire le programme $(\texttt{await}(s)||\texttt{stop}); P$, on a comme problème que la partie gauche a été destructurée. D'autre part, cette perte de la structure des termes pause également des problèmes pour des instructions comme `until` ou `local`.

1.6 Sémantique gloutonne

Cette sémantique suit le principe de la sémantique précédente. Nous introduisons une nouvelle structure $\mathcal{S}$. $\mathcal{S}$ remplace $\mathcal{W}$, il permet de garder la forme du terme à exécuter.

$\mathcal{S}$ est un arbre, où chaque nœud que l'on appelle aussi "groupe" est composé de trois parties :

- Les programmes en attente. Une instruction `await` dont la configuration n'est pas encore fixée est placée dans cette zone.
- Le programme à exécuter à l'instant suivant. L'exécution de l'instruction `stop` va mettre l'instruction `nothing` dans cette partie.
- le programme à exécuter en séquence une fois que tous les programmes du nœud et du sous-arbre ont fini leur exécution.

On associe également une étiquette a et un compteur n à chaque nœud. Les étiquettes permettent de rattacher chaque sous-programme à son groupe. La définition de $\mathcal{S}$ est la suivante :

$$\begin{array}{lcl}
\mathcal{S} & ::= & \{(\mathcal{T}|\mathcal{N}) \triangleright nothing\}^n_{top} \\
\\
\mathcal{T} & ::= & \mathcal{N} \\
 & | & \mathcal{T}, \mathcal{T} \\
 & | & \{(\mathcal{T}|\mathcal{N}) \triangleright prog\}^n_{id} \\
 & | & \{^s(\mathcal{T}|\mathcal{N}) \triangleright prog\}^n_{id} \\
\\
\mathcal{N} & ::= & \emptyset \\
 & | & prog
\end{array}$$

Les groupes sont composés :

- d'un groupe racine qui est d'étiquette top : $\{(\mathcal{T}|\mathcal{N}) \triangleright nothing\}^n_{top}$
- de groupes simples : $\{(\mathcal{T}|\mathcal{N}) \triangleright P\}^n_a$;
- et de groupes qui déclarent local un événement : $\{^s(\mathcal{T}|\mathcal{N}) \triangleright P\}^n_a$.

Dans cette sémantique les réactions sont de la forme suivante :

$$\mathcal{S} \vdash_{E,A} P^a \longrightarrow \mathcal{S}', E', A'$$

Ce qui se lit, le programme P d'étiquette a associé à la structure $\mathcal{S}$ produit dans l'environnement (E, A) une nouvelle structure $\mathcal{S}'$ avec l'environnement (E', A'). Une fois cette réaction terminée, la fin d'instant est décrétée. Il y a alors une seconde phase qui, à partir de $\mathcal{S}$, E et A, construit le terme à exécuter à l'instant suivant.

$$\mathcal{S}, E, A \rightsquigarrow \mathcal{S}', P'$$

La définition de A reste identique aux sémantiques précédentes, mais nous modifions celle de E. Nous gardons dans E, pour chaque événement généré, l'étiquette du groupe associé. On entend par groupe associé le groupe qui déclare s qui contient le groupe dans lequel s a été généré. Si s a été généré dans un endroit où l'événement n'a pas été déclaré local, alors on l'associe dans E à l'étiquette top.

Nous voulons une sémantique "gloutonne", qui avance toujours. Mais nous avons aussi vu que l'on avait des problèmes pour exprimer la séquence. Ici l'idée est de garder la structure du programme lorsque l'on met des instructions en attente. Lors de la réaction d'une séquence $P_1||P_2$, on met la branche droite (P_2) en attente de la branche gauche (P_1). Pour ne pas perdre la structure du programme, on crée un groupe pour P_1 et on indique qu'il faut exécuter P_2 en séquence une fois que ce nouveau groupe aura terminé son exécution.

Donc, dans le cas de la séquence, faire réagir :

$$\mathcal{S}[\{(\mathcal{T}|\mathcal{N}) \triangleright Q\}_a^n] \vdash_{E,A} (P_1; P_2)^a$$

est équivalent à faire réagir :

$$\mathcal{S}[\{(\mathcal{T}, \{(|) \triangleright {P_2}^a\}_b^0|\mathcal{N}) \triangleright Q\}_a^n] \vdash_{E,A} {P_1}^b$$

La technique utilisée pour savoir quand déclencher la séquence est de tester lors de l'exécution d'une instruction qui termine si ce groupe a complètement terminé son exécution. Les instructions qui terminent sont `nothing`, `atom` , `generate` et `await`.

1.6.1 La première phase

Nous allons voir les règles de réaction des instructions avant la fin d'instant.

Nothing

`nothing` est une instruction qui ne fait que terminer. Donc l'exécution de cette instruction ne fait que déclencher le programme qui est en séquence si nécessaire.

$$\frac{seq(a, \mathcal{S}) = (P^b, \mathcal{S}') \quad \mathcal{S}' \vdash_{E,A} P^b \longrightarrow \mathcal{S}'', E', A'}{\mathcal{S} \vdash_{E,A} \texttt{nothing}^a \longrightarrow \mathcal{S}'', E', A'}$$

C'est la fonction *seq* qui va vérifier dans $\mathcal{S}$ si le groupe a a terminé son exécution. Si c'est le cas *seq* retourne le programme à exécuter et une nouvelle structure $\mathcal{S}'$ dans laquelle le groupe a a été supprimé.

$$\begin{array}{rcll} seq(a, \mathcal{S}[\{(\mathcal{T}|\mathcal{N}) \triangleright Q^b\}_a^n]) & = & (\emptyset, \mathcal{S}) & \text{si } \mathcal{N} \text{ ou } \mathcal{T} \neq \emptyset \text{ ou } n \neq 0 \\ seq(a, \mathcal{S}[\{(\mathcal{T}|\mathcal{N}) \triangleright Q^b\}_a^0]) & = & (Q^b, \mathcal{S}\backslash a) & \text{si } \mathcal{N} \text{ et } \mathcal{T} = \emptyset \end{array}$$

Si le groupe a n'a pas terminé son exécution le programme retourné est $\emptyset$. La réaction de $\emptyset$ ne modifie rien.

$$\mathcal{S} \vdash_{E,A} \emptyset \longrightarrow \mathcal{S}, E, A$$

Stop

La réaction de $\texttt{stop}^a$ ajoute dans le groupe a l'instruction `nothing` à faire a l'instant suivant.

$$\mathcal{S}[\{(\mathcal{T}'|\mathcal{N}) \triangleright Q\}_a^n] \vdash_{E,A} \texttt{stop}^a \longrightarrow \mathcal{S}[\{(\mathcal{T}'|(\mathcal{N}||\texttt{nothing})^a) \triangleright Q\}_a^n], E, A$$

La composition parallèle de $\mathcal{N}$ avec un programme P est la mise en parallèle de deux programmes si $\mathcal{N}$ est différent de $\emptyset$. Sinon c'est égal au programme P.

Atom

L'instruction `atom` se comporte comme `nothing`.

$$\frac{seq(a, \mathcal{S}) = (P^b, \mathcal{S}') \quad \mathcal{S}' \vdash_{E,A} P^b \longrightarrow \mathcal{S}'', E', A'}{\mathcal{S} \vdash_{E,A} \texttt{atom}(j)^a \longrightarrow \mathcal{S}'', E', A'}$$

Generate

L'instruction `generate`(s) étant une instruction qui termine, elle va déclencher la séquence. Mais `generate` doit également débloquer les programmes qui sont en attente sur l'événement généré. Donc `generate` fait réagir ces deux programmes dans un environnement où le signal s a été ajouté.

$$\frac{seq(a, \mathcal{S}) = (P^b, \mathcal{S}') \quad gen(a, s, \mathcal{S}') = (d, Q^c, \mathcal{S}'') \quad \mathcal{S}'' \vdash_{E[(d,s)],A} P^b || Q^c \longrightarrow \mathcal{S}''', E', A'}{\mathcal{S} \vdash_{E,A} \texttt{generate}(s)^a \longrightarrow \mathcal{S}''', E', A'}$$

La fonction gen prend en paramètre a le nom du groupe où l'événement a été généré, s l'événement et la structure $\mathcal{S}$. Et cette fonction retourne le triplet $(b, P^c, \mathcal{S}')$ où :

- b est l'étiquette du groupe où s a été déclaré ;
- P^c la composition parallèle de tous les programmes débloqués ;
- $\mathcal{S}'$ est la structure où l'on a supprimé les programmes débloqués.

C'est dans cette fonction que l'on doit faire attention à la portée des signaux.

Await

Comme nous avons deux phases dans l'exécution d'un instant, et qu'ici nous sommes dans la première, alors nous sommes toujours dans le cas où la fin d'instant n'a pas été décidée.

Dans le cas où la configuration est satisfaite, l'instruction `await` termine et se comporte donc comme `nothing`.

$$\frac{sat(C, E, \mathcal{S}) \quad seq(a, \mathcal{S}) = (P^b, \mathcal{S}') \quad \mathcal{S}' \vdash_{E,A} P^b \longrightarrow \mathcal{S}'', E', A'}{\mathcal{S} \vdash_{E,A} \texttt{await}(C)^a \longrightarrow \mathcal{S}'', E', A'}$$

Quand la configuration est insatisfaite, on place alors l'instruction `await` dans la partie à exécuter à l'instant suivant de son groupe.

$$\frac{unsat(C, E, \mathcal{S})}{\mathcal{S}[\{(\mathcal{T}|\mathcal{N}) \triangleright Q\}_a^n] \vdash_{E,A} \texttt{await}(C)^a \longrightarrow \mathcal{S}[\{(\mathcal{T}|(\mathcal{N}||\texttt{await}(C))^a) \triangleright Q\}_a^n], E, A}$$

Enfin, si la configuration n'est pas fixée, c'est dans la partie d'attente que l'on met l'instruction.

$$\frac{unknown(C, E, \mathcal{S})}{\mathcal{S}[\{(\mathcal{T}|\mathcal{N}) \triangleright Q\}_a^n] \vdash_{E,A} \texttt{await}(C)^a \longrightarrow \mathcal{S}[\{(\mathcal{T}, \texttt{await}(C)|\mathcal{N}) \triangleright Q\}_a^n], E, A}$$

Add

L'instruction add ajoute une requête d'ajout, puis termine. Il faut donc faire le traitement de la séquence.

$$\frac{seq(a,\mathcal{S}) = (P^b,\mathcal{S}') \quad \mathcal{S}' \vdash_{E,A[b\backslash P]} P^b \longrightarrow \mathcal{S}'',E',A'}{\mathcal{S} \vdash_{E,A} \mathtt{add}(b,P)^a \longrightarrow \mathcal{S}'',E',A'}$$

Seq

On code la séquence avec un groupe. Ce qui correspond à créer un groupe lorsque l'on rencontre une séquence. On place la branche droite dans la partie séquence du groupe. On lance l'activation de la branche gauche dans ce nouveau groupe, et c'est la dernière instruction de cette branche qui déclenche la branche droite.

$$\frac{\mathcal{S}[\{(\mathcal{T},\{(|) \rhd {P_2}^a\}^0_b|\mathcal{N}) \rhd Q\}^n_a] \vdash_{E,A} {P_1}^b \longrightarrow \mathcal{S}',E',A'}{\mathcal{S}[\{(\mathcal{T}|\mathcal{N}) \rhd Q_a\}^n_a] \vdash_{E,A} ({P_1}^a;{P_2}^a)^a \longrightarrow \mathcal{S}',E',A'}$$

Par

Si on prend la règle suivante :

$$\frac{\mathcal{S} \vdash_{E,A} {P_1}^b \longrightarrow \mathcal{S}',E',A' \quad \mathcal{S}' \vdash_{E',A'} {P_2}^c \longrightarrow \mathcal{S}'',E'',A''}{\mathcal{S} \vdash_{E,A} ({P_1}^b||{P_2}^c)^a \longrightarrow \mathcal{S}'',E'',A''}$$

Regardons alors l'activation du programme $(\mathtt{nothing}||\mathtt{stop});\mathtt{atom}(j)$.

$$\dfrac{\dfrac{\dfrac{seq(a,\{\{(|) \rhd \mathtt{atom}(j)\}^0_a\}_{top}) = (\mathtt{atom}(j),\{\}_{top}) \quad \{\}_{top} \vdash_{\emptyset,\emptyset} \mathtt{atom}(j) \longrightarrow \{\}_{top},\emptyset,\emptyset}{\{\{(|) \rhd \mathtt{atom}(j)\}^0_a\}_{top} \vdash_{\emptyset,\emptyset} \mathtt{nothing}^a \longrightarrow \{\}_{top},\emptyset,\emptyset} \quad \dfrac{\dots}{\{\}_{top} \vdash_{\emptyset,\emptyset} \mathtt{stop}^a \longrightarrow \mathcal{S}',E,A}}{\{\{(|) \rhd \mathtt{atom}(j)\}^0_a\}_{top} \vdash_{\emptyset,\emptyset} (\mathtt{nothing}||\mathtt{stop})^a \longrightarrow \mathcal{S}',E,A}}{\{\}_{top} \vdash_{\emptyset,\emptyset} (\mathtt{nothing}||\mathtt{stop});\mathtt{atom}(j) \longrightarrow \mathcal{S}',E,A}$$

L'instruction atom est exécutée dès le premier instant alors qu'elle ne devrait l'être qu'à l'instant suivant. Le problème vient du fait que lors de l'activation de la première branche du parallèle on a perdu l'information qu'il y avait une seconde branche associée au groupe a.

C'est pour résoudre ce problème que nous avons introduit des compteurs sur les groupes. Donc avant de l'activation de la branche gauche, on incrémente le compteur du groupe. Ceci empêche le déclenchement de la partie séquence du groupe. C'est seulement une fois l'activation de la première branche terminée que l'on décrémente le compteur pour activer la seconde.

La règle du parallèle est donc la suivante :

$$\frac{\mathcal{S}[\{\mathcal{S}_a\}^{n+1}_a] \vdash_{E,A} {P_1}^b \longrightarrow \mathcal{S}'[\{\mathcal{S}'_a\}^m_a],E',A' \quad \mathcal{S}'[\{\mathcal{S}'_a\}^{m-1}_a] \vdash_{E',A'} {P_2}^c \longrightarrow \mathcal{S}'',E'',A''}{\mathcal{S}[\{\mathcal{S}_a\}^n_a] \vdash_{E,A} ({P_1}^b||{P_2}^c)^a \longrightarrow \mathcal{S}'',E'',A''}$$

Boucles

Nous utilisons la séquence pour coder les boucles.

$$\frac{\mathcal{S} \vdash_{E,A} (P; \texttt{loop}(P))^a \longrightarrow \mathcal{S}', E', A'}{\mathcal{S} \vdash_{E,A} \texttt{loop}(P)^a \longrightarrow \mathcal{S}', E', A'}$$

$$\frac{n > 0 \quad \mathcal{S} \vdash_{E,A} (P; \texttt{repeat}(n-1, P))^a \longrightarrow \mathcal{S}', E', A'}{\mathcal{S} \vdash_{E,A} \texttt{repeat}(n, P)^a \longrightarrow \mathcal{S}', E', A'}$$

Il faut faire attention que lorsque n est égal à 0, l'instruction `repeat` termine.

$$\frac{n \leq 0 \quad seq(a, \mathcal{S}) = (Q^b, \mathcal{S}') \quad \mathcal{S}' \vdash_{E,A} Q^b \longrightarrow \mathcal{S}'', E', A'}{\mathcal{S} \vdash_{E,A} \texttt{repeat}(n, P)^a \longrightarrow \mathcal{S}'', E', A'}$$

When

Comme pour `await` nous sommes ici dans le cas ou la fin d'instant n'est pas encore déclarée. Donc si la configuration est fixée on lance directement l'activation du programme qui convient.

$$\frac{sat(C, E, \mathcal{S}) \quad \mathcal{S} \vdash_{E,A} P^a \longrightarrow \mathcal{S}', E', A'}{\mathcal{S} \vdash_{E,A} \texttt{when}(C, P, Q)^a \longrightarrow \mathcal{S}', E', A'}$$

$$\frac{unsat(C, E, \mathcal{S}) \quad \mathcal{S} \vdash_{E,A} Q^a \longrightarrow \mathcal{S}', E', A'}{\mathcal{S} \vdash_{E,A} \texttt{when}(C, P, Q)^a \longrightarrow \mathcal{S}', E', A'}$$

Si la configuration n'est pas encore fixée, l'instruction va se placer en attente dans son groupe.

$$\frac{unknown(C, E, \mathcal{S})}{\mathcal{S}[\{(\mathcal{T}|\mathcal{N}) \triangleright Q\}_a^n] \vdash_{E,A} \texttt{when}(C, P, Q)^a \longrightarrow \mathcal{S}[\{(\mathcal{T}, \texttt{when}(C, P, Q)|\mathcal{N}) \triangleright Q\}_a^n], E, A}$$

Shell

L'instruction $\texttt{shell}(a, P)^a$ crée un nouveau groupe avec son nom comme étiquette. L'instruction placée dans la partie séquence du nouveau groupe est `nothing` avec l'étiquette a. Le fait d'avoir cette instruction en séquence permet de transmettre au groupe a l'information de terminaison de l'instruction `shell`.

$$\frac{\mathcal{S}[\{(\mathcal{T}, \{(|) \triangleright \texttt{nothing}^a\}_b^0|\mathcal{N}) \triangleright Q\}_a^n] \vdash_{E,A} P^b \longrightarrow \mathcal{S}', E', A'}{\mathcal{S}[\{(\mathcal{T}|\mathcal{N}) \triangleright Q\}_a^n] \vdash_{E,A} \texttt{shell}(b, P)^a \longrightarrow \mathcal{S}', E', A'}$$

Local

La déclaration locale d'événement crée un nouveau groupe et associe le sous-programme à ce nouveau groupe.

Les problèmes de localité des événements sont traités au niveau des fonctions *sat*, *unsat*, *unknown* et *gen*. Ce sont ces fonctions qui vont utiliser l'information qui est contenue dans $\mathcal{S}$. L'instruction `local` apporte cette information dans $\mathcal{S}$.

$$\frac{\mathcal{S}[\{(\mathcal{T}_a, \{^s(|) \rhd \mathtt{nothing}^a\}^0_b|\mathcal{N}_a) \rhd Q_a\}^n_a] \vdash_{E,A} P^b \longrightarrow \mathcal{S}', E', A'}{\mathcal{S}[\{(\mathcal{T}_a|\mathcal{N}_a) \rhd Q_a\}^n_a] \vdash_{E,A} \mathtt{local}(s, P)^a \longrightarrow \mathcal{S}', E', A'}$$

1.6.2 La fin d'instant

Nous allons définir la seconde phase de cette sémantique. Le rôle de cette phase est de parcourir la structure de $\mathcal{S}$ pour construire le nouveau programme à exécuter.

Le traitement à faire à chaque nœud est le suivant :

- Pour chaque programme en attente d'un signal, il faut tester la configuration et faire le traitement qui convient.
- Il faut récupérer le programme qui était à exécuter à l'instant suivant.
- Et il faut effectuer les requêtes d'ajout.

On peut constater que le traitement à faire à la fin de l'intant dépend uniquement de la partie des programmes en attente. Nous donnons une règle pour chaque instruction qui peut être bloquée. Mais comme seuls les programmes qui dépendent d'une configuration peuvent être en attente, nous n'avons pas de règles pour toutes les instructions.

Await

Si l'instruction qui était bloquée est `await`, et que maintenant la configuration est satisfaite, alors on peut remplacer l'instruction `await` par `nothing`.

$$\frac{sat(C, E, \mathcal{S}) \quad \mathcal{T}, E, A \rightsquigarrow \mathcal{T}', Q \quad A(a) = R}{\{(\mathtt{await}(C, E), \mathcal{T}|\mathcal{N}) \rhd P\}^n_a, E, A \rightsquigarrow \{(\mathcal{T}'|) \rhd P\}^n_a, (\mathtt{nothing}^a||Q||\mathcal{N}||R^a)^a}$$

On a choisi de mettre `nothing` en parallèle pour être sûr d'avoir un instruction qui peut déclencher la séquence à l'instant suivant. Si la configuration est insatisfaite, on réécrit `await` à la place de `nothing`.

$$\frac{unsat(C, E, \mathcal{S}) \quad \mathcal{T}, E, A \rightsquigarrow \mathcal{T}', Q \quad A(a) = R}{\{(\mathtt{await}(C, E), \mathcal{T}|\mathcal{N}) \rhd P\}^n_a, E, A \rightsquigarrow \{(\mathcal{T}'|) \rhd P\}^n_a, (\mathtt{await}(C)||Q||\mathcal{N}||R^a)^a}$$

When

Si l'instruction `when` n'a pas été fixée avant cette phase, cela veut dire qu'il faudra exécuter une des deux branches à l'instant suivant.

$$\frac{sat(C, E, \mathcal{S}) \quad \mathcal{T}, E, A \rightsquigarrow \mathcal{T}', R \quad A(a) = T}{\{(\mathtt{when}(C, P, Q), \mathcal{T}|\mathcal{N}) \rhd P\}^n_a, E, A \rightsquigarrow \{(\mathcal{T}'|) \rhd P\}^n_a, (P^a||R||\mathcal{N}||T^a)^a}$$

$$\frac{unsat(C, E, \mathcal{S}) \quad \mathcal{T}, E, A \rightsquigarrow \mathcal{T}', R \quad A(a) = T}{\{(\texttt{when}(C, P, Q), \mathcal{T}|\mathcal{N}_a) \triangleright P_a\}_a^n, E, A \rightsquigarrow \{(\mathcal{T}'|) \triangleright P_a\}_a^n, (Q^a||R||\mathcal{N}_a||T^a)^a}$$

Sous-groupe

Dans le cas où on a un groupe dans la zone d'attente, il faut propager les règles de fin d'instant.

$$\frac{\{(\mathcal{T}_b|\mathcal{N}_b) \triangleright P_b\}_b^{n_b}, \mathcal{T}_a, E, A \rightsquigarrow \mathcal{T}_b', Q_b \quad \mathcal{T}_a, E, A \rightsquigarrow \mathcal{T}_a', Q_a \quad A(a) = R}{\{(\{(\mathcal{T}_b|\mathcal{N}_b) \triangleright P_b\}_b^{n_b}, \mathcal{T}_a|\mathcal{N}_a) \triangleright P_a\}_a^{n_a}, E, A \rightsquigarrow \{(\mathcal{T}_b', \mathcal{T}_a'|) \triangleright P_a\}_a^{n_a}, (Q_b||Q_a||\mathcal{N}_a||R^a)^a}$$

Conclusion

Nous arrivons à exprimer la réaction d'un programme sans avoir à itérer notre système de règle pour touver un point fixe. Mais cette sémantique continue à couter cher. Lors de la réaction il y a de la création de groupes pour former la structure $\mathcal{S}$.

Il faudrait voir comment construire la structure $\mathcal{S}$ pour pouvoir travailler directement dessus. Il faudrait avoir une sémantique qui ne travaille pas par recopie mais par déplacement de points de contrôle.

Chapitre 2

Une implantation de Junior

Pour mettre en pratique notre sémantique nous avons réalisé une implantation de JUNIOR qui respecte son principe.

2.1 L'interface

Plusieurs implantations de JUNIOR ont été faites par Frédéric Boussinot, Jean-Ferdinand Susini et Laurent Hazard. Toutes ces implantations écrits en Java et respectent toutes la même interface. Il est donc possible d'exécuter le même programme JUNIOR avec les différentes implantations. Notre implantation utilise également la même interface.

Cette interface définit les classes `Program`, `Machine`, `Environment`, `Configuration`... et la classe `Jr` dans laquelle sont définies les méthodes qui permettent de créer les différents objets JUNIOR. C'est donc en appelant les méthodes de cette classe que l'on crée une machine réactive et les instructions qui composent son programme.

Par exemple, l'écriture d'un programme JUNIOR se fait de la manière suivante.

```
Program p = Jr.Atom(new Print("Hello, World!"));
```

La machine avec le programme `p` comme corps à exécuter.

```
Machine machine = Jr.Machine(p);
```

L'activation d'un instant avec la méthode `react`.

```
machine.react();
```

Pour plus d'informations sur l'API de JUNIOR et la façon d'écrire des programmes , on peut se reporter à [BHS00].

2.2 Les différences entre la sémantique et l'implantation

Il existe des divergences entre la sémantique présentée dans la section 1.6 et l'implantation que nous avons réalisée de JUNIOR. Ceci, d'une part pour rendre la sémantique plus compréhensible, et d'autre part pour en faciliter l'implantation. Il reste des modifications à faire pour qu'implantation et sémantique correspondent parfaitement.

L'implantation actuelle correspond plutôt à des règles de la forme suivante :

$$\mathcal{S} \vdash_{E,A,W} P \longrightarrow \mathcal{S}' \vdash_{E',A',W'} Q$$

Les deux différences majeures sont la création de l'environnement W, et le programme Q rendu par la réaction.

Le programme Q est le programme à exécuter à l'instant suivant. Il n'est donc pas stocké dans la structure $\mathcal{S}$ mais remplacé par un compteur.

L'environnement W, lui, a été introduit pour avoir un ordre dans l'exécution plus proche de celui des SugarCubes. W nous sert à enregistrer les instructions qui étaient en attente et qui ont été débloquées par la génération d'un signal. Cette modification induit des changements dans la gestion des instants : un instant n'est fini que lorsqu'il n'y a plus d'instruction à exécuter et que W est vide.

2.3 La structure de l'implantation

2.3.1 Groupes et Environnements

Un des points clés de notre sémantique est la structure $\mathcal{S}$ que l'on appelle groupe. Cette structure sert d'interface entre le programme et les environnements E, A et W. Par exemple nous verrons que l'instruction `Generate` appelle la méthode `generate` de son groupe pour que l'environnement prenne en compte l'émission de signal.

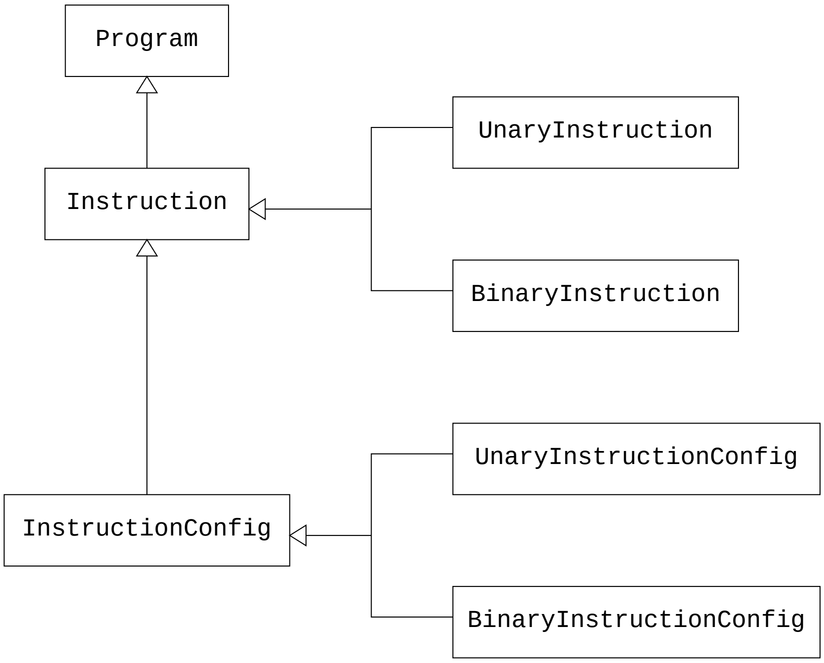
L'architecture que nous avons choisie pour implanter la structure $\mathcal{S}$ est la suivante :

La classe `Groupe` est une classe abstraite qui déclare les méthodes que doivent avoir les groupes. `GroupeImpl` est une implantation générale de `Groupe`. Les classes `GroupeSeq`, `GroupeLocal` et `GroupeLink` sont des groupes qui sont utilisés pour coder des instructions ; elles redéfinissent les méthodes de `GroupeImpl` qui demandent un traitement particulier. Et `GroupeTop` implante la racine de la structure $\mathcal{S}$, c'est dans ce groupe que nous allons principalement stocker les informations des environnements E, A et W.

2.3.2 Instructions et Configurations

Nous avons séparé les instructions en deux familles, selon qu'elles dépendent ou non d'une configuration. Les instructions qui dépendent d'une configuration sont typiquement `Await` ou `When`. Alors que `Nothing`, `Local` ou `Seq` ne dépendent pas de configuration. Ces deux familles sont définies par les classes `InstructionConfig` et `Instruction`.

Ces deux classes sont ensuite spécialisées pour les instructions unaires comme `Link` et binaires comme `Par`. Nous avons donc le diagramme d'héritage suivant :



Les instructions ont un lien vers leur groupe et doivent définir une méthode `rewrite` qui correspond à la règle de réaction de la sémantique. C'est également avec la méthode `rewrite` que l'on implante le mécanisme de déblocage des instructions en attente à la fin d'instant. Il y a dans `GroupeTop` un drapeau qui indique quelle règle utiliser.

Pour les instructions qui dépendent d'une configuration, il y a en plus un lien vers une configuration et une méthode `wakeup` qui est appelée lorsque l'instruction était en attente et que la configuration est fixée. Le comportement par défaut est le suivant :

```
public void wakeup(GroupeTop env){
  groupe.removeWait(this);
  env.addToWakeup(this);
}
```

Ce qui correspond à sortir de la zone d'attente du groupe pour aller s'enregistrer dans l'environnement W.

Le schéma général de traitement des instructions qui dépendent d'une configuration est le suivant. Lors de son activation, l'instruction teste sa configuration. Ce test fait appel à la méthode `fixed` de la configuration. Si `fixed` retourne la valeur vrai alors le traitement de l'instruction peut être fait. Sinon, la configuration s'enregistre auprès de l'environnement en attente sur les événements dont elle dépend et elle retourne le statut `unknown` à l'instruction. L'instruction à son tour se place dans la zone d'attente de son groupe et s'enregistre en attente sur la configuration.

Quand un événement est généré, l'environnement débloque les configurations qui en dépendent. Et si ces configurations sont alors fixées, elles appellent la méthode `wakeup` des instructions qui sont en attente sur elles.

2.3.3 L'implantation des configurations

La classe `Config`

C'est de cette classe que les configurations `Present`, `Not`, `And` et `Or` vont hériter. C'est donc ici que nous allons décrire les comportements qu'elles ont en commun.

Toutes les configurations doivent avoir les deux tables `programmes` et `configurations` qui servent à stocker les programmes et les configurations qui sont en attente que les configurations soient fixées. Il faut donc les fonctions `addProgram` et `addConfig` d'insertion dans ces tables.

Les configurations ont également une méthode `wakeup` qui est appelée quand une des configurations ou un des événements dont elles dépendent est fixé.

```
public void wakeup(Groupe env){
  if (this.fixed(env)) {
     /* on prévient les prog que la config est fixed */
     Enumeration list = programmes.elements();
     while (list.hasMoreElements()){
        InstructionConfig prog = (InstructionConfig)list.nextElement();
        prog.wakeup(env.top);
     }
     /* on supprime tous les programmes de la liste */
     programmes.removeAllElements();

     /* on prévient les config que la config est fixed */
     Enumeration listc = configurations.elements();
     while (listc.hasMoreElements()){
        Config config = (Config)listc.nextElement();
        config.wakeup(env);
     }
     /* on supprime la liste des config */
     configurations.removeAllElements();

  } else {
     /* La config n'est pas encore fixed */
  }
}
```

Cette méthode teste si sa configuration est complètement fixée. Si c'est le cas, alors elle appelle les méthodes `wakeup` des programmes et configurations qui sont en attente et les supprime de ses tables.

Les méthodes `sat` et `unsat` se codent ainsi :

```
public boolean sat(Groupe env){ return fixed(env) && eval(env); }
public boolean unsat(Groupe env){ return fixed(env) && !eval(env); }
```

Il ne reste plus que `fixed` et `eval` à définir pour chaque configuration.

Presence

La configuration `Presence` est fixée si l'événement a été généré ou que l'instant est terminé.

Si la configuration n'est pas fixée, alors il faut que la configuration s'enregistre dans l'environnement en attente de l'événement `event`.

```
public boolean fixed(Groupe env){
  boolean b = (env.isGenerated(event) || env.top.eoi);
  if ((recorded == false) && !b) {
    env.addConfig(event,this);
    recorded = true;
  }
  return b;
}
```

`recorded` est un attribut qui permet de savoir si l'enregistrement auprès de l'environnement a été fait ou non. Cet attribut est réinitialisé lors de l'appel de la méthode `wakeup`.

```
public void wakeup(Groupe env){
  super.wakeup(env);
  recorded = false;
}
```

La méthode `eval` teste si le signal a été émis.

```
public boolean eval(Groupe env){ return env.isGenerated(event); }
```

Not

La configuration `Not` définit ses méthodes `fixed` et `eval` en fonction de la configuration qu'elle a en paramètre.

```
public boolean fixed(Groupe env){
  boolean b = config.fixed(env);
  if ((recorded == false) && !b) {
    config.addConfig(this);
    recorded = true;
  }
  return b;
}

public boolean eval(Groupe env){ return ! config.eval(env); }
```

And et Or

Le traitement de ces deux configurations repose sur le même principe, c'est pour cela que l'on présente uniquement le `And`.

La méthode `fixed` est composée de deux parties, la première détermine si la configuration est fixée et la seconde traite l'enregistrement auprès des configurations qui compose le `And`.

```
public boolean fixed(Groupe env){
  boolean b1 = config1.fixed(env), b2 = config2.fixed(env);
  if (b1 && !config1.eval(env)) return true;
  if (b2 && !config2.eval(env)) return true;
  boolean b = b1 && b2;
  if ((recorded == false) && !b) {
    if (!b1) config1.addConfig(this);
    if (!b2) config2.addConfig(this);
    recorded = true;
  }
  return b;
}
```

L'évaluation se fait simplement en utilisant les booléens de Java.

```
public boolean eval(Groupe env){ return config1.eval(env) &&
config2.eval(env); }
```

2.3.4 L'implantation des instructions

Nous allons voir maintenant plus en détail l'implantation des instructions.

Nothing

`Nothing` est une classe qui hérite de `Instruction`. L'attribut `groupe` est le groupe associé à l'instruction. Et si `groupe` est égal à `null`, c'est que l'instruction est associée à la racine `GroupeTop`.

L'implantation de la règle du `nothing` est donc la suivante :

```
public Instruction rewrite(GroupeTop env){
    if (groupe != null) {
       groupe.removeProgramme(this);
       Instruction seq = groupe.getSequence();
       if (seq == null) {
          return null;
       } else {
          return (seq.rewrite(env));
       }
    }
    return null;
}
```

Si `Nothing` est dans `GroupeTop`, il ne peut pas y avoir de programme à exécuter en séquence, donc `Nothing` ne fait rien et ne donne rien à faire à l'instant suivant. Donc dans ce cas `rewrite` retourne `null`.

Si l'instruction appartient à un groupe il faut voir s'il y a un programme à exécuter en séquence. On appelle donc la méthode `getSequence` qui correspond à la fonction *seq* de la sémantique. S'il n'y a pas de séquence l'instruction retourne `null`, sinon on active le programme trouvé.

Stop

L'exécution de `Stop` consiste à remplacer `Stop` par `Nothing` dans le groupe. Et de dire que cette instruction `Nothing` sera à exécuter à l'instant suivant.

```
public Instruction rewrite(GroupeTop env){
    if (groupe == null) {
       groupe = env;
    } else {
       groupe.removeProgramme(this);
    }
    return new Nothing(groupe);
}
```

Atom

La classe `Atom` a un constructeur qui prend en paramètre un objet de type `Action`. Les objets de type `Action` doivent définir une méthode `execute` qui prend en paramètre un environnement et qui contient du code Java à exécuter. Voici un exemple d'objet `Action` :

```
new Action(){
   public void execute(Environment env){System.out.print("Hello");}
}
```

Le comportement réactif de cette instruction est le même que `Nothing` mais en plus on appelle la méthode `execute` de l'action.

```
public Instruction rewrite(GroupeTop env){
    if (groupe != null) {
       /* On exécute l'action */
       action.execute(groupe);

       /* On se comporte comme Nothing */
       groupe.removeProgramme(this);
       /* pour voir s'il y a un programme à exécuter en sequence */
       Instruction seq = groupe.getSequence();
       if (seq == null) {
          return null;
       } else {
          return (seq.rewrite(env));
       }
    }
    /* Quand l'instruction n'a pas de groupe on exécute l'action dans Top */
    action.execute(env);
    return null;
}
```

L'environnement passé en paramètre est le groupe associé à l'instruction. Ceci est fait pour pouvoir traiter les appels faits à l'objet associé au programme par une instruction `Link`. Voici comment on implante l'appel à la méthode `linkedObject` dans les groupes.

Dans le groupe racine il y a un objet associé par défaut. Un pointeur vers cet objet est gardé dans l'attribut `linkedObject`. Donc le code de la méthode est le suivant :

```
public Object linkedObject(){ return linkedObject; }
```

Pour les groupes qui codent l'instruction `Link` le comportement est le même. Mais pour les autres groupes il faut chercher dans la structure $\mathcal{S}$ le noeud qui définit l'objet associé. Donc dans `GroupeImpl` la définition de `linkedObject` est :

```
public Object linkedObject(){ return surGroupe.linkedObject(); }
```

où `surGroupe` est le père du groupe dans la structure $\mathcal{S}$

Generate

Pour l'instruction `Generate` nous avons la règle de sémantique suivante :

$$\frac{gen(a,s,\mathcal{S}) = (d, P^b, \mathcal{S}') \quad seq(a,\mathcal{S}') = (Q^c, \mathcal{S}'') \quad \mathcal{S}'' \vdash_{E[(d,s)],A,(W||P^b)^{top}} Q^c \longrightarrow \mathcal{S}''' \vdash_{E',A',W'} R^e}{\mathcal{S} \vdash_{E,A,W} (\texttt{generate}(s))^a \longrightarrow \mathcal{S}''' \vdash_{E',A',W'} R^e}$$

Et nous l'avons implantée de la façon suivante :

```
public Instruction rewrite(GroupeTop env){
   if (groupe != null) {
      /* On génère dans le groupe pour pouvoir intercepter
         les signaux locaux */
      groupe.generate(event,val);

      /* On termine comme Nothing */
      groupe.removeProgramme(this);
      /* pour voir s'il y a un programme à exécuter en séquence */
      Instruction seq = groupe.getSequence();
      if (seq == null) {
         return null;
      } else {
         return (seq.rewrite(env));
      }
   }
   /* Sinon on génère dans Top */
   env.generate(idwrap.evaluate(env),objwrap.evaluate(env));
   return null;
}
```

Comme pour `linkedObject`, il faut trouver le groupe qui définit le signal émis. Dans `GroupeImpl`, la méthode `generate` ne fait que transmettre l'information d'émission.

```
public void generate(Identifier event,Object val){
surGroupe.generate(event,val);
}
```

Dans le cas de `GroupeLocal` cette méthode est redéfinie. Elle vérifie si l'émission correspond au signal que le groupe déclare. Si c'est le cas elle fait le traitement de génération, sinon elle transmet l'information.

```
public void generate(Identifier event,Object val){
   if (local.equals(event)){
      localEvent.generate(top.instant,val);

      /* Il faut débloquer les config en attente sur ce signal */
      Enumeration list = attente.elements();
      while (list.hasMoreElements()){
         Config config = (Config)list.nextElement();
         config.wakeup(this);
      }
      /* On supprime les config en attente sur ce signal */
      attente.removeAllElements();
   } else {
      surGroupe.generate(event,val);
   }
}
```

Et dans `GroupeTop` la génération se fait toujours.

```
public void generate(Identifier event,Object val){
   EventDataImpl eventData = getEventData(event);
   eventData.generate(instant,val);

   /* Il faut débloquer les config en attente sur ce signal */
   Vector v = (Vector)attente.get(event);
   if (v != null) {
      Enumeration list = v.elements();
      while (list.hasMoreElements()){
         Config config = (Config)list.nextElement();
         config.wakeup(this);
      }
      /* On supprime les config en attente sur ce signal */
      v.removeAllElements();
   }
}
```

Await

La classe `Await` hérite de la classe `InstructionConfig`. Donc comme nous l'avons vu en 2.3.2, si la configuration est fixée alors `Await` se comporte comme `Nothing` ou `Stop`. Mais si la configuration n'est pas fixée, `Await` se met en attente dans son groupe et s'enregistre auprès de sa configuration.

```
public Instruction rewrite(GroupeTop env){
   /* On fixe le groupe */
```

```
    if (groupe == null) {
      setGroupe(env);
    }
    if (config.sat(groupe)) {
      groupe.removeProgramme(this);
      if (env.eoi) {
        /* on est à la fin de l'instant, il faut se comporter comme Stop */
        return new Nothing(groupe);
      }
      /* on n'est pas à la fin de l'instant, il faut se comporter comme
Nothing */
      Instruction seq = groupe.getSequence();
      if (seq == null) {
         return null;
      } else {
        return (seq.rewrite(env));
      }
    }
    if (config.unsat(groupe)) {
      return this;
    }
    /* Ici la config n'est pas encore fixed */
    /* On passe en attente (c'est la config qui s'enregistre dans l'env) */
    /* on s'enregistre dans la config */
    this.config.addProgram(this);
    groupe.addWait(this);
    return null;
}
```

Seq

L'instruction `Seq` commence par construire un nouveau groupe à l'intérieur du groupe où elle se trouve. Puis elle affecte ses deux branches aux groupes qui conviennent. Et enfin, elle lance l'activation de la branche gauche.

```
public Instruction rewrite(GroupeTop env){
    if(groupe == null) {
      groupe = env;
    } else {
      groupe.removeProgramme(this);
    }
    /* on crée un nouveau sous-groupe */
    GroupeSeq gr = new GroupeSeq(groupe);
    groupe.storeSousGroupe(gr);

    /* on défini le groupe des deux branches de la séquence */
    left.setGroupe(gr);
    right.setGroupe(groupe);
```

```
    /* on met la branche droite dans la partie séquence du groupe */
    gr.storeSequence(right);

    /* on exécute left dans ce sous-groupe */
    return left.rewrite(env);
}
```

On peut remarquer que le groupe construit est de type `GroupeSeq`. Contrairement à la sémantique seul `GroupeSeq` peut garder un programme à exécuter en séquence. On peut supprimer cette zone dans les autres groupes car c'était toujours l'instruction `Nothing` qui y était. L'instruction `Nothing` servait à transmettre l'information de terminaison. Cette transmission est maintenant directement codée dans la méthode `getSequence`.

Voici le code de cette méthode dans `GroupeImpl`.

```
public Instruction getSequence(){
  if (isVide()) {
    Groupe pere = surGroupe;
    surGroupe.removeSousGroupe(this);
    surGroupe = null;
    return pere.getSequence();
  }
  return null;
}
```

Dans `GroupeTop` il ne peut pas y avoir de séquence.

```
public Instruction getSequence(){ return null; }
```

Ce n'est que `GroupeSeq` qui peut avoir un programme à exécuter après la fin du groupe.

```
public Instruction getSequence(){
  if (isVide()) {
    Instruction aux = sequence;
    sequence = null;
    surGroupe.removeSousGroupe(this);
    surGroupe = null;
    return aux;
  }
  return null;
}
```

La méthode `isVide` teste s'il reste des programmes dans le groupe.

Par

Pour la composition parallèle nous avons le code suivant :

```
public Instruction rewrite(GroupeTop env){
  if (groupe == null) {
    groupe = env;
  } else {
```

```
    groupe.removeProgramme(this);
  }
  left.setGroupe(groupe);
  right.setGroupe(groupe);

  Instruction li = left.rewrite(env);
  Instruction ri = right.rewrite(env);

  return result(env,li,ri);
}
```

où `result` construit la composition parallèle de `li` et `ri`. Et la méthode `setGroupe` ne met à jour le groupe que si l'instruction n'a pas de groupe.

On peut constater que par rapport à la règle de sémantique on ne gère pas explicitement de compteur. Ceci n'est pas nécessaire car dans les groupes on a un compteur de programmes. Ce compteur est incrémenté quand une instruction est ajoutée dans un groupe et décrémenté quand une instruction est supprimée. Ce compteur joue bien le même rôle que le compteur de la sémantique.

Local

L'instruction `Local` se transforme en un groupe dans lequel on fait réagir le corps de l'instruction.

```
public Instruction rewrite(GroupeTop env){
  if (groupe == null) {
    groupe = env;
  } else {
    groupe.removeProgramme(this);
  }
  /* on crée un nouveau sous-groupe */
  GroupeLocal gr = new GroupeLocal(groupe, local, localEvent);
  /* On met le corps dans ce sousGroupe */
  body.setGroupe(gr);
  /* On fait réagir le corps */
  return body.rewrite(env);
}
```

2.4 Comparaison avec les autres implantations

Le but de notre implantation est de fournir un interpréteur JUNIOR efficace à la sémantique bien définie. Nous allons voir si nous nous sommes approchés de notre but sur un exemple.

2.4.1 Les performances

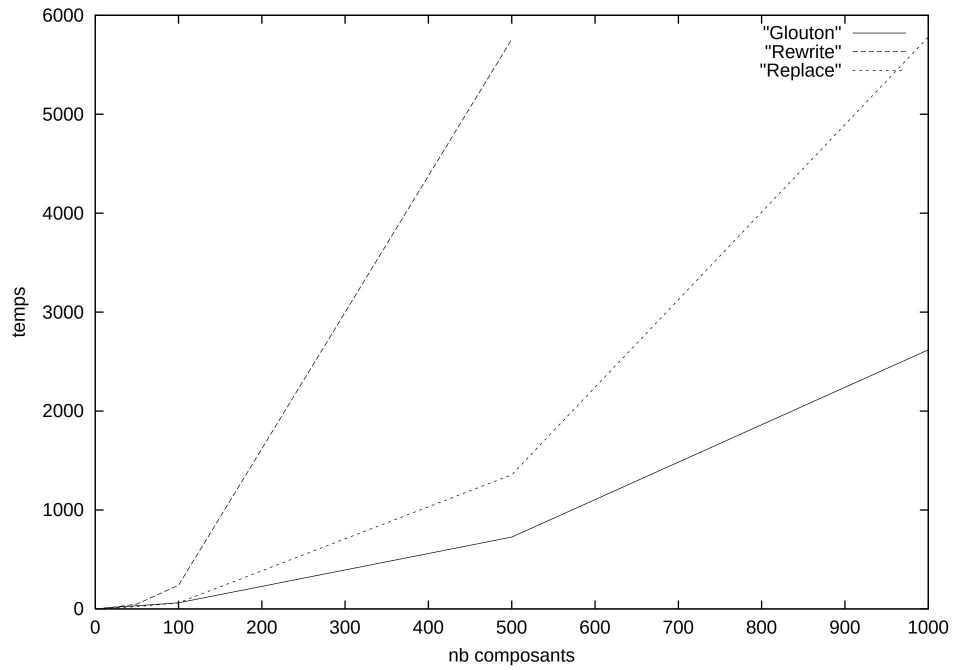
Nous avons testé les performances de notre programme sur l'exemple qui a la structure suivante :

```
Local(s3,
Local(s2,
Local(s1,
  Loop(
    Await(s3); print("fin") ||
    (await(s2); generate(s3) ||
    (await(s1); generate(s2) || generate(s1)))))))
```

Ce programme n'est pas avantageux pour les implantations Rewrite et Replace car les composants sont activés dans le sens inverse du déclenchement des événements.

Le test consistait à activer 50 fois le programme précédent et à mesurer à chaque fois le temps de réaction. Nous avons effectué les tests en faisant varier le nombre de composants parallèles de 10 à 1000.

La courbe suivante donne le temps moyen d'une réaction en milli-secondes en fonction du nombre de composants.



On constate que sur cette exemple notre implantation est nettement plus efficace. La principale raison de notre gain de performances est dûe au fait que nous n'avons besoin que d'un seul parcours du programme. En effet, lors du premier passage nous mettons toutes les instructions en attente dans la structure $\mathcal{S}$ et arrivé à l'émission du premier signal, on débloque directement l'instruction qui convient.

Une autre source de gain de performance est la "compilation" de l'instruction `Local`. Lors de sa première activation, l'instruction `Local` est transformée en `GroupeLocal`. Ceci a pour effet de supprimer l'instruction pour les autres activations, on n'a plus qu'à exécuter le corps du programme.

Mais le fait d'avoir structuré l'environnement a un coup. A chaque traitement d'événement il faut parcourir la structure $\mathcal{S}$ pour retrouver où l'événement a été déclaré.

Un autre handicap de notre implantation est l'allocation de mémoire. Pour chaque instruction `Seq` il y a création d'un nouveau groupe et destruction à la fin de l'exécution de la première branche. La gestion de la mémoire est très coûteuse. On peut le constater dans la différence de durée de deux réactions. Par exemple, dans le cas ou il y a 100 composants, le temps de réaction peut varier de $30ms$ à $210ms$. Cette grande variété est certainement due au GC de Java qui peut prendre la main au cours d'une réaction.

2.4.2 Conclusion

Il existe une implantation très efficace de JUNIOR faite par Laurent Hazard, appelée *Simple*. Cette implantation se base sur une gestion de files d'attente sur les événements. Mais contrairement aux autres implantations, Simple n'a pas de sémantique bien définie. C'est pour cela que notre implantation a l'avantage d'allier sémantique et efficacité.

Il serait intéressant de faire une implantation de notre sémantique qui au lieu de travailler par réécriture comme Rewrite, utilise le déplacement d'un point de contrôle comme Replace. Cette nouvelle implantation apporterait certainement une grande amélioration des performances.

Mais cette première implantation nous a permis de nous donner déjà "une validation par le code" de notre sémantique, et nous montre aussi des performances raisonnables.

Conclusion

Ce stage de DEA a mené à la réalisation de nouvelles sémantiques pour les SugarCubes et d'une implantation de Junior. Nous avons donné une première sémantique comportementale qui nous a permis de mieux comprendre les SugarCubes. Nous avons étendu le sémantique *Rewrite* proposée dans [Sus01] par les opérations d'ajout dynamique de composants. Nous avons réalisé une sémantique "gloutonne". Une nouvelle implantation de Junior basée sur cette sémantique a été réalisé.

Un travail bibliographique important a été réalisé. En particulier sur Fran(Functional Reactive ANimation) [EH97] et FRP (Functional Reactive Programming) [WH00, Hud00], ainsi que sur l'approche de Budde & Poigne [BP01, PBS99].

A court terme les perspectives sont d'affiner la sémantique et l'implantation de Junior. Le but est d'obtenir une implantation par déplacement de point de contrôle. Il faudrait que cette implantation ait des performances se rapprochant de celles des implantations efficaces de Junior, mais en apportant en plus une sémantique clairement définie.

Bibliographie

[BCG87] G. Berry, P. Couronné, and G. Gonthier. Programmation synchrone des systèmes réactifs , le langage Esterel. *Technique et Science Informatique*, 4 :305–316, 1987.

[BdS96] F. Boussinot and R. de Simone. The SL synchronous language. *Software Engineering*, 22(4) :256–266, 1996.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[BHS99] F. Boussinot, L. Hazard, and J-F. Susini. The junior reactive kernel. Technical Report RR-3732, INRIA, 1999.

[BHS00] F. Boussinot, L. Hazard, and J-F. Susini. Programming with junior. Technical Report RR-4027, INRIA, 2000.

[Bou91] F. Boussinot. Reactive-c : An extension of c to program reactive systems. *Software Practice and Experience*, 21(4) :401–428, 1991.

[Bou96] F. Boussinot. *La programmation réactive - Application aux systèmes communicants.* Collection technique et scientifique des télécommunications, masson edition, 1996.

[BP01] Reinhard Budde and Axel Poigné. Complex reactive control with simple synchronous models. *Lecture Notes in Computer Science*, 1985 :19– ? ?, 2001.

[BS98] F. Boussinot and J-F. Susini. The SugarCubes Tool Box : A Reactive Java Framework. *Software-Practice and Experience*, 28(14) :1531–1550, december 1998.

[BS00] F. Boussinot and J-F. Susini. Junior rewrite semantics. Technical report, INRIA, 2000.

[EH97] C. Elliot and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP).* ACM, 1997.

[Hud00] Paul Hudak. *The Haskell School of Expression - Learning Functional Programming Through Multimedia.* Cambrige University Press, New York, 2000.

[PBS99] P.G. Plöger, Reinhard Budde, and Karl H. Sylla. A synchronous object oriented design flow for embedded applications. In *2nd International Forum on Design Languages (FDL)*, Gières, 1999. ECSI Verlag.

[Sus01] J-F. Susini. *L'approche réactive au dessus de Java : sémantique et implémentation des SugarCubes et de Junior.* PhD thesis, Ecole des Mines de Paris, 2001.

[WH00] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of Symposium on Programming Language and Implementation*. ACM, 2000.