ReactiveML, un langage pour la programmation réactive en ML

Louis Mandel & Marc Pouzet

Laboratoire d'Informatique de Paris6,
8 rue du Capitaine Scott 75015 Paris, France
{louis.mandel, marc.pouzet}@lip6.fr

Résumé

Nous présentons REACTIVEML, un langage dédié à la programmation de systèmes réactifs complexes tels que les interfaces graphiques, les jeux vidéo ou les problèmes de simulation. Le langage est basé sur le modèle *réactif synchrone* introduit dans les années 90 par Frédéric Boussinot. Ce modèle combine les principes du modèle synchrone avec la possibilité de créer dynamiquement des processus.

REACTIVEML est une extension conservative d'un langage ML avec appel par valeur. Il ajoute des constructions supplémentaires pour décrire les comportements temporels des systèmes. Nous illustrons l'expressivité du langage sur des exemples et donnons une sémantique formelle du noyau du langage. La sémantique exprime les interactions entre les expressions ML et les constructions réactives. Le langage est statiquement typé avec un système d'inférence de type à la Milner.

1. Introduction

La programmation synchrone [3] a été introduite dans les années 80 pour implanter les systèmes temps-réel critiques. Elle se base sur un modèle idéal où les calculs et les communications sont supposés instantanés. Dans ce modèle, le temps est défini logiquement comme une succession de réactions à des signaux d'entrée. La conséquence principale de ce modèle est de concilier le parallélisme — permettant la description modulaire du système — et le déterminisme.

L'utilisation des langages synchrones est restreinte au domaine des systèmes temps-réel et leur sémantique a été conçue spécialement pour cela. En particulier, ils interdisent la récursion ou l'allocation dynamique de données afin d'assurer une exécution en temps et en mémoire bornés. Dans les années 90, Frédéric Boussinot montre qu'il est possible de concilier les principes de base des langages synchrones avec la possibilité de créer dynamiquement des processus dans le cas où le système ne peut réagir instantanément à l'absence d'un événement. Avec cette restriction, les contradictions logiques dues à la causalité où des signaux sont considérés présents et absents pendant le même instant ne peuvent plus apparaître pendant la combinaison synchrone de processus. Ainsi, il n'y a plus besoin d'effectuer une analyse statique servant à rejeter ces incohérences. Surtout, il devient possible de créer dynamiquement des processus en cours d'exécution. Ce modèle s'appelle le modèle *réactif synchrone* (ou simplement *réactif*). Il a été décrit dans SL [10], un calcul réactif dérivé de ESTEREL [4]. Plus tard, JUNIOR [14] a été introduit pour définir la sémantique des SUGARCUBES [12], une librairie pour la programmation réactive en JAVA. Ce modèle a été utilisé avec succès pour la programmation de systèmes interactifs complexes [11, 12, 2] et apparaît comme une alternative à l'approche classique des threads.

A partir de ces premières expériences, de nombreuses implantations ont été réalisées [6, 12, 20]. Elles ont été faites sous forme de librairies dans des langages de programmation généralistes. Cette approche semble très attirante car elle donne accès à toute l'expressivité du langage hôte et est assez légère à mettre en œuvre. Néanmoins, elle peut conduire à des confusions entre les valeurs du langage hôte et les constructions réactives. De plus avec cette approche, il est difficile d'introduire des techniques de compilation spécifiques, des optimisations et des analyses de programmes.

L'approche que nous avons choisie est de fournir la concurrence au niveau du langage. Nous enrichissons donc un langage ML strict avec de nouvelles primitives de programmation réactive. Un programme REACTIVEML est un ensemble de définitions de valeurs ML et de valeurs réactives que nous appelons des *processus*. Un processus est une machine à états dont le comportement dépend de l'histoire des entrées [1]. Les processus sont composés d'expressions ML et de constructions réactives. La sémantique clarifie la communication entre les deux niveaux du langage. Les programmes sont typés statiquement. L'un des rôle du typage est de faire une séparation nette entre les parties écrites en ML et la partie réactive afin de pouvoir exécuter le code ML sans surcoût. En comparaison avec l'approche sous forme de librairies, nous pensons que l'approche langage est plus sûre et conduit à une programmation plus naturelle.

Le langage REACTIVEML [2] est construit au dessus de OCAML [15]. D'autres experiences d'intégration de l'approche réactive ont été réalisées en C [7] et en JAVA [1].

Nous avons présenté une sémantique à grands pas de REACTIVEML dans [17] où une réaction représente l'exécution d'un instant. Ici, nous présentons une sémantique à petits pas où l'exécution d'un instant est une succession de micro-réactions. Cette sémantique à petits pas permet en particulier de décrire l'entrelacement de l'exécution des branches parallèles et donc de traiter les variables impératives.

La section 2 illustre l'expressivité du langage sur des exemples simples. Le noyau de REACTIVEML est défini dans la section 3. La section 4 présente la sémantique du langage. Le système de type est présenté section 5. La section 6 parle d'implantation. Nous comparons REACTIVEML aux travaux récents section 7.

2. Présentation du Langage

2.1. Les Poissons

Nous illustrons le langage à l'aide d'un problème de simulation. Des poissons se déplacent dans les limites d'un aquarium selon une certaine loi.

Nous définissons d'abord un processus `button` (à cliquer avec une souris). Il est paramétré par ses coordonnées (`frame`) et deux signaux (`mevent` et `oclick`). On suppose l'existence d'une fonction `draw` qui dessine un bouton et une fonction booléenne `inside (x,y) frame` qui teste quand le point de coordonnées `(x,y)` est à l'intérieur des coordonnées de `frame`. Le signal d'entrée `mevent` représente la souris. Ce signal est présent quand le bouton de la souris est appuyé et la valeur associée au signal est la position de souris. Le processus `button` émet le signal de sortie `oclick` si la souris clique à l'intérieur du bouton.

```
let process button frame mevent oclick =
  draw frame;
  loop
    await mevent (pos) in
      if inside pos frame then emit oclick;
  end
```

[1] Un processus correspond à une fonction séquentielle par opposition aux fonctions combinatoires (en reprenant la terminologie des circuits).

[2] La distribution de REACTIVEML et des exemples sont disponibles sur `www-spi.lip6.fr/~mandel/rml`.

Un poisson est une machine à états ayant cinq paramètres. `border`, `init_pos` et `size` définissent les limites de l'aquarium, la position initiale du poisson et sa taille. Avec le signal `others` un poisson reçoit en continu la position de tous les poissons présents dans l'aquarium et il reçoit la position de la souris avec `mevent`.

```
let process fish border init_pos size others mevent =
  signal kill in
  (* start from an initial position *)
  let pos = ref init_pos in
  do
    loop
      (* moves according to some law *)
      emit others (!pos);
      await others (o) in
      clear_fish !pos size;
      pos := law !pos border o;
      draw_fish !pos size;
    end
    ||
    loop
      await mevent (mouse_pos) in
      if inside_fish mouse_pos !pos size then emit kill;
    end
  until kill done;
  clear_fish !pos size
```

Le comportement d'un poisson est le suivant : il définit d'abord un signal local `kill` permettant de supprimer le poisson et une variable impérative `pos` représentant sa position puis il effectue deux actions en parallèle de manière cyclique. La première consiste à (1) émettre sa position, (2) récupérer la position des autres poissons, (3) calculer sa nouvelle position selon la loi `law`. La seconde surveille le signal `mevent` et si la souris clique sur le poisson, alors le signal `kill` est émis. Ces deux actions sont incluses dans une construction de préemption `do/until` de sorte que le processus `fish` termine quand `kill` est présent.

Nous définissons maintenant un banc de poissons. Ce banc (`shoal`) est agrandi avec un nouveau poisson chaque fois que le signal `add` est émis.

```
let rec process shoal frame add others mevent =
  await add;
  (* the initial position is taken randomly *)
  let init_pos = random_pos frame in
  run (fish frame init_pos ((Random.int 10)+1) others mevent)
  ||
  run (shoal frame add others mevent)
```

On peut remarquer que `shoal` est un processus récursif. Lorsque `add` est émis, le processus crée un poisson et se rappelle récursivement pour attendre de nouveau le signal `add`. Il n'y a pas de récursion instantanée car `await add` attend une occurrence de `add` dans le futur strict.

Maintenant, un aquarium est un processus qui contient un banc de poissons et un bouton pour en ajouter. Tous les poissons utilisent le signal `others` pour communiquer leur position. Ce signal doit donc collecter les valeurs émises. Il est déclaré avec la construction

`signal` *nom* `default` *valeur* `gather` *fonction* `in` *expression*

Si plusieurs valeurs sont émises sur *nom* pendant un instant donné, alors ces valeurs sont combinées pour construire la valeur associée au signal à la fin de l'instant. Si la valeur par défaut est d et f est la fonction de combinaison, et si $v_1, \ldots, v_n$ sont les valeurs émises pendant un instant, alors la valeur associée au signal est $(f\ v_n\ ...\ (f\ v_2\ (f\ v_1\ d))...)$. Ici la fonction de combinaison du signal `others` construit la liste des valeurs émises.

```
let process aquarium (x_min,y_min,x_max,y_max) mevent =
  let x_sep = x_min + (x_max-x_min) / 10 in
  (* compute the border for the water *)
  let shoal_frame = x_sep,y_min,x_max,y_max in
  (* compute the border for the create button *)
  let but_frame = x_min,y_min,x_sep,y_max in
  signal others default [] gather fun x y -> x :: y in
  signal add in
  run (button but_frame mevent add)
  ||
  run (shoal shoal_frame add others mevent)
```

Finalement, le programme principal est un processus sans paramètre. Ce processus définit d'abord quelques constantes puis appelle le processus prédéfini `mouse` (qui définit les signaux de souris) et le processus `aquarium` en parallèle.

```
let process main =
  Graphics.open_graph "";
  let border = (0, 0, Graphics.size_x(), Graphics.size_y()) in
  signal mevent default (0,0) gather fun x y -> x in
  run (mouse mevent)
  ||
  run (aquarium border mevent)
```

2.2. Création dynamique et échappement de portée

Nous présentons maintenant un exemple d'ordre supérieur où des processus sont émis sur des signaux. Nous reprenons ici la construction `Jr.Dynapar("add",Jr.Halt())` de Junior introduite dans [1] et utilisée pour la programmation de systèmes d'agents. Le processus `dynapar` reçoit sur un signal `add` des processus et les exécute en parallèle.

```
let rec process dynapar add =
  await add (p) in
  run p || run (dynapar add)
```

Ce processus est utile pour changer le contexte d'exécution d'un processus. Dans l'exemple suivant le processus `p` est défini dans une construction `do/until s` et semble donc dépendre de `s`. Mais ce processus `p` une fois créé, est envoyé sur le signal `add` au `dynapar`. Donc `p` ne peut plus être contrôlé par le signal `s`. C'est le phénomène bien connu d'échappement de portée [19].

```
let process main =
  signal add default (process ())
             gather (fun x y -> process (run x || run y)) in
  run (dynapar add)
  ||
  signal s in
```

```
  do
    ...
    let process p = ... in
    emit add p;
    ...
  until s done
```

Le processus `main` peut émettre plusieurs processus sur le signal `add` pendant un instant. La fonction de combinaison utilisée effectue la composition parallèle de ces processus.

Nous illustrons le problème d'échappement de portée avec le processus `send` qui émet un processus `p1` sur `add` et attend un acquittement de son exécution pour exécuter un processus `p2`.

```
let process send add p1 p2 =
  signal ack in
  emit add (process (run p1; emit ack));
  await immediate ack;
  run p2
```

L'expression `process (run p1; emit ack)` est la définition d'un processus anonyme qui exécute `p1` puis émet `ack`. Le processus `send` crée un signal local `ack`. Ce signal est libre dans le processus `process (run p1; emit ack)` qui est émis sur `add`. La portée de `add` étant plus grande que la portée de `ack`, on dit que `ack` échappe de sa portée.

Le compilateur ReactiveML et de nombreux exemples sont disponibles dans la distribution du langage.

3. Le noyau de ReactiveML

Nous considérons ici le noyau du langage. Une expression (e) peut être une expression ML classique c'est-à-dire une variable (x), une constante (c), une paire ((e, e)), une abstraction ($\lambda x.e$), une application ($e\ e$), une définition locale (`let` x = e `in` e), une récursion (`rec` $x = e$). Une expression (e) peut aussi être une définition de processus (`proc` e) ou une expression réactive : la composition parallèle de deux expressions (e`||`e), une séquence (e`;`e), une boucle infinie (`loop` e), une déclaration de signal (`signal` x `default` e_1 `gather` e_2 `in` e) avec une valeur par défaut e_1 et une fonction de combinaison e_2, un test de présence d'un signal (`present` e `then` e `else` e), une émission (`emit` e e), l'exécution d'une définition de processus (`run` e), une préemption (`do` e `until` e), une suspension (`do` e `when` e) et l'accès à la valeur d'un signal (`let` $e(x)$ `in` e).

$$
\begin{array}{rcl}
e & ::= & x \mid c \mid (e,e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\, x = e \mid \texttt{proc}\ e \\
& & \mid e\texttt{;}e \mid e\texttt{||}e \mid \texttt{loop}\ e \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& & \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \mid \texttt{emit}\ e\ e \\
& & \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{let}\ e(x)\ \texttt{in}\ e \mid \texttt{run}\ e \\
& & \mid \texttt{do}\ e\ \texttt{until}\ e \mid \texttt{do}\ e\ \texttt{when}\ e \\
\\
c & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{0} \mid \ldots \mid \texttt{+} \mid \texttt{-} \mid \ldots
\end{array}
$$

Les programmes donnés en introduction peuvent être traduits facilement dans ce noyau. Par exemple, la définition `let process` $f\ x = e_1$ `in` e_2 est un raccourci pour `let` $f = \lambda x.$`proc` e_1 `in` e_2.

A partir de ce noyau, on peut définir de nouveaux opérateurs :

$$
\begin{array}{lcl}
\texttt{emit}\ e & \stackrel{def}{=} & \texttt{emit}\ e\ \texttt{()} \\
\texttt{present}\ e_1\ \texttt{then}\ e_2 & \stackrel{def}{=} & \texttt{present}\ e_1\ \texttt{then}\ e_2\ \texttt{else ()} \\
\texttt{present}\ e_1\ \texttt{else}\ e_2 & \stackrel{def}{=} & \texttt{present}\ e_1\ \texttt{then ()}\ \texttt{else}\ e_2 \\
\texttt{signal}\ s\ \texttt{in}\ e & \stackrel{def}{=} & \texttt{signal}\ s\ \texttt{default}\ \emptyset\ \texttt{gather}\ \lambda x.\lambda y.\{x\} \uplus y\ \texttt{in}\ e \\
\texttt{pause} & \stackrel{def}{=} & \texttt{signal}\ x\ \texttt{in present}\ x\ \texttt{else ()} \\
\texttt{await immediate}\ s & \stackrel{def}{=} & \texttt{do () when}\ s \\
\texttt{await}\ s & \stackrel{def}{=} & \texttt{pause;await immediate}\ s \\
\texttt{await}\ s(x)\ \texttt{in}\ e & \stackrel{def}{=} & \texttt{await immediate}\ s\texttt{;let}\ s(x)\ \texttt{in}\ e
\end{array}
$$

En ReactiveML, les signaux sont toujours valués : il ont un statut de présence et une valeur associée. Un signal pur (sans valeur) est implanté avec un signal valué qui émet la valeur `()`. La construction **signal** s **in** e est un raccourci pour la déclaration de signal qui collecte dans un multi-ensemble toutes les valeurs émises pendant un instant. $\emptyset$ est le multi-ensemble vide et $\uplus$ est l'union disjonctive (si $m_1 = \{v_1, ..., v_n\}$ et $m_2 = \{v'_1, ..., v'_k\}$ alors $m_1 \uplus m_2 = \{v_1, ..., v_n, v'_1, ..., v'_k\}$).

L'instruction `pause` stoppe l'exécution pendant un instant. En effet contrairement à Esterel mais comme dans SL, la réaction à l'absence d'un signal est retardée d'un instant. Comme x est absent, l'exécution de la branche `else` de l'instruction `present` a lieu à l'instant suivant. La construction `await/immediate` attend la présence d'un signal. Attendre un signal pour récupérer la valeur associée s'écrit **await** $s(x)$ **in** e.

Afin de séparer clairement les parties réactives dont l'exécution peut prendre plusieurs instants des parties instantanées, les expressions doivent suivre des règles de bonne formation. Une expression e bien formée doit vérifier le prédicat $k \vdash e$ où $k \in \{0, 1\}$. Ce prédicat se lit "dans un contexte k l'expression e est bien formée". 0 représente le contexte des expressions ML et 1 le contexte réactif. Lorsque $0 \vdash e$, alors e est une expression *instantanée*. Sinon e est une expression *réactive*.

$$
k \vdash x \qquad k \vdash c \qquad \frac{0 \vdash e}{k \vdash \lambda x.e} \qquad \frac{1 \vdash e}{k \vdash \texttt{proc}\ e} \qquad \frac{0 \vdash e_1 \quad 0 \vdash e_2}{k \vdash e_1\ e_2}
$$

$$
\frac{0 \vdash e_1 \quad 0 \vdash e_2}{k \vdash (e_1, e_2)} \qquad \frac{0 \vdash e_1 \quad k \vdash e_2}{k \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2} \qquad \frac{0 \vdash e}{1 \vdash \texttt{run}\ e} \qquad \frac{0 \vdash e_1 \quad 0 \vdash e_2}{1 \vdash \texttt{emit}\ e_1\ e_2}
$$

$$
\frac{0 \vdash e}{k \vdash \texttt{rec}\ x\ \texttt{=}\ e} \qquad \frac{1 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash e_1 \texttt{||} e_2} \qquad \frac{k \vdash e_1 \quad k \vdash e_2}{k \vdash e_1\texttt{;}e_2} \qquad \frac{1 \vdash e}{1 \vdash \texttt{loop}\ e}
$$

$$
\frac{0 \vdash e \quad 1 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2} \qquad \frac{0 \vdash e_1 \quad 0 \vdash e_2 \quad 1 \vdash e}{1 \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e}
$$

$$
\frac{0 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{let}\ e_1(x)\ \texttt{in}\ e_2} \qquad \frac{0 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{do}\ e_2\ \texttt{until}\ e_1} \qquad \frac{0 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{do}\ e_2\ \texttt{when}\ e_1}
$$

Toute règle $k \vdash e$ est un raccourci pour $0 \vdash e$ et $1 \vdash e$. Elle signifie que e peut être utilisée dans tous les contextes. La règle de typage de $(\lambda x.e)$ indique que le corps d'une fonction doit nécessairement être instantané alors que le corps d'un processus (`proc` e) est typé dans un contexte réactif. Les expressions comme `run`, `loop` ou `present` ne peuvent être utilisées que dans des processus. On peut remarquer qu'aucune règle ne permet de conclure qu'une expression est typée uniquement dans un contexte 0 et pas dans un contexte 1. Par conséquence, toutes les expressions combinatoires peuvent être utilisées dans des processus.

Avec ces règles de bonne formation, nous avons fait des choix dans le design du langage. Par exemple pour la paire nous aurions pu autoriser l'utilisation d'expressions réactives :

$$\frac{k \vdash e_1 \quad k \vdash e_2}{k \vdash (e_1, e_2)}$$

Mais dans ce cas, le programme `(emit s, pause)` peut avoir plusieurs sémantiques. Si l'ordre d'évaluation d'une paire est de gauche à droite, `s` est émis dès le premier instant alors que si l'ordre d'évaluation est de droite à gauche, `s` est émis au second instant. On peut aussi faire le choix d'exécuter les deux expressions en parallèle. Nous trouvons plus naturel d'interdire cette possibilité dans une paire, elle peut être exprimée avec la séquence et la composition parallèle.

Ainsi dans ce langage, les expressions ML sont clairement identifiées. Ceci permet à la fois une plus grande lisibilité des programmes et un traitement particulier des parties instantanées. Les expressions ML peuvent ainsi être exécutées telles quelles sans aucun surcoût.

4. Sémantique opérationnelle

Nous présentons ici la sémantique dynamique de ReactiveML. Cette sémantique est paramétrée par celle des expressions ML (celles pour lesquelles $0 \vdash e$).

4.1. Sémantique de la partie combinatoire

On donne une sémantique grands pas de ML. Ainsi nous devons définir l'ensemble des valeurs (v) :

$$v \quad ::= \quad c \mid n \mid (v, v) \mid \lambda x.e \mid \texttt{proc}\ e$$

Une valeur peut être une constante (c), une valeur de signal n (appartenant à un ensemble dénombrable $\mathcal{N}$), une abstraction ($\lambda x.e$) ou un processus ($\texttt{proc}\ e$).

Pour chaque expression combinatoire e, on définit la relation $e \Downarrow v$:

$$v \Downarrow v \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)} \qquad \frac{e[x \leftarrow \texttt{rec}\ x = e] \Downarrow v}{\texttt{rec}\ x = e \Downarrow v}$$

$$\frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad e[x \leftarrow v_2] \Downarrow v}{e_1\, e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow v_1 \quad e_2[x \leftarrow v_1] \Downarrow v}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v}$$

4.2. Sémantique réactive

La réaction d'un instant est définie en deux étapes. La première est la réaction du programme pendant l'instant. Cette étape est présentée comme une extension de la sémantique à réduction de ML. La seconde étape est la préparation du programme à la réaction à l'instant suivant.

4.2.1. Sémantique à réduction

La réaction d'un instant est représentée par une succession de réactions de la forme $e/S \rightarrow e'/S'$ où S est un *environnement de signaux*. Ces réductions définissent la réaction du programme tout en construisant l'ensemble des valeurs émises.

$$\texttt{let } x \texttt{ = } v \texttt{ in } e/S \xrightarrow{\epsilon} e[x \leftarrow v]/S \qquad v\texttt{;}e/S \xrightarrow{\epsilon} e/S \qquad v_1 \,||\, v_2/S \xrightarrow{\epsilon} \texttt{()}/S$$

$$\texttt{loop } e/S \xrightarrow{\epsilon} e\texttt{;loop } e/S \qquad \texttt{run (proc } e\texttt{)}/S \xrightarrow{\epsilon} e/S$$

$$\texttt{emit } n\ v/S \xrightarrow{\epsilon} \texttt{()}/S + [v/n] \qquad \texttt{present } n \texttt{ then } e_1 \texttt{ else } e_2/S \xrightarrow{\epsilon} e_1/S \text{ si } n \in S$$

$$\texttt{signal } x \texttt{ default } v_1 \texttt{ gather } v_2 \texttt{ in } e/S \xrightarrow{\epsilon} e[x \leftarrow n]/S[(v_1, v_2, \emptyset)/n] \text{ si } n \notin Dom(S)$$

$$\texttt{do } v \texttt{ until } n/S \xrightarrow{\epsilon} \texttt{()}/S \qquad \texttt{do } v \texttt{ when } n/S \xrightarrow{\epsilon} \texttt{()}/S \text{ si } n \in S$$

Figure 1: Réduction en tête

S est défini comme une fonction :

$$S \quad ::= \quad [(d_1, g_1, m_1)/n_1, ..., (d_k, g_k, m_k)/n_k]$$

qui à un nom de signal n_i associe un triplet (d_i, g_i, m_i) où d_i est la valeur par défaut de n_i, g_i est une fonction de combinaison et m_i le multi-ensemble des valeurs émises pendant la réaction. Si $S(n_i) = (d_i, g_i, m_i)$, on note $S^d(n_i) = d_i$, $S^g(n_i) = g_i$ et $S^v(n_i) = m_i$. On note $n \in S$, si le signal n est présent dans l'environnement S ($S^v(n) \neq \emptyset$) et le signal n est absent ($S^v(n) = \emptyset$) est noté $n \notin S$.

On définit l'opérateur $+$ qui ajoute une valeur v à un multi-ensemble associé à un signal n dans un environnement S.

$$(S + [v/n])(n') = \begin{cases} S(n') & \text{si } n' \neq n \\ (S^d(n), S^g(n), S^v(n) \uplus \{v\}) & \text{si } n' = n \end{cases}$$

La relation d'ordre $\sqsubseteq$ sur les environnements est définie par :

$$S_1 \sqsubseteq S_2 \quad \text{ssi} \quad \forall n \in Dom(S_1) : S_1^v(n) \subseteq S_2^v(n)$$

Pour définir la réaction $\rightarrow$, on commence par se donner des axiomes pour la relation de réduction en tête de terme ($\xrightarrow{\epsilon}$) figure 1.

- L'axiome pour le `let` est le même que pour ML.
- La règle pour la séquence indique qu'une fois que la branche gauche est réduite en une valeur, alors cette valeur peut être oubliée.
- Lorsque les deux branches du parallèle sont des valeurs, alors le parallèle est réduit en la valeur `()`.
- L'instruction `loop` duplique son corps.
- Le `run` s'applique à une définition de processus. Il permet son évaluation.
- `emit` $n\ v$ se réduit en `()` et ajoute v au multi-ensemble des valeurs émises sur n.
- La construction `present` peut être réduite seulement si le signal est présent dans l'environnement.
- La réduction de la déclaration d'un signal x dans une expression e renomme x en n où n est un nom frais appartenant à $\mathcal{N}$. n est ajouté à l'environnement de signaux avec v_1 comme valeur par défaut et v_2 comme fonction de combinaison. Initialement le multi-ensemble associé à n est vide car les signaux sont absents par défaut.

- Lorsque le corps d'une construction `do/until` est une valeur, cela signifie que la réaction est terminée. Donc le `do/until` se réduit en `()`.

- Et le `do/when` peut se réduire en `()` quand le corps est une valeur et le signal est présent.

Nous venons de définir les réductions en tête. Nous pouvons maintenant définir la réduction $\rightarrow$ qui exprime les réductions en profondeur :

$$\frac{e/S \xrightarrow{\epsilon} e'/S'}{\Gamma(e)/S \rightarrow \Gamma(e')/S'} \qquad \frac{n \in S \quad e/S \rightarrow e'/S'}{\Gamma(\texttt{do } e \texttt{ when } n)/S \rightarrow \Gamma(\texttt{do } e' \texttt{ when } n)/S'} \qquad \frac{e_{nv} \Downarrow v}{\Gamma(e_{nv})/S \rightarrow \Gamma(v)/S}$$

Où Γ est un contexte d'évaluation. La première règle dit que si une expression e peut se réduire en tête, alors on peut la réduire dans n'importe quel contexte. La seconde règle définit la suspension, elle montre que le corps d'un `do/when` ne peut être évalué que si le signal est présent. La troisième est l'exécution atomique des expressions ML. e_{nv} doit appartenir à l'ensemble des expressions qui ne sont pas des valeurs (cette restriction est importante pour ne pas avoir de réductions infinies).

Les contextes d'évaluation sont définis de la façon suivante :

$$\begin{array}{rcl} \Gamma & ::= & [\,] \mid \texttt{let } x \texttt{ = } \Gamma \texttt{ in } e \mid \Gamma\texttt{;}e \mid \Gamma\texttt{||}e \mid e\texttt{||}\Gamma \mid \texttt{run } \Gamma \\ & & \mid \texttt{emit } \Gamma\ e \mid \texttt{emit } e\ \Gamma \mid \texttt{let } \Gamma(x) \texttt{ in } e \mid \texttt{present } \Gamma \texttt{ then } e \texttt{ else } e \\ & & \mid \texttt{signal } x \texttt{ default } \Gamma \texttt{ gather } e \texttt{ in } e \mid \texttt{signal } x \texttt{ default } e \texttt{ gather } \Gamma \texttt{ in } e \\ & & \mid \texttt{do } e \texttt{ until } \Gamma \mid \texttt{do } \Gamma \texttt{ until } n \mid \texttt{do } e \texttt{ when } \Gamma \end{array}$$

Les contextes pour le parallèle montrent que l'ordre d'évaluation n'est pas spécifié. Dans l'implantation de ReactiveML le choix de l'ordonnancement est fixé de sorte que l'exécution soit toujours déterministe mais il n'est pas donné.

4.2.2. Réaction de fin d'instant

Le modèle réactif est basé sur l'absence de réaction instantanée à l'absence d'un signal. Il faut donc attendre la fin d'instant pour traiter l'absence des signaux et préparer le programme pour l'instant suivant.

La réaction de l'instant s'arrête lorsqu'il n'y a plus de réductions $\rightarrow$ possibles. A partir de ce moment, l'environnement des signaux est figé, il ne peut plus y avoir de nouvelles émissions. Les signaux qui n'ont pas été émis sont absents.

Les règles pour le traitement de fin d'instant ont la forme suivante : $S \vdash e \rightarrow_{eoi} e'$ et sont définies figure 2. Nous pouvons remarquer que les règles ne sont données que pour un sous-ensemble d'expressions car elles seront appliquées seulement quand l'expression e ne peut plus être réduite par $\rightarrow$.

Commentons les règles de la figure 2 :

- Les valeurs restent des valeurs.

- La règle du parallèle applique le traitement de fin d'instant à ses deux branches.

- Le traitement de fin d'instant est appliqué seulement à la blanche gauche de la séquence car la branche droite n'est pas activée pendant l'instant.

- S'il y a une instruction `present` à la fin d'instant, c'est que le signal est absent. Donc à l'instant suivant il faudra exécuter la branche `else`.

$$S \vdash v \rightarrow_{eoi} v \qquad \frac{S \vdash e_1 \rightarrow_{eoi} e_1' \quad S \vdash e_2 \rightarrow_{eoi} e_2'}{S \vdash e_1 \,||\, e_2 \rightarrow_{eoi} e_1' \,||\, e_2'} \qquad \frac{S \vdash e_1 \rightarrow_{eoi} e_1'}{S \vdash e_1 ; e_2 \rightarrow_{eoi} e_1' ; e_2}$$

$$\frac{n \notin S}{S \vdash \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow_{eoi} e_2} \qquad \frac{S(n) = (d, g, m) \quad v = \mathit{fold}\ g\ m\ d}{S \vdash \texttt{let}\ n(x)\ \texttt{in}\ e \rightarrow_{eoi} e[x \leftarrow v]}$$

$$\frac{n \notin S \quad S \vdash e \rightarrow_{eoi} e'}{S \vdash \texttt{do}\ e\ \texttt{until}\ n \rightarrow_{eoi} \texttt{do}\ e'\ \texttt{until}\ n} \qquad \frac{n \in S}{S \vdash \texttt{do}\ e\ \texttt{until}\ n \rightarrow_{eoi} \texttt{()}}$$

$$\frac{n \in S \quad S \vdash e \rightarrow_{eoi} e'}{S \vdash \texttt{do}\ e\ \texttt{when}\ n \rightarrow_{eoi} \texttt{do}\ e'\ \texttt{when}\ n} \qquad \frac{n \notin S}{S \vdash \texttt{do}\ e\ \texttt{when}\ n \rightarrow_{eoi} \texttt{do}\ e\ \texttt{when}\ n}$$

Figure 2: Réaction de fin d'instant

- Le `let` $n(x)$ `in` e récupère les valeurs associées au signal n et les combine avec la fonction *fold* $g\ m\ d$ pour obtenir une valeur v. x est substitué par v dans e pour l'instant suivant. La fonction *fold* est définie par :

$$\begin{array}{lcl} \mathit{fold}\ f\ (\{v_1\} \uplus m)\ v_2 & = & \mathit{fold}\ f\ m\ (f\ v_1\ v_2) \\ \mathit{fold}\ f\ \emptyset\ v & = & v \end{array}$$

- Le traitement de la préemption a lieu en fin d'instant. Si le signal qui contrôle le `do/until` est présent, l'expression doit être preemptée. Dans ce cas, le `do/until` est réécrit en `()`, le programme qui ne fait rien.

- Et pour le `do/when`, si le signal est présent il faut propager le traitement de fin d'instant alors que s'il est absent il ne faut pas car le corps n'a pas été activé pendant l'instant.

4.2.3. Exécution d'un programme

L'exécution d'un programme est une succession d'instants. Un instant (i) lit des entrées I_i et produit des sorties O_i (et des signaux locaux). L'exécution d'un instant i est définie par la relation :

$$e_i/S_i \Rightarrow e_i'/S_i'$$

Avec S_i tel que $I_i = S_i^v$, ce qui veut dire que tous les signaux qui ne sont pas donnés en entrée sont absents. Les signaux observés sont un sous-ensemble de l'environnement en fin de réaction, $O_i \sqsubseteq S_i'^v$. Les valeurs par défaut et les fonctions de combinaison sont conservées d'une réaction à l'autre, $S_i'^d \subseteq S_{i+1}^d$ et $S_i'^g \subseteq S_{i+1}^g$.

L'exécution d'un instant se décompose en deux étapes. La réduction de e_i jusqu'à l'obtention d'un point fixe. Puis la préparation à l'instant suivant.

$$\frac{e_i/S_i \hookrightarrow e_i''/S_i' \quad S_i' \vdash e_i'' \rightarrow_{eoi} e_i'}{e_i/S_i \Rightarrow e_i'/S_i'}$$

Où $e/S \hookrightarrow e'/S'$ si $e/S \rightarrow^* e'/S'$ et $e'/S' \not\rightarrow$. La relation $\rightarrow^*$ est la fermeture réflexive et transitive de $\rightarrow$.

Le noyau du langage est défini sans références. On peut l'étendre très simplement (comme pour ML) afin qu'il traite les variables impératives. Le problème de cette extension est qu'elle fait perdre le déterminisme car l'ordre d'évaluation du parallèle n'est pas spécifié. Mais on peut remarquer que contrairement aux threads, même si deux branches parallèles accèdent à une donnée partagée, la donnée n'a pas besoin d'être protégée. En effet, toutes les actions combinatoires ($0 \vdash e$) ne peuvent pas être interrompues. Dans le cas où les opérations faites en parallèle sur la mémoire sont associatives et commutatives alors le programme reste déterministe.

5. Typage

Nous fournissons un système de type comme une extension du système de type de ML de Milner [18]. Ainsi, nous devons traiter les signaux et en particulier les valeurs qui peuvent être transmises sur les signaux. Le langage de type est :

$$\begin{array}{lcl} \sigma & ::= & \forall \alpha_1, \ldots, \alpha_n . \tau \\ \tau & ::= & T \mid \alpha \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \texttt{process} \mid (\tau, \tau)\,\texttt{event} \\ T & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots \end{array}$$

Un type (τ) peut être un type de base (T), une variable de type (α), le type d'une fonction ($\tau_1 \rightarrow \tau_2$), un type produit ($\tau_1 \times \tau_2$), le type d'un processus (`process`) ou le type d'un signal ($(\tau_1, \tau_2)\,\texttt{event}$). Dans le type $(\tau_1, \tau_2)\,\texttt{event}$, τ_1 est le type des valeurs émises sur le signal alors que τ_2 désigne le type de la valeur associée au signal en lecture (obtenue après la combinaison de toutes les valeurs émises pendant un instant). σ est un schéma de type.

Un environnement de typage H a la forme suivante :

$$H \quad ::= \quad [x_1 : \sigma_1; \ldots; x_k : \sigma_k]$$

L'instanciation et la généralisation sont définies comme suit :

$$\begin{array}{lcl} \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] & \leq & \forall \alpha_1, \ldots, \alpha_k . \tau \\ Gen(\tau, H) & = & \forall \alpha_1, ..., \alpha_n . \tau \text{ où } \{\alpha_1, ..., \alpha_k\} = FV(\tau) - FV(H) \end{array}$$

Les expressions sont typées dans un environnement de typage initial TC tel que :

$$TC = [\texttt{true} : \texttt{bool}; \texttt{fst} : \forall \alpha, \beta.\ \alpha \times \beta \rightarrow \alpha; ...]$$

Les expressions sont typées par un prédicat de la forme $H \vdash e : \tau$ définit figure 3. Les règles de typage des expressions ML ne sont pas modifiées. Dans le typage de `signal`, la valeur par défaut (e_1) a le type de la valeur associée et la fonction de combinaison (e_2) est une fonction qui prend en paramètre une valeur émise et la combinaison des valeurs déjà émises et retourne la nouvelle combinaison. La règle pour `emit` vérifie que le premier argument est de type $(\tau_1, \tau_2)\,\texttt{event}$ (le type des signaux) et que le type de la valeur émise est τ_1. `let` $e_1(x)$ `in` e récupère la valeur associée à un signal. Ainsi, si e_1 a le type $(\tau_1, \tau_2)\,\texttt{event}$, x doit avoir le type τ_2. L'instanciation `run` e est appliquée à un processus. Les constructions `present`, `until` et `when` peuvent être appliquées à n'importe quel signal.

6. Implantation et expériences

Nous avons développé le langage comme une extension d'un sous ensemble d'OCAML [15] (sans objets, labels, ni foncteurs) dans laquelle on peut mélanger des processus réactifs et des expressions OCAML. Des programmes REACTIVEML et des librairies OCAML peuvent être appelés depuis du code REACTIVEML. Les programmes sont typés puis traduits en code OCAML. Cette traduction ne modifie pas les expressions ML alors que chaque expression réactive est traduite en un programme OCAML. Les programmes réactifs peuvent finalement être exécutés en les liant à une librairie ad-hoc.

$$\frac{\tau \leq H(x)}{H \vdash x : \tau} \qquad \frac{\tau \leq TC(c)}{H \vdash c : \tau} \qquad \frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{H[x : \tau_1] \vdash e : \tau_2}{H \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$\frac{H \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad H \vdash e_2 : \tau_1}{H \vdash e_1\, e_2 : \tau_2} \qquad \frac{H \vdash e_1 : \tau_1 \quad H[x : Gen(\tau_1, H)] \vdash e_2 : \tau_2}{H \vdash \texttt{let}\ x \texttt{ = } e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{H \vdash e : \tau}{H \vdash \texttt{proc}\ e : \texttt{process}} \qquad \frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash e_1 \texttt{;} e_2 : \tau_2} \qquad \frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash e_1 \texttt{||} e_2 : \texttt{unit}}$$

$$\frac{H \vdash e : \texttt{process}}{H \vdash \texttt{run}\ e : \texttt{unit}} \qquad \frac{H \vdash e : \tau}{H \vdash \texttt{loop}\ e : \texttt{unit}} \qquad \frac{H \vdash e_1 : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e_2 : \tau_1}{H \vdash \texttt{emit}\ e_1\ e_2 : \texttt{unit}}$$

$$\frac{H \vdash e_1 : (\tau_1, \tau_2)\,\texttt{event} \quad H[x : \tau_2] \vdash e : \tau}{H \vdash \texttt{let}\ e_1(x)\ \texttt{in}\ e : \texttt{unit}} \qquad \frac{H \vdash e : (\tau, \tau')\,\texttt{event} \quad H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \texttt{unit}}$$

$$\frac{H \vdash e_1 : \tau_2 \quad H \vdash e_2 : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2 \quad H[s : (\tau_1, \tau_2)\,\texttt{event}] \vdash e : \tau}{H \vdash \texttt{signal}\ s\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e : \texttt{unit}}$$

$$\frac{H \vdash e_1 : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e : \tau}{H \vdash \texttt{do}\ e\ \texttt{until}\ e_1 : \texttt{unit}} \qquad \frac{H \vdash e_1 : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e : \tau}{H \vdash \texttt{do}\ e\ \texttt{when}\ e_1 : \texttt{unit}}$$

Figure 3: Système de Type

Contrairement à l'approche synchrone classique, les programmes ne sont pas ordonnancés statiquement mais dynamiquement en fonction de la présence/absence des signaux à l'exécution.

La stratégie d'ordonnancement que nous avons implantée est une stratégie *gloutonne* reprenant une technique introduite par Laurent Hazard [3]. Cette technique est connue pour être une des plus efficaces pour l'ordonnancement de Junior. Elle est basée sur l'utilisation de files d'attente : une action est activée seulement lorsque le signal qu'elle attend est émis. Pendant l'exécution, l'interpréteur garde dans $\mathcal{W}$ les actions qui dépendent de la présence de signaux. A la fin d'instant, quand $\mathcal{W}$ n'est pas vide il faut conserver les actions qui ne réagissent pas à l'absence des signaux.

Pour implanter cette technique gloutonne d'ordonnancement, nous associons deux files d'attente à chaque signal. Une file est utilisée pour les instructions qui attendent seulement un instant (comme `present`) et l'autre file est utilisée pour les instructions qui peuvent attendre plus d'un instant (comme `do/when`). Ainsi, si l'exécution d'une expression est stoppée par le test d'un signal alors le code à exécuter est enregistré dans la bonne file d'attente. Sinon sa continuation est mise dans un ensemble d'actions à exécuter dans l'instant courant ($\mathcal{C}$). L'exécution d'un instant consiste en l'exécution de toutes les actions qui sont dans l'ensemble $\mathcal{C}$. La fin d'instant est déclarée quand $\mathcal{C}$ est vide. Alors les instructions qui sont dans les files d'attente à court terme sont traitées pour préparer l'instant suivant.

Avec cette technique d'ordonnancement, un accès rapide aux signaux (pour le test de présence et l'accès aux files d'attente) est crucial pour avoir une exécution efficace des programmes. La plupart des implantations de l'approche réactive utilisent des tables de hachage pendant l'exécution pour

[3] Cette technique d'ordonnancement est bien connue dans la communauté synchrone, malheureusement elle n'a jamais été publiée.

% de cellules actives	0 %	4 %	42 %
OCAML	0.74 s	0.75 s	0.76 s
LOFT	0.02 s	0.11 s	0.93 s
REACTIVEML	0.05 s	0.08 s	0.89 s

Figure 4: Temps d'exécution moyen d'un instant pour un automate cellulaire de Fredkin de taille 500x500.

représenter l'environnement de signaux. Dans notre implantation les signaux sont des valeurs comme les autres et le nommage est fait par OCAML. Cette représentation des signaux est bien plus efficace et plus sûre que dans les autres implantations de l'approche réactive. De plus, on n'a pas besoin d'écrire un glaneur de cellules dédié pour les signaux comme cela est fait dans certaines implantations [13].

De nombreuses applications ont été écrites en REACTIVEML. Ces exemples vont de petits exemples graphiques jusqu'à des problèmes de simulations complexes. En particulier, nous avons programmé l'exemple des automates cellulaires écrit en LOFT par Boussinot [9] pour tester l'efficacité de notre implantation. La rapidité de cet exemple repose sur l'absence d'attente active. Les cellules inactives sont stoppées sur l'attente d'un signal d'activation de sorte que seulement les cellules actives sont exécutées. La figure 4 compare le temps d'exécution pour LOFT, REACTIVEML et une version impérative écrite en OCAML. La version impérative parcours le tableau des cellules avec des boucles `for`. Ces chiffres montrent que l'implantation de REACTIVEML est aussi efficace que celle de LOFT qui est écrite en C et lorsqu'il y a peu de cellules actives, l'approche paresseuse implantée en REACTIVEML et en LOFT est plus efficace que la version impérative.

La principale application est un simulateur de réseaux mobiles ad-hoc [16]. Il a été réalisé en collaboration avec Farid Benbadis (de l'équipe réseau du LIP6, Paris). Il s'agit d'un problème de simulation algorithmiquement coûteux puisque chaque nœud du réseau doit calculer à chaque instant l'ensemble des nœuds avec lesquels il peut communiquer directement. Le simulateur apparaît efficace et capable de simuler des systèmes de plus de 1000 nœuds. Afin de s'assurer de sa robustesse, nous avons exécuté une simulation pendant 20 jours et elle a utilisé une quantité de mémoire constante. Ce simulateur a permis de valider les choix d'implantation et en particulier la technique d'ordonnancement gloutonne sans laquelle la simulation n'aurait pas été possible.

7. Travaux similaires

Depuis REACTIVEC [6] et les SUGARCUBES [12], de nombreux travaux sont réalisés autour de la programmation réactive. Nous les rappelons brièvement.

Les FAIR THREADS [8, 21] sont une extension de l'approche réactive avec la possibilité de mélanger de l'ordonnancement préemptif et coopératif. Dans ce modèle plusieurs ordonnanceurs synchrones peuvent être exécutés de façon asynchrone. Les threads peuvent changer d'ordonnanceur en cours d'exécution ou même se détacher de tout ordonnanceur pour être exécutés de façon autonome. Les threads qui peuvent être exécutés sans ordonnanceur doivent être implantés au dessus des threads du système, ce qui en limite le nombre et pose des problèmes d'efficacité. Dans les FAIR THREADS, il y a du parallélisme au plus haut niveau uniquement : on ne peut pas écrire directement $(e_1 \,||\, e_2);e_3$, et il n'y a pas de structures de contrôle réactives imbriquées.

ULM [5] utilise les principes de la programmation réactive de Boussinot dans le cadre de la mobilité. ULM est plongé dans un λ-calcul avec appel par valeur. En ULM, les références sont codées comme des signaux car accéder à une référence qui n'est pas locale suspend l'exécution jusqu'à ce qu'elle devienne présente. En REACTIVEML, nous ne traitons pas le problème de la mobilité, donc l'accès à une référence est toujours instantané. En comparaison avec REACTIVEML, ULM n'impose pas de contraintes sur les endroits où l'on peut utiliser des constructions réactives, ceci induit un surcoût à l'exécution des programmes non réactifs. En effet, ne sachant pas si l'évaluation d'une expression peut être suspendue ou pas, il faut toujours mettre en place des mécanismes permettant de sauvegarder le contexte d'exécution alors qu'en REACTIVEML, on identifie les expressions ininterruptibles pour ne pas avoir à mettre en place des mécanismes de suspension et de reprise. En ULM, le parallélisme est comme pour les FAIR THREADS seulement au plus haut niveau mais les threads héritent des contextes de préemption et suspension de leurs parents.

Il existe également des plongements de l'approche réactive dans un langage impératif [7] ou dans un langage orienté objet [1]. Comparé à ces travaux, REACTIVEML donne une sémantique complète du plongement avec les signaux values. Ceci permet de comprendre précisément les communications entre le niveau combinatoire et réactif. En particulier cela permet de faire apparaître le problème d'échappement de portée.

8. Conclusion

Cet article présente une extension réactive d'un langage à la ML. Le langage obtenu est dédié à la programmation de systèmes dynamiques complexes présents dans les interfaces graphiques, les jeux vidéo ou les problèmes de simulation.

De nombreuses extensions de ce langage peuvent être considérées. La première d'entre elles est l'identification de sous programmes qui peuvent être ordonnancés statiquement, c'est à dire compilés. Le phénomène d'échappement de portée, qui n'existe pas dans les langages synchrones classiques rend cette compilation difficile. L'ajout d'une forme de threads de services à la manière des FAIR THREADS pour traiter les entrées/sorties bloquantes est également à faire. Enfin, l'ajout d'un opérateur de retard (`pre`) et la possibilité d'écrire des équations dataflow est dans le prolongement de ce travail.

Bibliographie

[1] R. Acosta-Bermejo. *Rejo Langage d'Objets Réactifs et d'Agents*. PhD thesis, Ecole des Mines de Paris, 2003.

[2] Raul Acosta-Bermejo. Reactive operating system, reactive java objects. In *NOTERE'2000*, Paris, November 2000. ENST.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 2003.

[4] G. Berry. The constructive semantics of Esterel, 1998.

[5] Gérard Boudol. ULM a core programming model for global computing. In *Proceedings of the European Symposium on Programming*, 2004.

[6] Frédéric Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, Apr 1991.

[7] Frédéric Boussinot. Concurrent programming with Fair Threads: The LOFT language, 2003.

[8] Frédéric Boussinot. FairThreads: mixing cooperative and preemptive threads in C. Research report 5039, INRIA, 2003.

[9] Frédéric Boussinot. Reactive programming of cellular automata. Technical Report 5183, INRIA, 2004.

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

[11] Frédéric Boussinot, Jean-Ferdinand Susini, Frédéric Dang Tran, and Laurent Hazard. A reactive behavior framework for dynamic virtual worlds. In *Proceedings of the sixth international conference on 3D Web technology*, pages 69–75. ACM Press, 2001.

[12] Frédéric Boussinot and Jean-Ferdinant Susini. The sugarcubes tool box - a reactive java framework. *Software Practice and Experience*, 28(14):1531–1550, 1998.

[13] Christian Brunette. *Construction et simulation graphiques de comportements: le modèle des Icobjs.* PhD thesis, Université de Nice-Sophia Antipolis, 2004.

[14] Laurent Hazard, Jean-Ferdy Susini, and Frédéric Boussinot. The Junior reactive kernel. Research report 3732, INRIA, 1999.

[15] Xavier Leroy. The Objective Caml system release 3.08. Documentation and user's manual. Technical report, INRIA, 2004.

[16] Louis Mandel and Farid Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. Submitted to publication.

[17] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. Technical report, Laboratoire d'Informatique de Paris 6, 2005. Available at `http://www-spi.lip6.fr/~mandel/rml`.

[18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[19] Robin Milner. *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press, 1999.

[20] Riccardo Pucella. Reactive programming in Standard ML. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 48–57. IEEE Computer Society Press, 1998.

[21] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme fair threads. In *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 203–214. ACM Press, 2004.