Orchestration synchrone et au-delà

Pejman Attar [*] — **Frédéric Boussinot** [**]

[*] *INRIA - Indes*
Pejman.Attar@inria.fr
[**] *Mines-ParisTech Cemef / INRIA - Indes*
Frederic.Boussinot@mines-paristech.fr

RÉSUMÉ. *Nous présentons le langage d'orchestration DSL construit sur le modèle réactif synchrone. En DSL, les systèmes sont composés de plusieurs sites exécutés de manière asynchrone. La découpe en sites permet de profiter partiellement des architectures multi-cœurs. Deux propriétés fondamentales sont requises par DSL: la réactivité des sites et l'absence d'interférences entre les scripts exécutés par des sites distincts. Nous présentons également DSLM qui étend DSL en introduisant de la mémoire au niveau de l'orchestration et en définissant un mécanisme pour adapter automatiquement la charge d'exécution aux cœurs disponibles.*

ABSTRACT. *We present DSL, an orchestration language based on the synchronous/reactive model. In DSL, systems are composed of several sites executed asynchronously. Within each site, scripts are run in a synchronous parallel way. Scripts may call functions that are treated in an abstract way: their effect on the memory is not considered, but only their "orchestration" i.e. the organisation of their calls in time and in place (the site where they are called). The mapping of sites onto cores allows one to benefit from multicore architectures. Two properties are required by DSL: reactivity of sites and absence of interferences between scripts run by distinct sites. We also introduce DSLM which adds a memory level to DSL and a way to automatically adapt the execution to get a maximal use of the available cores.*

MOTS-CLÉS : *Programmation synchrone, Langage d'orchestration, Système distribué, Multi-cœurs*

KEYWORDS: *Synchronous programming, Orchestration Languages, Distributed Systems, Multi-cores*

JESA – 45/2011. MSR 2011, pages 77 à 92

1. Introduction

Les recherches extrêmement actives liées au Web posent la question de l'extension des techniques standards de programmation parallèle et distribuée. Diverses propositions d'extension existent, parmi lesquelles on peut citer les *langages d'orchestration*, tel ORC (Misra *et al.*, 2007), et les langages de programmation pour les systèmes diffus, tel HOP (Serrano, 2006). La programmation synchrone (Benveniste *et al.*, 1991) simplifie la programmation parallèle, en comparaison des approches traditionnelles fondées sur l'utilisation exclusive d'un modèle classique de threads (pthreads ou multithreading de Java). Fondamentalement, la simplification résulte d'une sémantique plus claire et plus simple, réduisant le nombre d'interleavings possibles des calculs parallèles. Les langages synchrones introduisent cependant des problèmes spécifiques (en particulier, la possible non-terminaison des instants) et ont de profondes difficultés à traiter la création dynamique (de threads, de composants ou de signaux). De plus, ils ne permettent généralement pas de bénéficier du parallélisme réel fourni par les machines multicœurs.

Dans ce papier, nous adoptons une approche à deux niveaux : le premier niveau est celui d'un nouveau *langage d'orchestration synchrone*, appelé DSL (pour *Dynamic Synchronous Language*). Le second niveau est un langage qui étend DSL. Ce langage, que nous appelons DSLM (pour *DSL with Memory* ; il s'agit d'un nom provisoire), est plus qu'un langage d'orchestration et doit être vu comme un langage de programmation complet. Notre approche repose sur les choix suivants :

– Nous utilisons la variante réactive [1] de l'approche synchrone, capable d'exprimer la création dynamique. Dans cette variante, les "cycles de causalité" sont absents "par construction" grâce à l'interdiction de la réaction immédiate à l'absence des signaux. Cette variante est également utilisée par plusieurs autres langages (par ex. Reactive-C (Boussinot, 1991), SugarCubes (Boussinot *et al.*, 1998), ReactiveML (Mandel *et al.*, 2005)).

– Nous introduisons une notion de *site*, pour traiter dans un même formalisme les aspects synchrones et asynchrones des systèmes. Cette combinaison est inspirée du modèle des FairThreads (Boussinot, 2006) qui mélange threads préemptifs et threads coopératifs. Les sites peuvent être exécutés par des cœurs différents, donnant ainsi au programmeur l'accès à la puissance des architectures multicœurs.

DSL est un langage qui appartient à la famille GALS (*Globally Asynchronous, Locally Synchronous*) (Halbwachs *et al.*, 2002), (Malik *et al.*, 2009) et qui possède une primitive permettant la migration des scripts entre les sites. DSL a une sémantique simple (section 3) décrivant sans ambiguïté l'évolution des systèmes. Nous considérons tout d'abord le niveau d'orchestration de DSL et le modèle de calcul sur lequel il repose.

Modèle d'orchestration. Le modèle de calcul que nous considérons est le suivant : un système est formé de N sites, chacun étant composé de deux niveaux, le niveau de

1. Reactive programming: http://www-sop.inria.fr/indes/rp

l'orchestration et le niveau hôte. Les sites sont complètement autonomes et leur exécution est asynchrone (en utilisant des ressources pouvant être distinctes). Au niveau de l'orchestration, chaque site exécute un script parallèle et ses entrées (*inputs*) sont : (1) les nouveaux scripts déposés par les autres sites, par l'extérieur, ou bien par le niveau hôte. Ces nouveaux scripts sont mis en parallèle avec celui exécuté par le site. Les événements d'input sont simplement des scripts les générant. (2) les booléens et les entiers provenant du niveau hôte et utilisés par les instructions `if` et `repeat`.

De même, au niveau de l'orchestration, les sorties (*outputs*) d'un site sont : (1) les nouveaux scripts envoyés à d'autres sites ; (2) les appels de fonctions appartenant au niveau hôte du site. Les fonctions sont traitées abstraitement. On ne considère par leur effets sur la mémoire, mais uniquement leur "orchestration", c'est-à-dire l'organisation en temps (instant) et en lieu (site) de leurs appels. Les scripts exécutés par un même site se synchronisent par des événements locaux diffusés dans tout le site. Deux propriétés sont requises en DSL : la réactivité des sites et l'absence d'interférences entre les sites (les sites ne partagent ni leurs instants, ni leurs événements, ni leur mémoire). Ce sont ces deux propriétés qui permettent l'exécution des sites par des cœurs distincts tout en préservant la sémantique, et donc de tirer bénéfice des architectures multicœurs.

Il existe une analogie entre site et orchestre de musique : le niveau d'orchestration est celui du chef d'orchestre qui dirige les musiciens appartenant au niveau hôte. Le chef suit une partition musicale (script) et communique avec les musiciens par des ordres (appels de fonctions) et des signaux (événements), tout en écoutant leur musique (des événements pouvant être vus comme les ondes sonores qu'ils produisent). Le chef doit être capable de diriger et d'écouter les musiciens en parallèle. Les instants définissent l'horloge commune, indispensable à tout orchestre. La présence de plusieurs orchestres (sites) jouant de manière asynchrone correspond à ce qui se passe dans certains festivals, dans lesquels plusieurs scènes peuvent être utilisées simultanément (bien sûr, dans de tels cas, l'absence d'interférences est obligatoire pour éviter toute cacophonie : les sites doivent être suffisament éloignés pour ne rien partager). À noter la possibilité d'envoyer une partition d'un orchestre à un autre (migration de scripts).

Variantes de DSL. Dans le cadre de nos travaux, quatre variantes de DSL ont été considérées, différant par le langage sous-jacent (le langage hôte) dans lequel les fonctions et les tâches sont exprimées. Ces variantes et leurs implémentations sont décrites en détail dans (Attar *et al.*, 2011). Dans la première variante, le langage hôte est FunLoft (Boussinot *et al.*, 2008),(Boussinot, 2010). Dans la seconde variante, les fonctions sont définies en ReactiveML (Mandel *et al.*, 2005), une extension réactive de ML. Les fonctions d'ordre supérieur sont utilisables sans restriction dans cette variante de DSL. La troisième variante est bâtie sur les SugarCubes (Boussinot *et al.*, 1998), une plateforme Java pour la programmation réactive dans lequel le modèle objet de Java est disponible. La dernière variante est basée sur Scheme/Bigloo (Serrano *et al.*, 1995) et elle implémente directement les règles sémantiques de la section 3. La distribution des sites au travers du réseau est possible dans les variantes

bâties sur ReactiveML, SugarCubes et Scheme/Bigloo, en utilisant respectivement JoCaml, RMI et HOP.

Extension de DSL. En tant que langage d'orchestration, DSL considère les fonctions et tâches, représentant des services, d'une manière très abstraite (au niveau orchestration, ce sont juste des noms). De plus, la notion de site n'est pas suffisamment flexible pour pouvoir profiter pleinement des architectures multicœurs. Pour ces raisons, que nous préciserons dans la suite, nous proposons une extension de DSL appelée DSLM (*Dynamic Synchronous Language with Memory*). Cette extension rajoute de la mémoire à DSL ainsi qu'un mécanisme d'adaptation automatique permettant d'utiliser au maximum les cœurs disponibles. Pour cela, nous introduisons deux nouvelles notions : les *agents* et les *schedulers synchronisés* décrits dans la section 4.

Le reste du papier est organisé comme suit : DSL est tout d'abord présenté informellement à travers quelques exemples dans la section 2. Sa sémantique formelle est décrite en section 3. DSLM est présenté en section 4. Des travaux proches sont considérés en section 5 et une conclusion est donnée en section 6.

2. Description de DSL

Un programme de DSL est formé de plusieurs sites indépendants, chacun exécutant un script composé de plusieurs composants parallèles. Ajouter un nouveau script à un site signifie le mettre en parallèle avec les composants déjà présents. DSL ne permet pas de définir des fonctions, mais les scripts peuvent appeler les fonctions du langage hôte. Ces fonctions ont uniquement des paramètres ayant des types de base (entier, booléen, chaîne de caractères). Les tâches sont des fonctions spéciales dont l'exécution n'est pas instantanée. En fait, l'exécution d'une tâche ne commence pas immédiatement, mais est toujours décalée à l'instant suivant. De plus, l'exécution d'une tâche peut prendre plusieurs instants, voire ne jamais terminer. Les tâches sont appelées en utilisant un mot-clé spécifique (`launch`). Les scripts "orchestrent" l'exécution des fonctions et des tâches sur les divers sites qui composent un programme.

Scripts. Les instructions élémentaires pour construire les scripts sont les suivantes :

– `nothing` ne fait rien.

– `cooperate` termine pour l'instant présent. L'exécution sera reprise ensuite à l'instant suivant.

– $f\,(v_1, \ldots, v_n)$ appelle la fonction f avec les paramètres $v_1, ..., v_n$. L'exécution démarre immédiatement et est instantanée. Appeler une fonction qui n'existe pas est équivalent à `nothing`.

– `launch` $t\,(v_1, \ldots, v_n)$ lance la tâche t avec les paramètres $v_1, ..., v_n$. L'exécution peut durer plusieurs instants (au moins un) ou même ne jamais terminer. L'appel d'une tâche inexistante est équivalente à `nothing`.

– $s_1; s_2$ exécute les deux scripts s_1 et s_2 en séquence.

– $s_1 \parallel s_2$ exécute les deux scripts s_1 et s_2 en parallèle. La composition parallèle termine lorsque les deux scripts s_1 et s_2 sont terminés.

– `if` exp `then` s_1 `else` s_2 `end` exécute le script déterminé par le résultat de l'évaluation de l'expression booléenne exp.

– `loop` s `end` exécute cycliquement le script s. L'exécution de s est recommencée dès qu'elle termine, sauf si elle se termine instantanément (au même instant). Dans ce cas, la boucle attend le prochain instant pour recommencer s. Il n'y a donc pas de *boucles instantanées*, dont l'exécution cyclerait sans arrêt au cours d'un même instant.

– `repeat` exp `do` s `end` exécute n fois le script s, où n est le résultat de l'évaluation de l'expression entière exp.

– `generate` e génère l'événement e.

– `await` e bloque l'exécution tant que e n'est pas généré. Le blocage se termine dès que e est généré.

– `do` s `watching` e exécute le script s tant que l'événement e n'est pas généré. L'instruction `watching` termine normalement lorsque s termine.

– `drop` s `in` $site$ dépose le script s à destination du site distant $site$. L'exécution continue immédiatement sans attendre l'exécution ou la terminaison de s.

Sites. Les sites sont asynchrones (chaque site peut être exécuté par un thread natif distinct). Par contre, les scripts d'un même site sont exécutés de manière synchrone : ils partagent les mêmes instants et évoluent donc au même rythme. Par l'intermédiaire de l'instruction `drop`, un script peut influencer des sites distants. Remarquons que si rien ne reste à faire après une instruction `drop`, on peut voir celle-ci comme une *migration* vers un site distant. La création des sites n'est pas spécifiée dans le langage et on suppose que pour chaque instruction `drop` s `in` $site$, $site$ existe bien et est effectivement accessible. Une table d'association propre à chaque site associe à chaque nom de site (chaîne de caractères) le site correspondant (désigné par une url).

Input/Output. Les entrées-sorties (*input/output*) des programmes sont modélisées par des dépôts de scripts. La forme la plus simple d'input est le dépôt d'une génération d'événement dans un site. On suppose qu'il existe un site d'output particulier qui interprète les scripts qu'il reçoit.

Événements. Les événements sont présents ou absents au cours des instants et ils ne sont pas partagés entre les sites. Une fois qu'un événement est généré, il reste présent pendant tout l'instant. Les événements sont automatiquement réinitialisés à absent au début de chaque nouvel instant. Les événements sont identifiés par des chaînes de caractères et une table d'association spécifique à chaque site associe un événement à chaque nom. Les événements utilisés par les trois instructions `generate`, `await` et `watching` sont automatiquement créés et introduits dans la table d'association s'ils n'existent pas déjà sur le site d'exécution.

Fonctions et tâches. Il existe une différence fondamentale entre les fonctions et les tâches en DSL. L'exécution d'une fonction est instantanée (elle se termine à l'instant de l'appel) alors que celle d'une tâche peut prendre plusieurs instants, voire

même ne jamais se terminer. Les fonctions et tâches sont désignées par des chaînes de caractères et une table d'association propre à chaque site permet d'utiliser les fonctions et tâches définies sur le site.

Propriétés fondamentales. Deux propriétés fondamentales sont requises par DSL : (1) aucun site ne peut être empêché de passer à l'instant suivant (propriété de *réactivité*). Cela signifie qu'une fonction ou une tâche exécutée sur un site ne doit jamais utiliser toute la puissance de calcul disponible sur le site. (2) aucun conflit d'accès (*data-race*) entre scripts, fonctions et tâches ne doit jamais survenir (propriété de *non-interférence*). Dans la variante FunLoft de DSL, ces deux propriétés fondamentales sont automatiquement vérifiées par le compilateur. Dans les autres variantes, leur vérification est à la charge du programmeur.

Il existe deux manières d'exécuter les tâches : dans la première, la tâche est exécutée par le site sur lequel elle a été lancée (elle est *liée* au site) ; dans la seconde, elle exécutée par un thread natif dédié (la tâche est *déliée*). Un mécanisme similaire est utilisé par le formalisme CPC (Kerneis *et al.*, 2010).

Exemples. Nous présentons deux exemples d'utilisation de DSL. Le premier est un jeu très simple (appelé *Simon*) qui illustre les aspects d'orchestration et de réactivité de DSL. Le second exemple (que nous appellerons *Consume*) met en lumière les aspects multi-site et parallélisme réel du langage.

Simon. Dans le jeu Simon, qui a été populaire dans les années 80, le joueur doit reproduire une suite de couleurs (ou de sons) générée aléatoirement par la machine. En cas de succès, une nouvelle couleur est ajoutée à la suite, et ainsi de suite jusqu'à ce que le joueur commette une erreur. On considère un système mono-site. La première ligne du code qui suit est formée d'initialisations (fenêtre, suite, etc). Puis, on entre dans une boucle infinie (`loop`) qui commence par montrer la séquence à reproduire (l'argument est un délai). Une tâche d'observation (`observer`) est ensuite lancée en parallèle avec deux autres scripts : le premier compte les réponses correctes, tandis que le second réagit aux réponses incorrectes. La tâche `observer` analyse les réponses de l'utilisateur et génère les événements `OK` ou `NOK` suivant qu'elles sont correctes ou non. En cas d'erreur, le corps de la boucle est recommencé (`watching` error), tandis que la taille de la séquence est augmentée lorsque toutes les bonnes réponses ont été obtenues (`watching` done). Le code est le suivant :

```
launch sleep(1000000); init_windows(); launch new_frame();
loop
  do
    launch show_sequence(300000);
    (   launch observer()
     || (do
            repeat get_length() do await OK; cooperate end;
            generate done;
          watching done; launch success(); launch increment_frame();)
     || (await NOK; launch failure();
          launch new_frame(); generate error;)  )
  watching error; launch sleep(100000);
end
```

Consume. Nous considérons un système composé de 3 sites, `site1`, `site2` et `site3`, et d'un script initialement exécuté par `site1`. Ce script est composé de deux parties, exécutées sur `site2` et `site3`. Chacune des parties appelle la fonction `consume` (utilisant fortement le CPU, suivant la valeur de son paramètre) et ensuite envoie sur le site initial un script signalant sa fin. Les deux événements générés par les deux appels de `consume` sont attendus en parallèle. Le code est le suivant :

```
repeat 1000 do
  (  drop
        print("0"); consume(10000000); drop generate done0 in site1
     in site2
  || drop
        print("1"); consume(10000000); drop generate done1 in site1
     in site3
  || await done0 || await done1  );
   cooperate
end
```

Remarquons la duplication de code (par exemple, les deux appels à `consume`). Le langage ne donne actuellement aucun moyen de partager ou de paramétriser les scripts. De ce point de vue, le code DSL est donc plutôt destiné à être généré à partir d'un langage ayant une expressivité supérieure ; la définition d'un tel langage ne fait pas partie des objectifs de ce papier. Les deux appels de `consume` peuvent être exécutés en parallélisme réel (par exemple, sur une machine *dual-core*). On suppose qu'il n'y a aucune interférence entre ces exécutions (provenant, par exemple, de l'utilisation d'un compteur partagé). Cette propriété est vérifiée statiquement dans la variante de DSL bâtie sur FunLoft, tandis que sa validité est de la responsabilité du programmeur dans les autres variantes. Cet exemple est discuté plus à fond dans (Attar *et al.*, 2011), qui contient des mesures de temps d'exécution effectuées dans les diverses variantes.

3. Sémantique de DSL

La sémantique que nous donnons à DSL s'exprime sous la forme de règles de réécriture. Il s'agit d'une sémantique dite "à grands-pas" (*big-step*) : une réécriture d'un terme représente l'exécution globale du terme durant un instant (contrairement à une sémantique "petits-pas" (*small-step*) dans laquelle plusieurs réécritures sont nécessaires pour exprimer un instant).

Expressions. Les expressions sont soit des valeurs de base (de type entier, booléen, ou chaîne de caractères), soit des appels de fonction de la forme $f\,(v_1, \ldots, v_n)$ où les v_i sont des valeurs de base. Nous adoptons la notation suivante : on écrit $f\,(v_1, \ldots, v_n)\Uparrow$ si aucune fonction de nom f n'est définie ou bien si l'appel n'est pas bien typé ; dans ce cas, on parle d'un *appel incorrect*. Lorsque ce n'est pas le cas, on écrira $f\,(v_1, \ldots, v_n)\Downarrow$.

L'évaluation d'une valeur de base retourne la valeur elle-même. Il y a deux cas pour l'évaluation de $f\,(v_1, \ldots, v_n)$:

– Si $f\,(v_1, \ldots, v_n)\Downarrow$, l'évaluation retourne la valeur de **f** appliquée à la liste des valeurs v_i, où **f** est la fonction effectivement associée à f.

– Si $f\,(v_1, \ldots, v_n)\Uparrow$, alors la valeur retournée est la valeur par défaut du type de base attendu (0 pour les entiers, `false` pour les booléens, et la chaîne vide `""` pour les chaînes de caractères).

L'évaluation de l'expression exp retournant une valeur v est notée $exp \rightsquigarrow v$.

Comme pour les fonctions, nous écrivons $t\,(v_1, \ldots, v_n)\Uparrow$ si la tâche t n'est pas définie, ou bien si l'appel n'est pas bien typé, et nous écrivons $t\,(v_1, \ldots, v_n)\Downarrow$ sinon.

Scripts. Le format général de la réécriture d'un script est le suivant :

$$P \vdash s \xrightarrow{b} s', G, D$$

– P est l'ensemble des événements présents ; les événements qui n'appartiennent pas à P sont absents.

– s est le script qui est réécrit.

– s' est le résultat de la réécriture (le *résidu*, ce qui reste à exécuter à l'instant suivant).

– G est l'ensemble des événements générés lors de la réécriture de s.

– D est le multi-ensemble des dépôts de script, à destination des sites distants. Les éléments de D sont de la forme $site{\downarrow}u$, où $site$ est un nom de site et u est un script. L'union des multi-ensembles est notée $\uplus$.

– b est un booléen, qui vaut vrai (tt) si s' est complètement terminé et faux (ff) sinon. La conjonction booléenne est notée $\wedge$.

La figure 1 contient les règles de réécriture définissant la sémantique de DSL. Nous ne commenterons pas ces règles en détail et ne mentionnerons que quelques points. L'exécution et l'évaluation d'un appel de fonction sont équivalentes (règle *call*) ; la valeur retournée est ignorée et l'appel est uniquement exécuté pour ses effets de bord. Un appel incorrect est sans effet.

Trois points sont à noter concernant les tâches : (1) la règle $task_1$ établit qu'un appel incorrect est équivalent à `nothing` ; (2) dans la règle $task_2$, e est un nouvel événement [2] qui signale la terminaison de la tâche lancée. Cet événement est automatiquement généré par le système lorsque l'appel de t est complètement terminé ; (3) en cas de préemption effective, c'est-à-dire lorsque la règle $watch_2$ s'applique, la tâche n'est pas lancée et l'attente de la terminaison est abandonnée.

Dans une instruction `repeat`, l'expression exp est évaluée lorsque la règle s'applique, à l'exécution et non à la compilation. Si exp est un appel incorrect, on considère que n vaut 0 et l'instruction est alors égale à `nothing` [3].

2. On suppose qu'il existe un mécanisme capable de toujours créer de nouveaux événements.
3. Une séquence de $n \leq 0$ éléments est par définition égale à `nothing`.

$$P \vdash \texttt{nothing} \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset \; (nothing) \quad P \vdash \texttt{cooperate} \xrightarrow{ff} \texttt{nothing}, \emptyset, \emptyset \; (cooperate)$$

$$P \vdash \texttt{drop}\ s\ \texttt{in}\ site \xrightarrow{tt} \texttt{nothing}, \emptyset, \{site{\downarrow}s\} \; (drop)$$

$$\frac{P \vdash s_1 \xrightarrow{ff} s_1', G, D}{P \vdash s_1; s_2 \xrightarrow{ff} s_1'; s_2, G, D}\,(seq_1) \qquad \frac{P \vdash s_1 \xrightarrow{tt} s_1', G_1, D_1 \;\; P \vdash s_2 \xrightarrow{b} s_2', G_2, D_2}{P \vdash s_1; s_2 \xrightarrow{b} s_2', G_1 \cup G_2, D_1 \uplus D_2}\,(seq_2)$$

$$\frac{P \vdash s_1 \xrightarrow{b_1} s_1', G_1, D_1 \;\; P \vdash s_2 \xrightarrow{b_2} s_2', G_2, D_2}{P \vdash s_1 \parallel s_2 \xrightarrow{b_1 \wedge b_2} s_1' \parallel s_2', G_1 \cup G_2, D_1 \uplus D_2}\,(par) \qquad \frac{P \vdash s \parallel \texttt{cooperate} \xrightarrow{ff} s', G, D}{P \vdash \texttt{loop}\ s\ \texttt{end} \xrightarrow{ff} s'; \texttt{loop}\ s\ \texttt{end}, G, D}\,(loop)$$

$$P \vdash \texttt{generate}\ e \xrightarrow{tt} \texttt{nothing}, \{e\}, \emptyset \; (generate)$$

$$\frac{e \in P}{P \vdash \texttt{await}\ e \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset}\,(await_1) \qquad \frac{e \notin P}{P \vdash \texttt{await}\ e \xrightarrow{ff} \texttt{await}\ e, \emptyset, \emptyset}\,(await_2)$$

$$\frac{P \vdash s \xrightarrow{tt} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ e \xrightarrow{tt} \texttt{nothing}, G, D}\,(watch_1) \qquad \frac{e \in P \;\; P \vdash s \xrightarrow{ff} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ e \xrightarrow{ff} \texttt{nothing}, G, D}\,(watch_2)$$

$$\frac{e \notin P \;\; P \vdash s \xrightarrow{ff} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ e \xrightarrow{ff} \texttt{do}\ s'\ \texttt{watching}\ e, G, D}\,(watch_3)$$

$$\frac{f\,(v_1, \ldots, v_n) \rightsquigarrow v}{P \vdash f\,(v_1, \ldots, v_n) \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset}\,(call)$$

$$\frac{t\,(v_1, \ldots, v_n){\Uparrow}}{P \vdash \texttt{launch}\ t\,(v_1, \ldots, v_n) \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset}\,(task_1) \qquad \frac{t\,(v_1, \ldots, v_n){\Downarrow}}{P \vdash \texttt{launch}\ t\,(v_1, \ldots, v_n) \xrightarrow{ff} \texttt{await}\ e, \emptyset, \emptyset}\,(task_2)$$

$$\frac{exp \rightsquigarrow n \;\; P \vdash \overbrace{s; \ldots; s}^{n\ times} \xrightarrow{b} s', G, D}{P \vdash \texttt{repeat}\ exp\ \texttt{do}\ s\ \texttt{end} \xrightarrow{b} s', G, D}\,(repeat)$$

$$\frac{exp \rightsquigarrow tt \;\; P \vdash s_1 \xrightarrow{b} s_1', G, D}{P \vdash \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \xrightarrow{b} s_1', G, D}\,(if_1) \qquad \frac{exp \rightsquigarrow ff \;\; P \vdash s_2 \xrightarrow{b} s_2', G, D}{P \vdash \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \xrightarrow{b} s_2', G, D}\,(if_2)$$

Figure1. *Sémantique des scripts de DSL*

Plus petit point-fixe. L'exécution des scripts est déterministe : si on a deux réécritures $P \vdash s \xrightarrow{b_1} s_1, G_1, D_1$ et $P \vdash s \xrightarrow{b_2} s_2, G_2, D_2$, alors $s_1 = s_2, G_1 = G_2, D_1 = D_2$, et $b_1 = b_2$.

Soit s un script ; grâce à la propriété de déterminisme, nous pouvons définir la fonction f_s qui, étant donné un ensemble P d'événements présents, retourne l'ensemble G des événements générés par la réécriture de s:

$$f_s(P) = G \text{ où } \; P \vdash t \xrightarrow{b} s', G, D$$

La fonction f_s a deux caractéristiques principales : elle est totale et croissante. Elle est totale parce que, pour chaque script et chaque ensemble d'événements présents, il existe une (unique) réécriture :

$$\forall s, P, \exists s', G, D, b \;\; P \vdash s \xrightarrow{b} s', G, D$$

La fonction f_s est croissante (pour l'inclusion ensembliste) car on a :

$$\text{si } P_1 \subseteq P_2 \text{ alors } f_s(P_1) \subseteq f_s(P_2)$$

Grâce au théorème de Kleene, nous savons donc que la fonction f_s possède un plus petit point-fixe, noté μf_s, vérifiant :

$$f_s(\mu f_s) = \mu f_s$$
$$\mu f_s \vdash s \xrightarrow{b} s', \mu f_s, D$$
$$\forall Q, f_s(Q) = Q \text{ implique } \mu f_s \subseteq Q$$

Le plus petit point-fixe μf_s est la limite supérieure de la suite d'approximations $X_0, X_1, \ldots$ définies par $X_0 = \emptyset$ et $X_{n+1} = f_s(X_n)$, noté $\mu f_s = \bigcup f_s^n(\emptyset)$. Pour finir, nous écrivons simplement $s \Rightarrow s', D$ au lieu de $\mu f_s \vdash s \xrightarrow{b} s', \mu f_s, D$ lorsque seule l'existence du point fixe (et non sa valeur) est significative.

Sites. Un site est un couple $(site, s)$ formé d'un nom de site $site$ et d'un script s ; il est noté $site : s$. Un programme est un multi-ensemble (fini) de sites et de dépôts de script. Un programme est donc un multi-ensemble S dont les éléments sont soit des sites de la forme $site_i : s_i$ soit des dépôts de script de la forme $site_i \downarrow s_i$. On suppose qu'il y a au moins un site et que tous les sites ont des noms distincts :

$$\forall site_i : s_i, site_j : s_j \in S, i \neq j \Rightarrow site_i \neq site_j$$

Remarquons que le même dépôt de script peut apparaître plusieurs fois dans un programme (un programme est un multi-ensemble), comme dans :

$$\{site : \texttt{nothing}, site \downarrow f(), site \downarrow f()\}$$

L'exécution d'un programme S_0 est une séquence de réécritures de la forme :

$$S_0 \mapsto S_1 \mapsto S_2 \mapsto ...$$

où la flèche $\mapsto$ est définie par les règles de la figure 2.

Exécution des sites. Les dépôts de script sont ajoutés au programme par la règle $site_1$ (figure 2). Le choix du site à exécuter est arbitraire (sites asynchrones). Un même site peut être exécuté plusieurs fois de suite alors que d'autres ne le sont pas (il

$$\frac{s \Rightarrow s', D}{\{..., site : s, ...\} \mapsto \{..., site : s', ...\} \uplus D} \ (site_1)$$

$$\{..., site : s, site{\downarrow}u, ...\} \mapsto \{..., site : s \parallel u, ...\} \ \ (site_2)$$

$$S \mapsto S \uplus \{site{\downarrow}s\} \ \ (site_3)$$

Figure 2. *Sémantique des sites de DSL*

n'existe en particulier aucune assurance d'absence de famine d'un site qui ne serait jamais choisi). Les scripts déposés attendent d'être absorbés par la règle $site_2$. Dans la définition de $\Rightarrow$, notons que le plus petit point-fixe n'est pas explicitement construit : la sémantique n'est pas explicite sur ce point car elle ne précise pas *comment* il peut être calculé.

La règle $site_2$ décrit l'absorption par le site $site$ d'un script déposé u : le script déposé est simplement mis en parallèle avec le script s déjà présent dans $site$.

Les inputs d'un programme sont simplement des depôts de scripts. La règle $site_3$ modélise l'input d'un script s à destination du site particulier $site$ du programme S.

4. Extensions de DSL

DSL ne permet pas d'utiliser les ressources de manière optimale. Considérons, par exemple, un système formé de deux sites. L'exécution sur une machine possédant deux cœurs pourra utiliser ceux-ci de manière optimale, en attribuant un cœur à chacun des sites. En revanche, l'exécution ne sera plus optimale sur une machine quadri-cœurs. Notons que, dans ce cas, il est nécessaire pour l'optimalité d'assigner aux cœurs des sous-systèmes de taille plus petite que celle des sites. Parce que nous considérons DSL comme un langage d'orchestration, nous avons fait le choix de ne pas considérer la mémoire au niveau de l'orchestration et de la confiner au niveau hôte. Ce choix qui semble raisonnable dans certains contextes (par exemple, celui des services Web) est certainement trop restrictif dans le cas général. Le besoin d'améliorer l'utilisation des cœurs et celui de disposer de mémoire au niveau de l'orchestration nous ont conduits à considérer une extension de DSL, appelée DSLM (*Dynamic Synchronous Language with Memory*). La conception de DSLM est un travail en cours et nous ne discuterons ici que de ses principales caractéristiques. Remarquons cependant que, de par la présence de mémoire, DSLM ne peut plus être considéré comme un langage d'orchestration (du moins dans le sens où nous avons utilisé ce terme pour DSL).

Introduction de la mémoire. L'ajout de la mémoire, pour rendre la programmation plus flexible, introduit les problèmes standards d'interférence (par exemple, les *data-races*). Pour éviter ces problèmes, nous contrôlons en DSLM la mémoire de deux

manières : premièrement en introduisant une notion d'*agent* qui possède sa propre mémoire ; deuxièmement, en restreignant aux threads coopératifs les accès concurrents à la mémoire partagée.

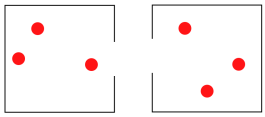
Agent. Un agent est une entité contenant un script (en fait, un script parallèle) et sa propre mémoire. Les agents sont toujours hébergés par un site, pouvant changer au cours du temps (les agents peuvent migrer). Pour préserver l'isolation de la mémoire, le partage de mémoire entre agents (même situés sur un même site) est interdit. La mémoire d'un agent ne peut être utilisée que par les scripts qu'il contient. De plus, les scripts d'un agent sont toujours exécutés de manière coopérative, ce qui interdit les data-races entre eux. Ces interdictions résultent d'une analyse statique utilisant un système de types. Ainsi, la propriété de non-interférence, supposée valide au niveau hôte, est étendue au langage complet. Mentionnons que nos agents sont proches des *agents mobiles* de *ULM* (Boudol, 2004), avec cependant une structure plus simple.

Sites. En DSLM, les sites sont asynchrones (non nécessairement exécutés au même rythme). Chaque site contient (1) un ensemble d'agents, chaque agent contenant lui-même un ensemble de scripts, et (2) un ensemble d'événements partagés entre les agents du site (pour accéder à l'environnement d'événements, un agent doit être sur le site).

Extension et contraction dynamique des sites. Un site est formé d'un ensemble de *schedulers synchronisés*, dont le nombre varie dynamiquement au cours de l'exécution, et qui sont exécutés par les cœurs de l'architecture d'exécution. En adaptant dynamiquement les schedulers synchronisés au nombre d'agents hébergés par les sites, il devient possible d'adapter l'utilisation des cœurs à la charge d'exécution des sites. Les schedulers synchronisés, empruntés à FunLoft, partagent les mêmes instants et les mêmes événements, tout en pouvant être exécutés en vrai parallélisme. Au niveau de l'implémentation, chaque scheduler synchronisé est exécuté par un thread natif dédié, et il existe une barrière de synchronisation à la fin de chaque instant pour tous les schedulers d'un même site. Un agent exécuté par un scheduler synchronisé peut donc de manière transparente migrer sur un autre scheduler du même site, sans risque de data-races. Notons que cette propriété repose fortement sur l'absence de partage de mémoire entre agents. Dans l'approche que nous proposons, un site peut dynamiquement augmenter son nombre de schedulers (1) en choisissant un cœur libre, c'est-à-dire inutilisé par le programme en train de s'exécuter ; (2) en associant à ce cœur un nouveau scheduler (thread natif) ; (3) en synchronisant ce nouveau scheduler avec les autres schedulers du site. Inversement, lorsqu'un scheduler devient inutile, il peut être supprimé ; dans ce cas, on note que le cœur qui l'exécutait devient libre, de telle manière qu'il puisse être réutilisé par les autres sites. À l'aide de ce mécanisme, les programmes peuvent s'adapter dynamiquement, non seulement aux diverses architectures, mais également à des charges d'exécution (agents) différentes.

Auto-migration des agents. En plus du mécanisme de migration automatique des agents entre les schedulers synchronisés d'un même site (migration *objective*), nous introduisons la possibilité pour un agent de migrer de manière autonome vers d'autres sites (migration *subjective*).

Exemple. L'exemple suivant illustre la notion d'agent et la migration de ceux-ci entre divers sites. Considérons deux boîtes contenant des balles se collisionnant et rebondissant sur leurs bords. Chaque boîte contient une porte qui la connecte à l'autre : une balle atteignant la porte passe immédiatement dans l'autre boîte. Ce système est représenté sur l'image suivante :



Il est naturel de représenter chaque balle par un agent dont la mémoire contient l'état de la balle (coordonnées et vitesse). L'agent contient plusieurs scripts pour animer la balle (par exemple, pour lui imprimer un mouvement inertiel, pour la faire rebondir sur les bords, etc.), pour l'afficher, et pour lui permettre de communiquer son état. La mémoire de l'agent est partagée par ces scripts. Notons que les scripts de l'agent étant exécutés de manière coopérative par le même scheduler, aucune data-race ne peut advenir à ce niveau. La migration (subjective) des balles est implémentée simplement avec l'instruction `drop`. Remarquons que le nombre de cœurs utilisés par une boîte devrait s'adapter automatiquement au nombre de balles qu'elle simule. Remarquons également que, grâce à la notion d'agent de DSLM, nous avons l'assurance qu'aucune data-race ne peut survenir, qu'elle soit liée à la migration automatique intra-site, ou bien à la migration subjective inter-sites. La mémoire d'un agent est directement accessible et partagée par les scripts de l'agent, et un système de types assure qu'elle n'est jamais partagée par les scripts d'un autre agent.

5. Travaux proches

Esterel. DSL diffère d'Esterel (Berry *et al.*, 1992) par deux aspects principaux. Premièrement, il permet la dynamicité. L'idée centrale la rendant possible est l'interdiction de la réaction instantanée à l'absence des signaux (appelés événements en DSL) ; il s'agit de l'idée de base de l'approche réactive. Deuxièmement, DSL introduit l'asynchronie par la notion de site. L'asynchronie étant absente d'Esterel[4], les questions de protection de la mémoire contre des accès concurrents asynchrones ne se posent pas dans ce langage.

Scripts Reactifs. Les Scripts réactifs (Boussinot *et al.*, 1996) sont très proches de notre proposition. Ils permettent la migration à l'aide de RPC et sont définis indépendamment des actions atomiques (appels de fonction). Les principales questions abordées en DSL ne sont, cependant, pas considérées par les Scripts réactifs : terminaison des instants et absence d'interférences. Les Scripts réactifs introduisent une

4. Excepté dans les premières versions du langage, dans lesquelles une notion de tâche asynchrone (`exec`) était définie.

notion d'objet qu'il serait sûrement intéressant d'introduire en DSL aussi (cette notion repose sur une primitive `control` actuellement absente de DSL).

FunLoft. DSL est fortement inspiré de FunLoft (Boussinot *et al.*, 2008), qui est fondé sur le travail de (Dabrowski, 2007). FunLoft utilise des threads alors que DSL utilise un opérateur explicite de parallélisme. En fait, l'implémentation de DSL en FunLoft est une illustration de l'implémentation d'un opérateur de parallélisme avec des threads. Le compilateur FunLoft vérifie que l'utilisation des ressources est bornée ; ce contrôle est débranché pour implémenter DSL, dans lequel l'aspect du contrôle de ressource est totalement absent.

SugarCubes. Un programme DSL correct doit être réactif (la séquence des instants est infinie) et sans data-races. Ces deux propriétés sont également demandées en SugarCubes (Boussinot *et al.*, 1998), mais elles ne sont pas vérifiées par les implementations. En SugarCubes, les boucles instantanées sont détectées mais les appels de fonction ne terminant pas ne le sont pas. Cependant, SugarCubes fournit un ensemble d'instructions beaucoup plus riche, en particulier pour la programmation distribuée ; de plus, il propose une couche objet, absente de DSL.

ReactiveML. Le langage ReactiveML (Mandel *et al.*, 2005) offre une grande flexibilité en permettant d'exécuter des scripts qui sont des programmes ReactiveML standards, compilés "à la volée". Construit au dessus de ML, ReactiveML est sûr (pas de possibilité de crash lors de l'exécution). La terminaison des instants n'est cependant pas vérifiée par le compilateur. De plus, ReactiveML, comme ML, n'est pas actuellement adapté aux architectures multi-cœurs.

Sπ-Calculus. DSL est relié à l'approche théorique du Sπ-Calcul (Amadio, 2007), les deux étant fondés sur des modèles synchrones proches. En Sπ-Calcul, l'accent est mis sur le contrôle des ressources : les ressources sont prouvées être *polynomialement bornées*. Comme mentionné plus haut, DSL ne considère pas cet aspect.

Cooperative multithreading. Des techniques d'analyse statique dans le contexte du multithreading sont étudiées dans plusieurs travaux récents (Boudol, 2007), (Yi *et al.*, 2010) qui ne considèrent cependant pas le cas du parallélisme synchrone.

ORC. ORC (Misra *et al.*, 2007) est un langage d'orchestration pour le Web. Au niveau de l'orchestration, les objectifs de ORC et ceux de DSL sont très proches et il serait sûrement intéressant de comparer en détail ces deux langages.

Lucid Synchrone. Lucid Synchrone (Caspi *et al.*, 1995) est un langage synchrone d'ordre supérieur basé sur le langage ML. Comme avec l'approche réactive, la création dynamique de composants (dataflow) parallèles est possible en Lucid Synchrone. Cependant, contrairement à notre modèle, Lucid Synchrone ne se préoccupe pas de l'utilisation des cœurs.

Reactor. Reactor (Field *et al.*, 2009) est un langage qui mélange la programmation DataLog avec le monde synchrone. Reactor et DSL sont fondés sur un même modèle GALS. Comme en DSLM, il existe une façon de synchroniser une collection de *reactors*, qui sont donc de ce point de vue très proches des schedulers synchronisés.

6. Conclusion

Nous avons présenté une approche de programmation parallèle dynamique, fondée sur le modèle GALS, dans sa variante réactive-synchrone. Le parallélisme synchrone est plus simple que le parallélisme asynchrone des approches traditionnelles, fondées sur l'utilisation exclusive de threads préemptifs (associés à des verrous).

Dans le langage d'orchestration synchrone DSL, les sites sont exécutés en vrai parallélisme. La spécificité de DSL est de fournir une primitive syntaxique minimale (`drop`) pour la migration de scripts entre les sites. Dans la variante FunLoft de DSL, les propriétés de base de DSL (réactivité et absence d'interférences) sont vérifiées statiquement par le compilateur. L'extension DSLM de DSL a pour objectif de permettre d'adapter dynamiquement l'utilisation des ressources de calcul (cœurs) à la charge d'exécution, tout en préservant les propriétés de non-interférence et de réactivité.

Nous envisageons les pistes de travail suivantes : (1) implémentation d'un protocole de communication pour interconnecter plusieurs systèmes codés dans diverses variantes de DSL ; (2) extension de la sémantique de DSL à DSLM. Notons que la présence de la mémoire rend une approche "petits-pas" inévitable ; (3) définir un système de types pour DSLM, pour contrôler l'accès aux mémoires des agents ; (4) implémenter DSLM, en particulier le mécanisme pour maximiser l'utilisation des cœurs.

Remerciements.

Nous remercions l'ANR, projet PARTOUT 08-EMER-010.

7. Bibliographie

Amadio R. M., « A Synchronous pi-Calculus », *Journal of Information and Computation*, vol. 205, n° 9, p. 1470-1490, 2007.

Attar P., Boussinot F., Mandel L., Susini J.-F., « Proposal for a Dynamic Synchronous Language », *http://hal.archives-ouvertes.fr/hal-00590420*, 2011.

Benveniste A., Berry G., « The Synchronous Approach to Reactive and Real-Time System », *Proceedings of the IEEE*, vol. 79, n° 9, p. 1270-1282, 1991.

Berry G., Gonthier G., « The Esterel Synchronous Programming Language: Design, Semantics, Implementation », *Science of Computer Programming*, vol. 19, n° 2, p. 87-152, 1992.

Boudol G., « ULM, a core programming model for global computing », *Lecture Notes in Computer Science*, vol. 2986, p. 234-248, 2004.

Boudol G., « Fair Cooperative Multithreading », *CONCUR 2007*, p. 272-286, 2007.

Boussinot F., « Reactive C: An Extension of C to Program Reactive Systems », *Software Practice and Experience*, vol. 21, n° 4, p. 401-428, april, 1991.

Boussinot F., « FairThreads: Mixing Cooperative and Preemptive Threads in C », *Concurrency and Computation: Practice and Experience*, vol. 18, p. 445-469, 2006.

Boussinot F., *Safe Reactive Programming. The FunLoft Language*, Lambert Academic Pub., 2010.

Boussinot F., Dabrowski F., « Safe Reactive Programming: the FunLoft Proposal », *Proc. of MULTIPROG – First Workshop on Programmability Issues for Multi-Core Computers*, Göteborg, January, 2008.

Boussinot F., Hazard L., « Reactive Scripts », *Proc. International Conference on Real-Time Computing Systems and Applications, RTCSA'96*, Seoul, p. 267-274, October, 1996.

Boussinot F., Susini J.-F., « The SugarCubes Tool Box - A Reactive Java Framework », *Software Practice and Experience*, vol. 28, n° 14, p. 1531-1550, december, 1998.

Caspi P., Pouzet M., « A Functional Extension to Lustre », *in* M. A. Orgun, E. A. Ashcroft (eds), *International Symposium on Languages for Intentional Programming*, World Scientific, Sydney, Australia, May, 1995.

Dabrowski F., *Programmation réactive synchrone - Langage et contrôle des ressources*, PhD thesis, Université Paris 7, 2007.

Field J., Marinescu M.-C., Stefansen C., « Reactors: A data-oriented synchronous / asynchronous programming model for distributed applications », *Theoretical Computer Science*, vol. 410, n° 2-3, p. 168 - 201, 2009.

Halbwachs N., Baghdadi S., « Synchronous Modelling of Asynchronous Systems », *in* A. Sangiovanni-Vincentelli, J. Sifakis (eds), *Embedded Software*, vol. 2491 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, p. 240-251, 2002.

Kerneis G., Chroboczek J., « CPC: programming with a massive number of lightweight threads », 2010, http://hal.archives-ouvertes.fr/hal-00563369. To appear in PLACES'11.

Malik A., Salcic Z., Girault A., Walker A., Lee S.-C., « A Customizable Multiprocessor for Globally Asynchronous Locally Synchronous Execution », *International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES'09*, Madrid, Spain, p. 120-129, September, 2009.

Mandel L., Pouzet M., « ReactiveML, a Reactive Extension to ML », *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July, 2005.

Misra J., Cook W., « Computation Orchestration », *Software and Systems Modeling*, vol. 6, p. 83-110, 2007.

Serrano M., Hop, a Language for Programming the Web 2.0, Technical report, INRIA Sophia Antipolis, mai, 2006. http://hop.inria.fr.

Serrano M., Weis P., « Bigloo: a portable and optimizing compiler for strict functional languages », *in* A. Mycroft (ed.), *Static Analysis*, vol. 983 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, p. 366-381, 1995.

Yi J., Flanagan C., « Effects for Cooperable and Serializable Threads », *ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI 2010*, 2010.