Les SugarCubes v5

Rapport de Recherche,
livrable du projet ANR PARTOUT: Tâche T2.4
ANR-08-EMER-010

version 3

J-F Susini
CNAM-CEDRIC
jean-ferdinand.susini@cnam.fr

27 janvier 2013

Résumé

Dans ce document, nous présentons le langage de programmation réactive **SugarCubes** v5 en insistant particulièrement sur le modèle de calcul. Nous donnons un système de règles de réécriture, définissant la sémantique opérationnelle des instructions. À l'aide d'un outil de preuve de terminaison sur des systèmes de réécriture, nous construisons la preuve (en vue de sa certification en Coq) de la terminaison des instants (preuve de réactivité) et la preuve de confluence (preuve de déterminisme des exécutions réactives) du système de règles. Ces propriétés sont garanties sous l'hypothèse de terminaison des exécutions de code atomique.

1 Introduction

Le modèle de programmation réactif/synchrone proposé par F. BOUSSINOT [1][2][3] au début des années 90 a été étudié depuis maintenant plus d'une vingtaine d'années. Mais tandis que ce modèle de programmation s'appuie principalement sur l'utilisation de la composition parallèle de programmes au niveau de l'expression des systèmes réactifs, les implantations classiques de ce modèle de calcul produisent un code efficace mais essentiellement séquentiel.

Les **SugarCubes**[4] sont un ensemble de classes Java permettant d'implanter des systèmes réactifs basés sur le modèle de calcul "à la" BOUSSINOT. Dans ce modèle, les composants parallèles fonctionnent au même rythme. L'exécution globale du système est découpée en une succession potentiellement infinie d'instants. Au cours de chaque instant d'exécution, tous les composants parallèles du système réagissent pour l'instant en cours. Si bien qu'à la fin d'un instant tous les composants se retrouvent naturellement synchronisés. Un intérêt de cette approche provient du fait que l'on peut donner un sens précis à la notion de diffusion d'information, d'absence d'information, de simultanéité d'information et que la programmation parallèle est placée au cœur du modèle de calcul.

S'inspirant de la notion de signaux utilisé dans le langage synchrone Esterel[5], la communication entre composants réactifs en **SugarCubes** est réalisée au travers d'événements diffusés instantanément à l'ensemble du système. Les événements sont identifiés par des noms. Un événement est une valeur partagée entre tous les composants parallèles. Cette valeur est booléenne et unique à chaque instant : un événement est présent (son nom appartient à l'environnement d'exécution) ou absent. L'absence d'un événement dans l'approche réactive "à la" BOUSSINOT ne peut être décidée qu'à la fin de l'instant courant et la réaction à l'absence d'un événement est donc repoussée à l'instant suivant pour éliminer les problèmes de causalité de l'approche synchrone dont cette approche s'inspire[6].

Les **SugarCubes** v5 proposent un langage de programmation qui à l'instar des versions antérieures s'appuie de façon importante sur l'utilisation de la construction parallèle au niveau de la spécification des programmes mais dont l'implantation concrète utilise le parallélisme sous-jacent offert par les architectures matérielles actuelles (SMT, architecture multicores,...). Ce framework a été développé dans le cadre du projet **PARTOUT** financé par l'**ANR** (ANR-08-EMER-010). Les **SugarCubes** v5 sont la principale contribution du laboratoire **CEDRIC** à ce projet.

La sémantique et le modèle de programmation ont été repensés pour utiliser le parallélisme physique des ressources de calcul au cœur de l'implantation d'une horloge réactive/synchrone. Jusqu'aux versions 4.x, l'horloge globale rythmant l'exécution d'un système était implanté par un code purement séquentiel. Le parallélisme physique était pris en compte à travers un modèle de type GALS (Globaly Asynchronous Locally Synchronous). Dans cette dernière approche, on considère plusieurs systèmes synchrones indépendants les uns des autres, pouvant communiquer entre eux par le biais de mécanismes de communication asynchrone ou synchrone (par exemple en utilisant Java RMI depuis la version 2 des **SugarCubes**, ou en gérant des files d'attente de messages). Des expérimentations ont également été menées pour ajouter des mécanismes de synchronisation distribués [7] ou des mécanismes plus dynamiques autorisant l'attachement temporaire à un synchroniseur distribué [8]. Les **SugarCubes** v4 ont permis d'explorer des versions intermédiaires de ces mécanismes en introduisant des primitives de communication inter horloges permettant de retrouver certaines propriétés synchrones sur des communications asynchrones[9]. Cependant, la contrepartie inévitable derrière ces notions est la nécessaire introduction de la notion de tolérance aux pannes dans le modèle de calcul global rendant les choses très complexes.

Les **SugarCubes** v5 reprennent un certain nombre d'éléments des versions antérieures. Ainsi, la plupart des primitives de programmation ont été reprises dans ce nouveau framework ainsi que les principes d'implantation (une classe par instruction, une méthode d'activation appelée récursivement, construction arborescente des programmes. . .), mais quelques transformations majeures ont également été proposées et modifient assez radicalement la façon dont les systèmes réactifs sont désormais pensés et exécutés dans ce modèle de calcul.

Dans ce document, nous rappelons dans une première section les grands principes de l'approche réactive/synchrone. Ces rappels sont l'occasion d'expliquer certain choix dans le modèle de calcul qui a été retenu pour les **SugarCubes** v5 et également de présenter certains choix d'implantation.

Nous définissons ensuite le langage et sa sémantique opérationnelle structurelle sous forme de règles de réécriture. À l'aide de ces règles, nous cherchons à valider 3 propriétés :

1. **cohérence** : (il existe au moins une solution) pour tout programme écrit à l'aide des instructions du langage, on peut toujours construire un arbre de preuve d'exécution de ce programme à chaque instant ;
2. **déterminisme** : (il existe au plus une solution) à chaque instant, pour un programme donné et un environnement d'événements d'entrée donné, il existe une et une seule trace d'exécution événementielle (ensemble des événements présents et des listes des valeurs qui leur sont associées à l'instant donné après exécution de cet instant) ;
3. **réactivité** : l'exécution d'un programme au cours d'un instant se déroule par des vagues d'activations successives. Un instant termine toujours au bout d'un nombre fini (non fixé) d'itérations sur le programme réactif de l'horloge (machine d'exécution réactive) qui rythme son exécution et définit sont environnement d'exécution.

Ensuite, nous présentons l'implantation qui en a été faite en Java, ainsi que le support d'OpenCL et d'une implantation en C qui ont été ajoutés à l'implantation initiale qui était 100% pur Java.

Nous cherchons également à définir des mécanismes permettant de réécrire des programmes en s'appuyant par exemple sur une notion d'équivalence comportementale pour optimiser les programmes en cours d'exécution.

Enfin, nous décrivons tout un système d'outils permettant à partir de la description formelle de la sémantique de produire une implantation de référence permettant de comparer les résultats des implantations plus optimisées.

2 Les principes

Avant d'entrer, dans une explication détaillés des principes sous-jacents aux **SugarCubes**, il nous semble important de bien délimiter le but poursuivi par ces développements.

2.1 Les systèmes réactifs

L'objectif des SugarCubes est de proposer un cadre de programmation adapté à la construction de systèmes réactifs. Les systèmes réactifs que nous considérons dans ce document reposent sur la définition proposée par D. HAREL and A. PNUELI[10] au milieu des années 80. Un système réactif est un programme en cours d'exécution qui est en situation de réagir de façon permanente à toute nouvelle sollicitation de son environnement. Il ne s'agit pas dans ce cas de programmes ayant pour vocation à terminer c'est à dire à s'exécuter pendant un temps fini et à produire un résultat à la fin de leur exécution, mais au contraire, ces programmes sont en interaction constante avec leur environnement. Dans ce document nous considérons donc des programmes qui évoluent dans un environnement d'exécution dans lequel nous distinguons des entrées et des sorties. Une entrée est une donnée qui vient de l'extérieur à destination du programme et correspond en ce sens à une sollicitation ou un stimuli. Tandis qu'une sortie est une donnée provenant du programme et à destination de l'environnement extérieur correspondant à une action à réaliser par exemple par un actuateur. Les données de l'environnement d'exécution du programme peuvent donc être des entrées, des sorties ou

bien les deux à la fois.

Dans le cadre de la programmation réactive “à la” BOUSSINOT, chaque donnée de l’environnement est un événement, et chaque événement possède une liste de valeurs associées. Un événement pur est une donnée booléenne dont la valeur unique à chaque instant correspond à la présence ou à l’absence de cet événement. Un événement est présent chaque fois qu’il a été généré. À chaque génération d’un événement une valeur particulière peut-être associée on parle alors d’occurrence d’un événement. La liste des valeurs associées à chaque occurrence simultanée d’un événement constitue la liste des valeurs associées à un événement à un instant donné.

2.2 Notion d’horloge et exécution séquentielle

En tant que paradigme réactif/synchrone, la notion centrale derrière les **SugarCubes** est la notion d’horloge globale d’exécution. Une horloge est définie de façon discrétisée comme une succession (qui peut-être infinie) d’instants de durée finie (la durée est réputée ici aussi brève que possible à la différence d’un modèles synchrone pur où les instants ont une durée nulle). Le paradigme présenté ici, suppose que l’exécution d’un système est cadencée par une unique horloge globale. Cette horloge globale découpe l’exécution du système en morceaux. Chaque morceau d’exécution correspond d’un point de vue logique à ce qu’il faut exécuter du système au cours d’un instant de l’horloge globale. Par abus de langage on parlera d’instant d’exécution du système ou plus simplement d’instant. Ainsi, l’exécution complète du système est une simple succession d’instants d’exécution. Cela permet de définir de façon très simple et naturelle la notion d’exécution séquentielle de notre modèle de programmation. Une exécution séquentielle est un enchaînement d’instants.

Dans les langages de programmation synchrone comme Esterel ou Lustre, les instants d’exécution sont réputés atomiques et de durée d’exécution nulle. Dans ce modèle de calcul, le temps n’est consommé qu’entre deux instants consécutifs ; lorsqu’aucun calcul n’est effectué. Le programmeur est donc amené à raisonner sur son programme en supposant que les calculs qu’il programme sont infiniment rapides et ne prennent donc pas de temps (les machines sur lesquels s’exécutent ces calculs sont infiniment rapides) et que le temps physique ne s’écoule qu’entre deux instant de calculs. Bien évidemment, ce point de vue n’est que le cadre théorique dans lequel le programmeur doit travailler. C’est le niveau sémantique du langage de programmation et l’implantation concrète devra être "aussi proche que possible" de cette hypothèse. G. Berry appelle ce point de vue théorique le “paradigme de Newton” par analogie avec le cadre de raisonnement de la mécanique classique.

Du point de vue de l’implantation, on s’efforce donc de produire une approximation acceptable de la contrainte synchrone. L’idée clé ici est de déterminer quand est-ce que le système devient suffisamment réactif pour satisfaire les propriétés du modèle théorique. G. BERRY explique cela à travers l’exemple suivant : Un orateur qui s’exprime à voix haute dans une salle. Nous savons tous qu’il existe un certain temps entre le moment ou la bouche de l’orateur prononce ses mots, faisant vibrer (oscillation de la pression d’air locale) l’air ambiant et le moment où les ondes de pressions correspondantes parviennent aux oreilles de ses auditeurs. Mais en pratque, personne ne tient compte de ces temps de propagation (très négligeable par rapport aux grandeurs temporelles du signal lui-même). Si bien que tous (orateur et auditeurs) considèrent recevoir instantanément l’information diffusée. C’est ce que l’on entend par approximation raisonnable.

Les compilateurs de langages synchrones réalisent entre autre chose ce genre d’approximation afin de produire du code exécutable. G. BERRY appelle cela le paradigme vibrationnel. En pratique pas exemple, les compilateurs tentent de produire des approximation assez précise en générant de automates explicites (qui sont une représentation efficace du code mais qui passent difficilement à l’échelle[] et posent de nombreux obstacles techniques à la mise en place de mécanismes de compilation modulaire[]) ou en produisant non plus du code exécutable mais des circuits électroniques synchrones (pour mémoire la très grande majorités des circuits électroniques que nous produisons utilisent une horloge…).

Le modèle de calcul “à la” BOUSSINOT propose une approche légèrement différente : on ne distingue plus le niveau théorique et le niveau de l’implantation. Dans cette approche, on prend en considération au niveau sémantique le fait que l’exécution des instants prenne du temps. Mais ce temps d’exécution est fini (propriété de réactivité) et si possible doit satisfaire les contraintes temporelles garantissant la disponibilité du système vis à vis de son environnement extérieur. La durée d’un instant peut être vue comme le temps nécessaire au système pour se stabiliser après son activation afin de produire ses sorties. Ce point de vue est similaire au paradigme vibrationel évoqué plus haut. Mais F. BOUSSINOT et R. DE SIMONE profitent de cet affaiblissement sémantique pour se débarrasser d’un problème très difficile à maîtriser dans l’approche synchrone, qui est la possibilité de réagir instantanément à l’absence d’un signal (appelé événement dans notre cas) dans un système. Cette approche affaiblit le pouvoir expressif du modèle de programmation (plus de réaction instantanée à l’absence), mais la principale conséquence de ce choix est qu’il

élimine tous les problèmes de causalité devant être analysés par les compilateurs de l'approche synchrone forte avant d'accepter ou de refuser un programme. Ce faisant, le modèle réactif "à la" BOUSSINOT offre des moyens très simple de prendre en compte de changement dynamiques de configuration des systèmes (ajout en cours d'exécution de nouveau composants) et facilite l'utilisation d'une approche modulaire ascendante dans la conception de systèmes réactifs.

2.3 Notion d'instant et exécution parallèle

Si la notion d'horloge globale nous fournit la définition d'exécution séquentielle, les instants nous donnent un cadre de définition de la notion de parallélisme. Ceci est particulièrement vrai dans le cadre de l'approche synchrone pure puisque l'exécution d'un instant ne prend pas de temps d'un point de vue sémantique. Si tout se passe en même temps, il devient difficile d'imaginer que des opérations se déroulent en séquence. Ainsi l'exécution à un instant est parallèle. Cela revient à considérer que chaque composant du système doit réaliser en temps nul ce qu'il a à faire pour l'instant considéré. Toutes les opérations qui sont activés à cet instant ne prennent pas de temps d'exécution et nous ne pouvons donc pas préférer un ordonnancement plutôt qu'un autre. On doit donc considérer que toutes les opérations se déroulent en parallèle synchrone stricte. La seule considération à laquelle il faut s'attacher alors est de vérifier les éventuelles dépendances instantanées du système, pouvant affecter la causalité. Ces dépendances peuvent éventuellement compromettre la cohérence (système sans solution), le déterminisme (système ayant plus d'une solution), la causalité (solutions dont la conséquence valide une hypothèse) du système. La solution la plus commune à ces problèmes est de procéder à une analyse statique du système permettant (par approximation conservative) de rejeter les programmes présentant des cycles de causalité.

Cette réflexion sur la dualité entre instant et parallélisme s'applique également au modèle de calcul "à la" BOUSSINOT. Les instants d'exécution permettent de définir la notion d'exécution parallèle tandis que l'horloge globale fournit le cadre de définition de la notion d'exécution séquentielle. La principale différence ici est que par construction les systèmes réactifs construits en utilisant le paradigme "à la" BOUSSINOT ne peuvent pas exhiber de cycles de causalité.

2.4 État du système et opérations atomiques de calcul

Un programme réactif basé sur le modèle de calcul "à la" BOUSSINOT que nous présentons dans ce document est constitué de deux types d'instruction différentes :

- *Les instructions réactives*. Ils s'agit d'instructions permettant de manipuler l'ordonnancement des programmes (séquences, parallélismes, boucles...), de manipuler le temps et la communication par événements diffusés. Ces opérations ont généralement un état propre qui évolue à partir du moment où elles sont activées d'un instant d'exécution à l'autre. Parmi les instructions réactives, on peut différentier les instructions en deux sous-catégories :
  - les instructions qui consomment du temps (des instants) ;
  - les instructions qui terminent leur exécution instantanément (qui ne consomme pas de temps).
- *Les opérations atomiques* dont l'exécution est instantanée (ne consomme pas de temps) et qui sont responsable des calculs sur les données du système. Les opérations atomiques en **SugarCubes** v5 ne sont plus des instructions des programmes réactifs mais une instructions (appelée `action`) permet de signaler au système qu'une action particulière doit être collectée pour être exécutée.

L'état d'un système réactif est représenté par l'état des instructions réactives activées, l'ensemble des événements valués présents et l'ensemble des valeurs des cellules mémoire manipulées par le système à un instant donnée. Dans la plupart des implantations du modèle réactif "à la " BOUSSINOT, les cellules mémoire sont manipulées par l'exécution d'opérations atomiques dont un des principaux effets est d'affecter de nouvelles valeurs à ces cellules mémoire. C'est en particulier le cas de versions antérieurs à la v5 des **SugarCubes**. Ces opérations atomiques sont généralement définies dans un langage hôte (TCL-Tk dans les scripts réactifs, Java en SugarCubes...). Généralement les opérations atomiques sont exécutées pendant la résolution d'un instant au fur et à mesure de leur activation dans le programme. **SugarCubes** v5 propose de gérer différemment l'état des cellules mémoires et des actions atomiques.

En particulier, **SugarCubes** v5 introduit explicitement la notion de cellules mémoire qui se trouve désormais être la donnée des éléments suivants :

- un nom identifiant la cellule. Cet identifiant est utilisé par l'instruction réactive `action` pour annoncer au système que l'opération atomique associée à cette cellule doit être exécutée. Si plusieurs instructions `action` référencent la même cellule au cours d'un même instant une seule demande d'exécution sera enregistrée ;
- une valeur courante de la cellule accessible la plupart du temps en lecture seule ;
- une valeur future de la cellule accessible en écriture dans laquelle on affecte au plus une fois par instant le résultat du calcul de l'opération atomique associée ;

– une opération atomique qui est une fonction responsable de calculer à chaque instant où elle est activée une nouvelle valeur qui sera affectée à la valeur future de la cellule ; Cette opération accède en lecture seule à l'ensemble des valeurs des cellules mémoire du système et à la liste des événements valués.

Comme les instants ne sont pas de durée nulle, le modèle de calcul "à la" BOUSSINOT propose de décomposer l'exécution d'un instant d'un système réactif en une succession finie de micro-étapes. Dans le cadre des **SugarCubes** v5 la structure d'un instant est constituée de la séquence suivante :

– *l'exécution réactive*, qui correspond à l'exécution des instructions réactives seules. Durant cette étape, les événements peuvent être générés et ainsi débloquer des composants en attentes sur ces événements, les action atomiques sont collectées et ordonnées dans des files d'attente mais non exécutées…
– *la construction des listes des valeurs associées aux événements émis* À la fin de cette première étape les listes de valeurs associées aux événements générés sont construites dans un ordre déterminé par la seule structure syntaxique du programme. Si bien que cet ordre peut être précisément déterminé tant qu'il ne résulte que de l'exécution même du programme.
– *l'évaluation des opérations atomiques* toutes les opérations atomiques collectées au cours de l'étape précédente sont ensuite exécutée dans un environnement d'événement déterminé (figé au cours de l'étape précédente) et un environnement mémoire accessible en lecture seule. Chaque opération atomique calcul une valeur pour une cellule mémoire donnée de sorte qu'il ne peut y avoir qu'au plus une seule nouvelle valeur calculée pour une cellule mémoire donnée.
– *l'écrasement des anciennes valeurs des cellules mémoire par les nouvelles* au cours de cette dernière étape. Le système ne procède qu'à des écritures et pour chaque cellule mémoire la nouvelle valeur calculée vient remplacer la valeur précédente de sorte que la nouvelle valeur sera utilisable à l'instant suivant.

Cette séquence d'opération permet d'isoler chaque opération en mémoire centrale. Ainsi, au cours de chaque étape, une case de la mémoire centrale quelle qu'elle soit est accessible exclusivement en lecture ou en écriture. Si elle est accessible en écriture, alors il n'y a qu'un seul et unique écrivain possible au cours de cette étape. De sorte qu'aucun mécanisme de protection n'est nécessaire. Cette solution permet de reposer sur une implantation réellement parallèle, avec cependant la contrainte suivante qui est la nécessaire barrière de synchronisation que doivent franchir tous les fils d'exécution parallèle utilisés pour implanter cette solution entre chaque étape. De fait en fonction de la complexité de la tâche à exécuter, l'implantation pourra choisir de paralléliser ou non une étape.

2.5 Ordonnancement en ligne et exécution interprétée des programmes

Écrire un programme réactif dans une approche "à la" BOUSSINOT revient à décrire comment en fonction de données d'entrée (les événements d'entrée du système) et de l'état du système, les actions atomiques qui vont être enregistrées ou non pour une exécution au cours de l'instant. On peut donc considérer qu'écrire le comportement d'un programme réactif revient à exprimer une politique d'ordonnancement des actions atomiques en s'appuyant sur la notion d'instant.

Dans le cas des langages synchrones, l'implantation favorise un ordonnancement dit "off-line" résolu à la compilation où la génération de code revient à produire une séquenceur sous la forme d'un automate (explicite ou implicite). Cette mise en œuvre est cohérente avec le fait que la compilation du système complet dans l'approche synchrone nécessite une analyse statique globale. Dans le cas le cas de l'approche réactive "à la" BOUSSINOT, cette analyse globale n'est plus de mise et nous pouvons donc envisager une mise en œuvre plus dynamique permettant d'ajouter en cours d'exécution de nouveaux composants parallèles. Dès lors, il devient intéressant de privilégier une ordonnancement "on-line" car la production statique d'un séquenceur empêche la mise en œuvre de ce dynamisme. Une approche statique a été explorée avec le langage SL au milieu des années 90[]. Mais depuis, l'essentiel des formalismes étudiés dans le cadre de l'approche réactive "à la" BOUSSINOT repose sur une exécution interprétée. Ainsi, les **SugarCubes** proposent depuis la première implantation une machine d'exécution, jouant le rôle d'un interprète de l'exécution des instructions réactives qui constituent le programme. Cet interprète, à l'instar d'un ordonnanceur dans un système d'exploitation, interprète la politique d'ordonnancement en cours d'exécution du système et exécute les tâches (qui elles sont généralement compilées) en fonction de cette politique. Cette exécution interprétée de la politique d'ordonnancement d'un système d'exploitation est très bien mise en évidence dans les travaux autour de l'environnement système BOSSA[11] and [12].

Dans le cas de l'approche réactive "à la" BOUSSINOT, un programme réactif va un peu au delà de la stricte politique d'ordonnancement d'un système d'exploitation, puisque le programme réactif prend aussi en compte la communication entre les composants parallèles. Ainsi les **SugarCubes** reposent sur une machine d'exécution qui joue le rôle d'un interprète des programmes réactifs et alloue les ressources de calculs physique du système en fonction des tâches atomiques à réaliser à chaque instant. Cette solution permet de modifier la structure d'un système réactif

en cours d'exécution en autorisant l'ajout de nouveaux composants sans avoir besoin d'arrêter le système. On peut en quelque sorte considérer que l'ordonnancement des tâches spécifique à chaque système réactif est donc réalisé au niveau utilisateur et que les services du système d'exploitation sous-jacent sont sollicités à minima pour exécuter le système.

3 Les constructions du langage

Les **SugarCubes** constituent une librairie de classes au dessus de Java et non pas un véritable langage avec une syntaxe spécifique. Le programmeur construit des programmes en enchaînant des appels de méthodes de la classe `SC` qui lui permettent de construire l'arbre de syntaxe abstraite du programme réactif qu'il souhaite voir exécuter par une machine d'exécution. Il est envisagé de produire un analyseur lexical et grammatical par exemple en utilisant JavaCC ou AntLR pour faciliter la construction des programmes réactifs mais ceci n'est toujours pas implanté. Cependant nous allons présenter dans cette section le langage non pas à travers la réalité des appels aux méthodes java de la classe `SC` mais à travers une syntaxe plus simple facilitant la lecture et qui pourrait être à terme implantée. Afin de simplifier notre modèle nous ne considérons pas dans cette section la notion d'événement valué. Les seuls informations manipulées sur les événements sont la présence ou l'absence de ceux-ci (on parle d'événements purs, à l'instar des signaux purs d'Esterel).

3.1 Les instructions

Le langage de programmation réactif que nous présentons ici est inspiré du langage Esterel[**?**] et du langage des scripts réactifs proposé par L. HAZARD et F. BOUSSINOT dans [13]. Dans cette section nous présentons la syntaxe des instructions permettant la construction des programmes réactifs. Le langage contient les 15 instructions suivantes :

- *clock* : permet d'exécuter un instant d'un système réactif dont le programme lui est donné en paramètre.
  Syntax : `clock:*c t end`
  où l'identificateur `c` représente le nom de l'horloge et le terme `t` représente le programme du système.
- || : représente le parallélisme. C'est un opérateur binaire qui dans notre cas est déterministe. $a \mid\mid b$ représente l'exécution parallèle à chaque instant de a et de b. L'opérateur de combinaison parallèle termine lorsque les deux constructions mises en parallèle terminent.
  Syntax : `t || u`
  où les termes `t` et `u` sont des termes quelconques.
- ; : est un opérateur permettant de construire la séquence : $a; b$ dit que l'on commence par exécuter l'instruction a puis dès que a termine, on exécute b
  Syntax : `t ; u`
- *loop* : représente la boucle infinie qui à chaque itération exécute le corps. Par exemple t dans $loop\ t\ end$
  Syntax : `loop t end`
- *repeat* :
  Syntax : `repeat n times t end`
- *control* : active un programme réactif à chaque instant ou un événement est présent. L'alternance de présence et d'absence de l'événement au cours des instants constitue la sous horloge d'exécution du programme contrôlé.
  Syntax : `control t by &e`
- *kill* :
  Syntax : `kill t on &e handle u end`
- *when* :
  Syntax : `when &e then t else u end`
- *await* : est une instruction bloquante. La progression d'un composant sequenctiel est bloquée jusqu'à ce que l'événement attendu (par exemple e dans l'instruction $await\ e$) soit présent dans l'environnement. Elle termine alors "instantanément" et l'exécution de la séquence reprend immédiatement (dans le même instant).
  Syntax : `await &e`
- *pause for ever* : est une instruction qui suspend indéfiniment l'exécution d'une séquence d'instructions.
  Syntax : `pause for ever`
- *pause n times* : fonctionne exactement comme `repeat n times pause end`
  Syntax : `pause n times`
- *pause* : est une instruction qui suspend l'exécution d'une séquence d'instructions pour l'instant courant. L'exécution reprendra à l'instant suivant juste après l'exécution du *pause*
  Syntax : `pause`
- *action* : est une instruction permettant d'enregistrer le calcul d'une (et d'une seule) nouvelle valeur d'une cellule mémoire au cours de l'instant courant.

Syntax : `action @a`

- $generate$: est une instruction atomique qui génère un événement désigné par le nom e dans l'environnement d'exécution et termine immédiatement. L'événement est présent à l'instant où il est généré.
  Syntax : `generate &e`
- $nothing$: est une instruction atomique qui ne fait rien...
  Syntax : `nothing`

3.2 La machine d'exécution

La machine d'exécution fournit un contexte d'exécution sous la forme d'un ensemble de noms définissant la présence ou l'absence des événements. L'environnement d'exécution évolue donc en accumulant les noms des événements présents au cours d'un instant. Au début de chaque instant cet ensemble est vide et le programme est activé dans cet environnement.

3.3 traduction des constructions concrètes du langage dans la syntaxe abstraite des règles

On définit une fonction de traduction des instructions écrites dans la syntaxe concrète du langage en termes écrits dans la syntaxe abstraite :

- $tr($ `"clock:*c t end"` $) \rightarrow close(tr($ `"t"` $))$
- $tr($ `"t || u"` $) \rightarrow par(cons(tr($ `"t"` $), cons(tr($ `"u"` $), nil), nil))$
- $tr($ `"t ; u"` $) \rightarrow seq(cons(tr($ `"t"` $), cons(tr($ `"u"` $), nil)))$
- $tr($ `"loop t end"` $) \rightarrow loop(tr($ `"t"` $))$
- $tr($ `"repeat n times t end"` $) \rightarrow repeat\_n\_times(tr($ `"n"` $), t)$
- $tr($ `"control t by &e"` $) \rightarrow control(tr($ `"e"` $), tr($ `"t"` $))$
- $tr($ `"kill t on &e handle u end"` $) \rightarrow kill(tr($ `"e"` $), tr($ `"t"` $), tr($ `"u"` $))$
- $tr($ `"when &e then t else u end"` $) \rightarrow when(tr($ `"e"` $), tr($ `"t"` $), tr($ `"u"` $))$
- $tr($ `"await &e"` $) \rightarrow await(tr($ `"e"` $))$
- $tr($ `"pause for ever"` $) \rightarrow pause\_for\_ever$
- $tr($ `"pause n times"` $) \rightarrow pause\_n\_times(tr($ `"n"` $))$
- $tr($ `"pause"` $) \rightarrow pause$
- $tr($ `"action @a"` $) \rightarrow action(tr($ `"a"` $))$
- $tr($ `"generate &e"` $) \rightarrow generate(tr($ `"e"` $))$
- $tr($ `"nothing"` $) \rightarrow nothing$

La machine d'exécution est activée de façon régulière. Chaque activation calcule l'exécution d'un nouvel instant et produit une trace d'exécution (l'ensemble des événements présents à cet instant). Au début de chaque instant on considère le programme t dans un environnement vide : $t, \emptyset$

3.4 Les règles

La sémantique est donnée sous forme d'un ensemble de règles de réécriture de termes exprimés dans la syntaxe abstraite. Ces règles décrivent la sémantique opérationnelle structurelle (SOS) des constructions du langage, c'est-à-dire le comportement élémentaire d'une construction dans son contexte d'exécution au cours d'un seul instant. Cette sémantique est "quasi petit pas". Chaque règle représente le comportement d'une instruction à chaque activation au cours d'un unique instant du programme dans la machine d'exécution. La machine d'exécution converge par réécritures successives du programme jusqu'à un état stable (point fixe) appelé fin d'instant. On décompose donc l'exécution d'un instant du programme en une succession d'applications de l'ensemble des règles de réécritures au programme appelées activation.

Format des règles SOS à la Plotkin :

$$activ\,(t, E, M) \rightarrow \alpha\,(t', P, A)$$

Un terme t activé dans un environnement d'événements E et un environnement de cellules mémoire M, se réécrit en un terme t' produisant un ensemble d'événements P et un ensemble de nouvelles cellules mémoire à activer à la fin de l'instant A. Cette activation retourne un statut d'exécution α dont les valeurs possibles sont `SUSP`, `STOP` ou `TERM`. Les valeurs du status d'exécution α ont le sens suivant :

- `SUSP` signifie que l'activation de l'instruction t l'a conduite dans un état non stabilisé, il faudra réactiver cette instruction avant que l'instant ne soit terminé. En général, un état non stabilisé est atteint par une instruction tant

que celle-ci attend un événement non encore généré. La situation ne pourra se débloquer que si l'événement est généré (l'événement est finalement là) ou que la fin d'instant est décrétée (l'événement attendu est finalement absent) ;
– `STOP` signifie que l'activation de l'instruction t l'a conduite dans un état stable marquant la fin de l'instant pour cette instruction. Il ne faut plus réactiver cette instruction dans l'instant courant. L'instruction devra être activée à l'instant suivant ;
– `TERM` signifie que l'instruction à complètement terminé son exécution. Elle ne sera plus réactivée.

On définit également le format des règles conditionnelles de la façon suivante :

$$\frac{\textit{predicates}}{\textit{activ}\,(t, E, M) \rightarrow \alpha\,(t', P, A)}$$

les prédicats sont placés au dénominateur tandis que la règle de réécriture à appliquer au terme est placée au numérateur. L'interprétation de ces règles est la suivante : lorsque la conjonction des prédicats est vraie, alors on applique la règle de réécriture au terme à réécrire.

Nous allons maintenant présenter des règles décrivant la sémantique de quelques instructions constituant la syntaxe abstraite afin d'expliquer les principes. L'ensemble des règles sont données dans l'annexe A de ce document.

Ainsi, pour l'instruction $nothing$ nous avons la règle suivante :

$$\textit{activ}\,(\textit{nothing}, E, M) \rightarrow \texttt{TERM}\,(\emptyset, \emptyset)$$

Cette règle montre que l'instruction $nothing$ termine immédiatement et produit ni événement, ni n'enregistre de cellule mémoire à activer à la fin de l'instant. Cette instruction ne fait rien et termine instantanément et ne peut plus être activée.

Pour l'instruction $pause$ qui permet de consommer du temps en repoussant la fin de son exécution à l'instant suivant nous avons la règle suivante :

$$\textit{activ}\,(\textit{pause}, E, M) \rightarrow \texttt{STOP}\,(\textit{nothing}, \emptyset, \emptyset)$$

Outre le statut `STOP` on voit que le programme est réécrit en une simple instruction $nothing$ qui sera activée à l'instant suivant. Ici encore aucune production d'événement, ni d'action n'est effectuée.

Pour l'instruction $generate$ qui permet de générer un événement au cours d'un instant nous avons la règle suivante :

$$\textit{activ}\,(\textit{generate}\,(v), E, M) \rightarrow \texttt{TERM}\,(\{v\}, \emptyset)$$

Le comportement de cette instruction est similaire à celui de l'instruction $nothing$ mais un nouvel événement est produit, il s'agit de l'événement référencé par le contenu de la variable v. On retrouve donc ce nom dans l'ensemble des événements produits. Par contre aucune action n'est produite sur les cellules mémoires du système par cette opération.

L'instruction $action$ est similaire à l'instruction $generate$, mais elle opère sur les cellules mémoire et non pas sur les événements du système. La règle de réécriture est la suivante :

$$\textit{activ}\,(\textit{action}\,(v), E, M) \rightarrow \texttt{TERM}\,(\emptyset, \{v\})$$

L'instruction $await$ permet de suspendre l'exécution d'une séquence tant qu'un événement n'est pas présent. Elle correspond à l'ensemble des règles suivantes :

$$\frac{v \in E}{\textit{activ}\,(\textit{await}\,(v), E, M) \rightarrow \texttt{TERM}\,(\emptyset, \emptyset)} \tag{3.1}$$

$$\frac{v \notin E}{\textit{activ}\,(\textit{await}\,(v), E, M) \rightarrow \texttt{SUSP}\,(\textit{await}\,(v), \emptyset, \emptyset)} \tag{3.2}$$

$$\textit{eoi}\,(\textit{await}\,(v), E) \rightarrow \textit{await}\,(v) \tag{3.3}$$

La première règle, montre comment se comporte l'instruction, lorsque l'événement attendu est présent : on voit que l'instruction termine immédiatement à l'image de l'instruction $nothing$.

Par contre, si l'événement e n'est pas là, ce sont les règles 2 et 3 qui sont utilisées pour décrire le comportement de l'instruction. Pendant la phase d'activation et en l'absence de l'événement attendu l'instruction se réécrit en elle même et retourne un statut d'exécution `SUSP` indiquant qu'aucune décision définitive sur l'exécution n'a pu être prise pour le moment. On constate qu'au moment où l'instruction $await$ est activée l'événement n'est pas présent mais rien n'empêche un autre composant parallèle de pouvoir le générer un peu plus tard (c'est là que la différence avec l'approche synchrone est sensible) auquel cas il faudra toujours au cours du même instant réactiver ce composant pour qu'il prenne connaissance de la présence de l'événement attendu. Si la fin de l'instant vient à être décrétée par la machine d'exécution (cf. A.9) sans que l'événement attendu n'ait été vu présent alors durant la phase d'activation à la fin de l'instant la règle 3 3 sera utilisée. La décision finale est prise quant à l'absence de l'événement attendu et l'instruction se réécrit en elle même pour attendre l'instant suivant.

4 Preuves de propriétés sur le système de règles [1]

Après avoir décrit les principales constructions du langage et formalisé leur sémantique nous allons nous attacher à prouver certaines propriétés sur le comportement de la machine d'exécution réactive. Pour ce faire, nous concentrons nos efforts sur la partie purement réactive du langage, en supposant que les actions atomiques terminent. En particulier, nous cherchons à garantir que :

- *cohérence des programmes* : pour tout programme réactif syntaxiquement correct, exprimé dans la syntaxe concrète du langage, il existe toujours un arbre de preuve d'exécution permettant de d'écrire à chaque instant l'exécution dans ce programme modulo les signaux d'entrée du système collectés à chaque instant ;
- *déterminisme des programmes* : pour tout programme réactif syntaxiquement correct, exprimé dans la syntaxe concrète du langage, il existe un seul et terme résiduel de l'exécution d'un instant du programme, une seule trace d'exécution événementielle (ensemble des événements générés et des listes de valeurs associées) et une seule configuration de valeur des cellules mémoire en fonction des entrées du système à chaque instant ;
- *réactivité du système* : pour tout programme réactif syntaxiquement correct, exprimé dans la syntaxe concrète du langage, l'exécution d'un instant termine toujours quelque soit la configuration des cellules mémoires et les entrées du système.

Pour ce faire, nous allons utiliser des outils de construction de preuve automatisée manipulant des systèmes de réécriture. Il s'agit de l'outil CIME[] développé par l'équipe CPR du laboratoire CEDRIC, et de l'outil AProVE[]. Ces outils vont nous permettre de construire automatiquement les preuves de terminaison ou au moins de construire une bonne partie de cette preuve, puis au moyen d'outils de génération de trace sur la construction de la preuve de soumettre ces traces à l'outil Coq afin de certifier ces preuves.

La construction de ces preuves a résulté d'un long travail de maturation et la spécification même du système a été à de nombreuses reprises entièrement réécrites. Dans un premier temps nous allons retracer quelques éléments relatifs à l'historique de ce travail, puis nous présenterons les différents résultats correspondant aux différentes propriétés que nous cherchons à prouver.

4.1 Transformation du système de réécriture

Les spécifications formelles de la sémantique sur lesquelles nous avons basé nos travaux utilisent une représentation de la sémantique opérationnelle sous la forme de systèmes de règles de réécriture conditionnelles. Cependant, les outils sur lesquels nous nous sommes appuyés pour construire les preuves souhaitées (CIME et APRoVE) sont avant tout des outils ne considérant que des systèmes de réécriture non conditionnelle. Un premier travail a donc consisté à mettre en forme le système de règles conditionnelles pour être transformé en un système de réécriture non conditionnel capable d'être traité par les outils en question. Les premières manipulations du système de règles ont été réalisées en utilisant l'outil CIME. Les règles ont alors été réécrites manuellement pour correspondre au format d'entré de l'outil. Ce format d'entré accepte un système de règles conditionnelles qu'il faut dans un premier temps, «déconditionnaliser» par une transformation automatique implantée dans l'outil CIME. Une fois déconditionnalisé le système est soumis à nouveau à l'outil pour tenter de construire la preuve de terminaison. Une première évaluation de la faisabilité a ainsi été faite sur un système de règles très simple représentant partiellement le fonctionnement du système réel. Une fois transformé en système de réécriture simple, le TRS peut-être fourni indifféremment aux outils APROVE ou CIME. La traduction du système de règles dans le format d'entrée de CIME a été l'objet de nombreuses expérimentations. Lorsque le système de preuve devenait plus complexe, l'outil de preuve échouait. Après un certain

1. Ce travail a été mené en étroite collaboration avec l'équipe CPR et en particulier, X. Urbain, P. Courtieu et J. Forest

nombre d'expérimentations, nous avons considéré que la traduction d'un CTRS en TRS dans l'outil CIME, n'était pas satisfaisante, car si elle préservait les propriétés nécessaires pour rendre possible la preuve de terminaison, le système produit donnait naissance à une myriade de réécritures possibles qui n'existaient pas dans le système d'origine. Nous avons alors commencé l'étude de transformation dite «computationally equivalent» à partir de l'article []. Ceci nous permettrait peut-être de comprendre pourquoi la preuve échouait. Parallèlement l'équipe CIME a produit des outils permettant de réduire des termes **SugarCubes** en y appliquant le système de règles et d'observer les résultats.

Parmi les travaux qui ont été menés en parallèle, nous avons repensé l'ensemble des outils visant à produire le sytème de réécriture dans le format d'entrée de CIME.

Un format textuel ASCII de spécification de règles

Nous avons commencé par définir un format spécifique de représentation des règles de sémantiques associés à chaque instruction (un fichier par instruction contenant l'ensemble des règles décrivant la sémantique petit pas de l'instruction) :

Par exemple, pour l'instruction $pause$ le fichier ou se trouve spécifié la sémantique de cette instruction s'appelle `Pause.rules` et contient l'unique règle suivante :

```
activ(pause, E, M) -> STOP(nothing, void, void)
```

Traduction au format CIME 3

Un outil d'extraction permet de transformer ce fichier en une équation latex qui est par exemple utilisé pour composer ce document, tandis qu'un autre outil construit la règle dans le format d'entrée de l'outil CIME :

```
(* Instruction pause *)
  activ(pause, E, M) -> STOP(nothing, void, void);
```

Pour se rendre compte des différences, il faut considérer une règle conditionnelle plus complexe. Par exemple, considérons l'instruction $await$:

```
v in E |activ(await(v), E, M) -> TERM(void, void);
v notin E |activ(await(v), E, M) -> SUSP(await(v), void, void);
eoi(await(v), E) -> await(v)
```

Les règles conditionnelles vont être transformées en règles d'un système non conditionnel de la façon suivante :

- un nouveau terme va être créé pour chaque prédicat de la condition (dans le cas de l'instruction $await$ il n'y a qu'un seul prédicat, ce qui nous donne un seul nouveau terme $U\_activ\_await\_1$). Ce nouveau terme a pour but de forcer la réécriture du prédicat. il doit donc toujours pouvoir se réécrire. Ce terme va avoir comme premier argument le terme représentant le prédicat, le terme à réécrire mais non activé (ce qui bloque sa réécriture) $await(v)$, l'environnement d'événement dans lequel à lieu la réécriture E et l'ensemble des cellules mémoire de l'environnement M.

  ```
  activ(await(v), E, M) -> U_activ_await_1(in(v, E), await(v), E, M);
  ```

- Dans notre cas le terme intermédiaire force la réécriture de la condition $in(v, E)$ qui est un terme calculant l'appartenance du nom contenu dans la variable v (nom de l'événement attendu) dans l'ensemble des événement présents au cours de l'instant E. Ce terme se réécrit en 2 valeurs possibles : $true$ ou $false$. On a alors autant de réécritures du terme intermédiaire qu'il peut y avoir de résultats à la réécriture du prédicat. Dans notre cas nous avons 2 termes.

  ```
  U_activ_await_1(true, await(v), E, M) -> TERM(void, void);
  U_activ_await_1(false, await(v), E, M) -> SUSP(await(v), void, void);
  ```

- Le réécriture d'un terme intermédiaire conduit à un nouveau terme intermédiaire permettant l'évaluation du prédicat suivant ou si on a évalué tous les prédicats, conduit à la réécriture du terme de départ.

pour fonctionner, cette traduction doit donner un ordre aux prédicats et les traiter toujours dans le même ordre et une réécriture des termes intermédiaire doit-être donnée pour toute forme nomale du terme représentant le prédicat.

Dans ces conditions, les termes intermédiaires doivent disparaitre lorsque l'on réécrit un terme en forme normale.

```
(* Instruction await *)
    U_activ_await_1(true, await(v), E, M) -> TERM(void, void);
  eoi(await(v), E) -> await(v);
    U_activ_await_1(false, await(v), E, M) -> SUSP(await(v), void, void);
  activ(await(v), E, M) -> U_activ_await_1(in(v, E), await(v), E, M);
```

L'annexe B de ce document fournit le listing complet des règles obtenu grâce à l'outil de traduction des règles dans le format d'entrée de l'outil CIME.

4.2 Propriétés

Nous avons concentré nos efforts sur la preuve de confluence et la preuve de terminaison. La preuve de confluence du système a été réalisé à l'aide de l'outil CIME sous la condition que le système termine. La preuve de terminaison des instants repose sur la preuve de terminaison du système de règles de réécriture puisque ce système décrit le calcul d'un point fixe correspondant à l'exécution d'un instant. Cette preuve n'est pas encore achevée. Une preuve a été faite sur le système complet privé de la règle A.8 à l'aide de l'outil APRoVE. Nous cherchons maintenant à terminer la preuve manuellement en Coq en tentant de montrer que la preuve peut-être étendue au système de règles complet. Cette preuve est en cours de finalisation.

5 Implantation

L'implantation en Java reprend dans les grandes lignes les implantations des versions antérieures. Ainsi le framework dispose d'une implantation 100% pure Java dans laquelle on retrouve une API exposée à l'utilisateur à travers le package `fr.cedric.sugarcubes`. Ce package Java renferme un ensemble de classes et d'interfaces permettant de manipuler, de créer et de mettre en œuvre les programmes réactifs.

Un second package (que le développeur utilisant les **SugarCubes** n'est pas sensé connaître) contient l'implantation effective de l'environnement.

Dans cette section, nous allons commencer par décrire l'implantation des Cubes et le nouveau modèle d'objet réactif qui leur sont associé. Nous allons décrire l'interface, en détail avant de discuter des principes utilisés dans l'implantation.

5.1 Les cubes

Un programme **SugarCubes** est avant tout construit autour de la notion d'objet réactifs appelé «cubes». Un cube est instancié à partir d'une sous classe de la classe `Cube`.

```
import fr.cedric.sugarcubes.*;

public class FloatFieldExample extends Cube
{
}
```

La notion de Cube se traduit dans la version 5 maintenant par le mécanisme d'héritage contrairement aux versions précédentes. Les champs du cubes doivent-être des Cellules Java (déclarées à partir des interfaces de manipulation des cellules élémentaires : `DoubleCell`, `FloatCell`, `IntCell`, `LongCell`, `OCLCell`, `OCLFloatCell`, `StringCell`).

Chacune des interfaces permettant de manipuler un champ cellule possède une méthode permettant d'accèder à la valeur du champ :

```
package fr.cedric.sugarcubes;

public interface DoubleCell extends Program
{
  double getDoubleValue();
}
```

La création d'une instance du champ est réalisée par l'appel à l'une des méthodes idoines de la classe `Cube` :

```
public final DoubleCell a = floatCell("a", 0, new JavaFloatComputation(){
   public float evaluate(final Environment env, final float self){
     return self+1;
   }
});
```

Pour implementer la fonction permettant de calculer les nouvelles valeurs de la cellule, l'utilisateur des **SugarCubes** doit fournir à la construction une instance d'une classe implantant une interface spécifique à la valeur devant être calculée : `JavaDoubleComputation`, `JavaFloatComputation`, `JavaIntComputation`, `JavaLongComputation`, `JavaStringComputation`.

Les cellules utilisant le support OpenCL n'ont pas d'interface correspondante, le mécanisme étant différent (cf. 5.5).

Ces interfaces proposent une méthode `evaluate()` (dont la signature est généralement différente) qui est appelée par la machine d'exécution réactive à la fin d'un instant si l'état de la cellule à laquelle elle est attachée doit-être mis à jour. L'implantation de cette méthode doit être faite indépendamment de l'état éventuel de la classe qui l'implémente. Le calcul de la nouvelle valeur ne doit dépendre que :

- de la valeur d'autres cellules mémoire gérées par la même horloge réactive
- de la valeur courante de la cellule elle-même que l'on reçoit comme deuxième argument de la méthode `evaluate()`
- des valeurs associées à des événements générés au cours de l'instant et accessible à travers les méthodes , `getValuesOf`, `getFloatValuesOf` de l'objet implantant l'interface `Environment` que l'on reçoit en premier argument de la méthode `evaluate()`.

5.2 La construction d'un programme et de sa machine d'exécution

Cette API est accessible à travers le package `fr.cedric.sugarcubes`. On y retrouve un certain nombre de classes et d'interfaces permettant à l'utilisateur de manipuler l'environnement d'exécution **SugarCubes** et de créer les programmes. Une classe joue ici un rôle essentiel ; il s'agit de la classe `SC`. Cette classe propose un ensemble de méthodes de classe permettant d'instancier les principaux composants :

- la machine d'exécution réactive : `machine()`
- des identificateurs d'événements : `simpleID()`
- des instruction réactives : `nothing()`, `pause()`, `pauseForEver()`, `repeat()`, `loop()`, `seq()`, `par()`, `await()`, `generate()`, `when()`, `kill()`, `control()`.

L'interface `EventIdentifier` permet de manipuler des noms d'événements. Les noms peuvent être facilement générés grace à la méthode `simpleID()` de la classe `SC`.

L'interface `Machine` permet de manipuler des machines d'exécution réactive. Cette interface ne doit pas être implantée par l'utilisateur, car elle n'offre pas suffisamment les services minimaux permettant l'implantation d'une véritable machine d'exécution. La création d'une machine d'exécution se fait à l'aide d'une des méthodes correspondantes de la classe `SC`.

L'interface `Program` permet de manipuler un programme réactif enchaînant des instructions réactives créées au moyen des méthodes correspondantes. L'activation des cellules mémoire (champs des objets réactifs) est représentée par la cellule elle-même qui implante l'interface `Program`.

Le programme réactif une fois construit est ajouté dans la machine d'exécution (en parallèle aux autres programmes si d'autres programmes ont été rajoutés préalablement) à l'aide de la méthode `addProgram()` de la machine d'exécution.

```
m.addProgram(SC.seq(new Program[]{
          SC.pause(10)
          , c.a
          , SC.pause(10)
      }));
```

5.3 Exemple

```
package test;

import fr.cedric.sugarcubes.*;

public class DoubleFieldExample extends Cube
{
  public static void main(final String args[]){
    final DoubleFieldExample c = new DoubleFieldExample();
    final Machine m = SC.machine();
    System.out.println("starting value of c.a = "+c.a.getDoubleValue());
    m.addProgram(SC.seq(new Program[]{
              SC.pause(10)
              , c.a
              , SC.pause(10)
          }));
    m.run();
    System.out.println("final value of c.a = "+c.a.getDoubleValue());
  }

  public final DoubleCell a = doubleCell("a", 0, new JavaDoubleComputation(){
     public double evaluate(final Environment env, final double self){
       return self+1;
     }
  });
}
```

Ce programme crée un «Cube» en étendant la classe `Cube`. Celui-ci possède un champ de type `DoubleCell`, initialisé par l'appel à la méthode `doubleCell()` de l'objet Cube. Remarquons que la cellule elle-même est initialisée une fois pour toute donc elle est déclarée avec le mot clé `final`, tandis que sont contenu évoluera à partir d'une valeur initiale (0), chaque fois que la méthode `evaluate()` de l'interface `JavaDoubleComputation` sera appelée par la machine d'exécution réactive. Ici la méthode se contente d'incrémenter la valeur courante de la cellule (`self`). Le programme réactif appelle 2 fois la cellule `a` du cube `c`. Ce programme incrémentera une fois la valeur du champ après 10 instants d'exécution, puis encore une fois après 10 autres instants.

5.4 L'implantation Java pure

L'implantation en Java pur se trouve dans le package fr.cedric.sugarcubes.i qui contient l'ensemble des classes les machines d'exécution et les instructions réactives. Ces objets ne sont pas manipulables directement par le programmeur **SugarCubes**. La machine d'exécution découpe l'exécution d'un instant en 4 phases consécutives :

- la phase purement réactive durant laquelle les instructions réactives sont activées. Cette phase correspond à l'exécution de la politique d'ordonnancement pour l'instant courant. L'environnement d'exécution collecte les événements générés et les action atomiques à réaliser ;
- la phase de construction des listes d'événements valués ;
- la phase de calcul des nouvelles valeurs de cellules mémoire. Cette phase peut-être efficacement parallélisée si le matériel sous-jacent le permet, les nouvelles valeurs calculées sont stockées dans des variables temporaires ;
- la phase de mise à jour de l'état mémoire. Cette phase permet de recopier les valeurs précédemment calculées dans les cellules mémoires exposées par le système.

L'état actuel de l'implantation permet d'utiliser les ressources parallèles pendant les deux dernières étapes. Nous travaillons à la parallélisation des deux premières. Les barrières de synchronisation à chaque étape facilitant la parallélisation et évitant la prise de verrous au cours d'une étape.

Concernant l'implantation de la partie purement réactive, chaque instruction réactive correspond à une classe particulière implantant essentiellement l'interface src.fr.cedric.sugarcubes.i.Instruction :

```
package fr.cedric.sugarcubes.i;

import fr.cedric.sugarcubes.*;

public interface Instruction extends Program
```

```
{
  public static final int SUSP = 1;
  public static final int WEOI = 2;
  public static final int STOP = 3;
  public static final int WAIT = 4;
  public static final int HALT = 5;
  public static final int TERM = 6;
  public static final String SPACE = "␣";

  public Instruction bindTo(LocalEnv parent, ClockImpl env);
  public int activate(Seq seq);
  public void activateAtEOI(Seq seq);
  public void reset();
  public String getAddress();
  public String toXML(String indent);
  public void notPurgeable();
}
```

L'algorithme choisi pour exécuter efficacement les instructions est similaire à l'algorithme «Thunderbolt» utilisé dans les **SugarCubes** v4. une méthode `activate()` implante pour chaque instruction l'équivalent des règles avec le terme $activ$ en tête de la sémantique formelle, tandis que la méthode `activateAtEOI()` implante les règles avec le terme eoi en tête. Par contre contrairement à la sémantique formelle les statuts retournés par l'activation d'une instruction sont au nombre de 6 :

- SUSP : indique que l'instruction doit être réactivée au cours de l'instant courant ;
- WEOI : indique que l'instruction doit être activée si l'événement sur lequel l'instruction est en attente est généré ou à défaut si la fin de l'instant est décidé sans que cet événement n'ait été généré ;
- STOP : indique que l'instruction a terminé son exécution pour l'instant courant et devra être réactivée à l'instant suivant ;
- WAIT : indique que l'instruction doit être activée uniquement si l'événement sur lequel l'instruction est en attente est généré ;
- HALT : indique que l'instruction ne doit plus jamais être activée mais n'a pas terminé pour autant ;
- TERM : indique que l'instruction a définitivement terminé son exécution.

L'utilisation de ces statuts, combinée à la gestion de files d'attente d'instructions associées aux différents événements de l'environnement d'exécution, permet une implantation efficace de la sémantique décrite en 3.4.

La méthode `reset()` permet de rétablir l'état d'une instruction à son état initial en cas de préemption.

Lorsque une instruction réactive est activée (rep. à la fin de l'instant) celle-ci est exécutée dans le contexte de l'instruction séquence parente qui l'a activée (il n'y a pas de séquence parente si cette instruction est placée au plus haut niveau de l'arborescence d'instruction représentant le programme). Ce contexte d'exécution séquentiel permet de mémoriser les générations de valeurs d'événements et d'agir directement sur l'avancement du programme sur une branche séquentielle.

Nous illustrons ce mécanisme par les quelques instruction suivantes :

- l'instruction `Nothing` :

```
package fr.cedric.sugarcubes.i;
import fr.cedric.sugarcubes.*;
final class Nothing extends InstructionImpl
{
...
  public int activate(final Seq seq){
    return TERM;
  }
...
}
```

  seule sa méthode `activate()` est importante et se contente de retourner le statut d'exécution `TERM`.

- l'instruction `Pause` :

```
package fr.cedric.sugarcubes.i;
import fr.cedric.sugarcubes.*;
final class Pause extends InstructionImpl
{
...
```

```
  public int activate(final Seq seq){
    seq.PC++;
    return STOP;
  }
...
}
```

seule sa méthode `activate()` est importante elle incrémente le pointeur d'instruction de la séquence englobante et retourne le statut `STOP`. L'état de l'instruction `Pause` (mémorisant pour l'instant suivant qu'une pause à l'instant courant a été effectuée) est en fait encodé dans la séquence n-aire englobante.

– l'instruction `Seq` :

```
package fr.cedric.sugarcubes.i;
import fr.cedric.sugarcubes.*;
final class Seq extends InstructionImpl
{
  Instruction p[];
  int PC = 0;
  ClockImpl env;
  Seq(){}
  Seq(final Instruction p[]){
    this.p = p;
  }
  public Instruction bindTo(final LocalEnv parent, final ClockImpl env){
    final Seq copy = new Seq();
    copy.p = new Instruction[this.p.length+1];
    for(int i = 0 ; i < p.length; i++){
      copy.p[i] = this.p[i].bindTo(parent, env);
    }
    copy.env = env;
    copy.p[this.p.length] = Nothing.NOTHING;
    return copy;
  }
  public int activate(final Seq seq){
    int res = p[PC].activate(this);
    while(TERM == res){
      PC++;
      if(PC == p.length){
        PC = 0;
        return TERM;
      }
      res = p[PC].activate(this);
    }
    return res;
  }
  public void activateAtEOI(final Seq seq){
    p[PC].activateAtEOI(this);
  }
...
}
```

La séquence est une instruction n-aire qui possède un tableau d'instructions à exécuter les unes après les autres dans l'ordre croissant de leur indice dans le tableau. La méthode `bindTo()` est appelée au moment de l'ajout effectif de l'instruction dans une machine d'exécution et permet de créer un clone correctement initialisé de l'instruction et de le lier au bon contexte d'exécution de type `ClockImpl`.
Lorsque la méthode `activate()` est activée, la séquence active l'instruction du tableau référencée par le compteur `PC`. Si l'instruction activée renvoie le statut `TERM` alors le compteur est incrémenté et la nouvelle instruction pointée par la séquence est appelée, sinon la séquence retourne le statut de la dernière instruction exécutée. Enfin, nous pouvons constater que la méthode `activateAtEOI()` se contente d'appeler la même méthode sur l'instruction actuellement pointée par le compteur `PC` dans le tableau d'instructions de la séquence.

– une action atomique de type `StringCell` est implantée par la classe `StringCellData`

```
public int activate(final Seq seq){
  if(!this.posted){
    env.post(this);
    this.posted = true;
  }
  return TERM;
}
```

La méthode `activate()` héritée de la classe `CellData` se contente d'enregistrer la cellule auprès de la machine d'exécution pour une exécution après la phase purement réactive. Ce code n'autorise qu'un seul enregistrement par instant. Lors de l'enregistrement de la cellule, la machine d'exécution choisit parmi un pool de threads un thread pour exécuter cette cellule au cours de la phase de calcul.
La classe elle-même correspond au code suivant :

```
class StringCellData extends InnerCellData implements StringCell
{
  private JavaStringComputation computation;
  private volatile String value;
  private String nextValue;
  private String name;
  StringCellData(final String name, final String initValue, final
      JavaStringComputation computation){
    this.value = this.nextValue = initValue;
    this.computation = computation;
    this.name = name;
  }
  void computeValue(final EnvironmentImpl e){
    nextValue = computation.evaluate(e, value);
  }
  void swapValue(){
    this.value = nextValue;
    this.posted = false;
  }
  public String getStringValue(){
    return this.value;
  }
    ...
}
```

La méthode `computeValue()` appelée par le thread d'exécution de la machine réactive pendant la phase de calcul. Cette méthode permet de calculer une nouvelle valeur pour la cellule. (methode `evaluate()`) cette valeur est stockée dans une variable temporaire (`nextValue`) avant d'être recopiée comme nouvelle valeur de la cellule pendant la phase de recopie correspondant à l'appel de la méthode `swapValues()`.

5.5 le support OpenCL

En s'appuyant sur la libraire JOCL, nous proposons une extension des **SugarCubes** utilisant OpenCL. Cette extension est activée au chargement du framework **SugarCubes**. Pour ce faire 2 conditions sont nécessaires :

- disposer d'un support Open CL sur le matériel utilisé ;
- disposer aussi d'un accès à la librairie JOCL[**?**] correctement installée.

La librairie Java JOCL est une bibliothèque de programmation implantant un pont entre les API natives d'OpenCL généralement mise en œuvre à travers des programmes écrits en langage C et des appels à des fonctions natives en Java [2].

Lorsque la classe `SC` est chargée pour la première fois, l'environnement **SugarCubes** vérifie la disponibilité du support OpenCL. Si matériel et logiciel correspondants sont effectivement disponibles, un profile du matériel est réalisé et les ressources sont initialisées. Il est possible de modifier des éléments du profile OpenCL retenu par l'implantation des SugarCubes en utilisant le mécanisme des propriétés Java couramment utilisées pour des solutions de paramétrisation.

2. Au moment où nous nous sommes lancé dans un support OpenCL, il n'y avait pas encore de support officiel dans la version de Java utilisée.

Une fois les ressources initialisées, il devient alors possible de définir de nouveaux types d'actions atomiques : `OCLFloatCell`

Si ces actions atomiques utilisent le même principe de fonctionnement que les actions 100% java, leur mise en œuvre est un peu différente puisqu'au lieu de déclarer une une classe implantant une interface de type `Java*Computation` pour le calcul des nouvelles valeurs, nous allons utiliser des objets de types `OCL*Kernel`. Ces «kernels» permettent de définir le calcul des nouvelles valeurs comme étant le calcul d'un kernel OpenCL. Pour être efficace, le support OpenCL suppose que les objets de type `OCLKernel` sont partagés par un grand nombre de cellules à rafraîchir à chaque instant.
La définition d'un kernel est indépendante des cellules qui le manipulent :

```
final static OCLFloatKernel ocl_fk_inc = SC.openCLFloatKernel("  self += 1;
    ", null, null, 1);
```

Un kernel OpenCL en SugarCubes est créée par la méthode correspondante de la classe `SC`. La chaine de caractères passée en premier argument représente le code du kernel exprimée dans un langage très proche du langage OpenCL standard. Cependant, de légères modifications permettent de faciliter le lien entre les kernels OpenCL et les programmes réactifs :

- Un kernel OpenCL en **SugarCubes** est gérée par une unique fonction et on ne définit dès lors que le corps de cette fonction. On ne définit donc pas l'interface de la fonction qui est automatiquement générée par l'environnement **SugarCubes**.
- Un kernel Open CL en **SugarCubes** possède une variable prédéfinie `self` qui représente la valeur traitée par le kernel OpenCL. Cette valeur est accessible en lecture et en écriture. En général, le kernel OpenCL calcul une nouvelle valeur pour cette variable. Dans l'exemple ci-dessus le kernel incrémente la valeur `self`.
- Le deuxième argument permet de réaliser un lien entre des variables manipulées dans le kernel et des cellules mémoires du système réactif. Dans notre cas, la valeur nulle indique qu'il n'y a pas de lien avec d'autres cellules.
- Le troisième argument permet de réaliser passer les valeurs associées à des événements **SugarCubes** sous la formes de tableaux de valeurs exploitables par le kernel. Dans notre cas, la valeur nulle indique que le kernel n'accède pas aux valeurs de l'environnement d'événements **SugarCubes**.

Le kernel n'est pas une cellule mais une fonction permettant de calculer de nouvelles valeurs. Il faut donc créer des cellules mémoire tirant partie de ce kernel :

```
class MyCube extends Cube
{
  final FloatCell test;
  MyCube(){
    this.test = this.openCLFloatCell("test", 1, ocl_fk_inc, null);
  }...
}
```

Pour ce faire on utilise une méthode idoine de la classe `Cube`. Le dernier argument permet de lier les variables à des cellules particulière pour permettre le calcul de la nouvelle valeur de la cellule. Dans notre cas il n'y a pas de lien, donc la valeur de ce quatrième argument est nulle. Finalement, la mise en œuvre dans un programme réactif reste similaire à celle des autres cellules :

```
    machine.addProgram(SC.loop(this.test));
```

Considérons la mise en œuvre d'un kernel permettant d'intégrer une valeur numérique :

```
final static OCLFloatKernel oclfk_plus = SC.openCLFloatKernel("self += dt*x;"
                                        , new OCLKernelArg[]{
                                          SC.oclFloatCellArg("x")
                                          , SC.oclFloatCellArg("dt")
                                        }
                                        , null, 0);
```

Ce kernel permet d'intégrer selon la méthode d'Euler une grandeur d'état `x` sur un pas de temps `dt`. Les valeurs `x` et `dt` sont déclarées dans le code sources du kernel automatiquement par le support SugarCubes, tandis que le tableau `OCLKernelArg` crée les poignées nécessaire pour lier ces valeurs à différentes cellules mémoire. Le lien est réalisé à la création d'une cellule :

```
    x = this.openCLFloatCell("x", (float)ix, oclfk_plus
                            , new OCLKernelBinding[]{
                              new OCLKernelBinding(1){public Object get()
                                  {return sx;}}
                              , new OCLKernelBinding(1){public Object get
                                  (){return dt;}}
                            });
```

Le tableau de type `OCLKernelBinding` permet de lier les variables manipulées par le kernel à des valeurs de cellules mémoire. Dans notre cas la variable `dt` est liée à une cellule mémoire contenant qui définit le pas de temps d'intégration temporelle. Tandis que le calcul de la nouvelle valeur de la cellule `x` est obtenue par intégration de la valeur de la cellule `sx` définie par ailleurs.

Pour finir, considérons le kernel suivant, permettant de calculer la force s'exerçant sur l'axe des x, d'un corps ponctuel en influence gravitationnel avec un ensemble d'autres corps. Chaque corps ponctuel du système émet à chaque instant sa position et sa masse comme valeur associée à un ensemble de 4 événements : `HERE_X`, `HERE_Y`, `HERE_Z`, `HERE_M`.

```
  final static OCLFloatKernel oclfk_gravx = SC.openCLFloatKernel("self␣=␣0;\n
      "
                                          +"␣␣for(int␣i␣=␣0;␣i␣<␣px1_length;␣
                                              i++){\n"
                                          +"␣␣␣␣float␣rx␣=␣px0-px1[i],␣ry␣=␣
                                              py0-py1[i],␣rz␣=␣pz0-pz1[i];\n"
                                          +"␣␣␣␣float␣r2␣=␣rx*rx+ry*ry+rz*rz
                                              ;\n"
                                          +"␣␣␣␣r2␣=␣(1e6␣>␣r2)?1e15:r2;\n"
                                          +"␣␣␣␣self␣-=␣6.673e-11␣*␣M[i]␣*␣rx
                                              ␣/␣(r2␣*␣sqrt(r2));\n"
                                          +"␣␣}\n"
                                          , new OCLKernelArg[]{
                                            SC.oclFloatCellArg("px0")
                                            , SC.oclFloatCellArg("py0")
                                            , SC.oclFloatCellArg("pz0")
                                            , SC.oclFloatCellArg("m")
                                           }
                                          , new OCLKernelEvent[]{
                                            SC.oclFloatEvent("px1",HERE_X)
                                            , SC.oclFloatEvent("py1",HERE_Y)
                                            , SC.oclFloatEvent("pz1",HERE_Z)
                                            , SC.oclFloatEvent("M",HERE_M)
                                           }
                                          , 0
                                         );
```

Le calcul de la résultante en x de la force exercé par l'ensemble des corps ponctuels du système sur le corps ponctuel considéré est obtenu par itération sur l'ensemble des valeurs associées aux événements `HERE_*`. Ainsi à chaque événement correspond un tableau de valeurs dans manipulable dans le kernel. Ce tableau récupère les valeurs associée à un événement par l'entremise d'un objet de type `OCLKernelEvent`. Cet objet associe un nom de tableau qui sera déclaré dans le code du kernel à un identifiant d'événement **SugarCubes**. Ce lien est indépendant des cellules utilisant le kernel et est donc réalisé une fois pour toute. L'environnement d'exécution **SugarCubes** prend en charge les différents transferts mémoire entre le système réactif et les processeur dédiés au traitement OpenCL. Enfin, comme OpenCL est un dialecte de C, il existe également une variable associée de même nom que le tableau avec le suffixe `_length` permettant de connaitre la taille du tableau.

6 Conclusion

Nous avons présenté dans ce document les principes qui nous ont présidés à la définition du modèle de calcul des **SugarCubes** v5. Sur ce modèle de calcul nous avons retenu un certain nombre d'instructions permettant d'exprimer

des systèmes réactifs et nous avons formalisé leur sémantique. À partir de la définition formelle de la sémantique, nous avons dérivées de manière automatique une implantation de référence, une documentation en ligne, un fichier d'entrée pour un outil de preuve automatique (CIME) et ce document lui-même. Nous avons pu fournir la sémantique formelle du langage à des outils de preuves automatiques pour certifier la correction des systèmes réactifs au regard de 3 propriétés : la cohérence, le déterminisme, la terminaison de instants. Ce travail est encore incomplet mais produit des résultats encourageants. Enfin nous avons présenté une implantation efficace de ce système, comprenant également un support matériel particulier à base de GPU, s'appuyant sur la technologie OpenCL.

Les travaux de preuve de correction sont sur le point d'aboutir à une certification dans l'assistant de preuve Coq. À l'instar du projet COMPCERT nous envisageons de poursuivre nos travaux sur la formalisation et la preuve de propriétés de la sémantique du langage des **SugarCubes** par une modélisation complète en Coq et la possibilité d'extraire de façon automatique l'implantation depuis cette modélisation.

Nous continuons le développement du support d'OpenCL pour le calcul scientifique et en particulier la simulation de la dynamique moléculaire. Nous envisageons d'élargir notre approche à d'autres formes de mise en œuvre du calcul parallèle et notamment au support de grilles de calcul avec une adaptation du modèle de calcul à des technologies telles que MPI.

Enfin, nous poursuivons nos efforts sur l'optimisation des performances des **SugarCubes** à travers la mise en œuvre d'une machine virtuelle dédiée à l'exécution du code **SugarCubes** écrite en C et permettant une gestion plus fine de l'allocation mémoire que ne permet pas Java dans sa version standard.

A Annexe A

clock

$$\frac{\mathit{instant}\,(t,E,M) \to \mathit{prod}\,(t',P,M') \qquad \mathit{Run}\,(t',M') \to u}{\mathit{Run}\,(t,M) \to u} \tag{A.1}$$

$$\frac{\mathit{instant}\,(t,E,M) \to \bot\,(E',M')}{\mathit{Run}\,(t,M) \to \bot\,(E',M')} \tag{A.2}$$

$$\frac{\mathit{activ}\,(\mathit{close}\,(t)\,,E,M) \to \texttt{STOP}\,(t',P,A)}{\mathit{instant}\,(t,E,M) \to \mathit{prod}\,(t',E \oplus P, \mathit{swap}\,(\mathit{exec}\,(A,M)))} \tag{A.3}$$

$$\frac{\mathit{activ}\,(\mathit{close}\,(t)\,,E,M) \to \texttt{TERM}\,(P,A)}{\mathit{instant}\,(t,E,M) \to \bot\,(E \oplus P, \mathit{swap}\,(\mathit{exec}\,(A,M)))} \tag{A.4}$$

$$\frac{\mathit{activ}\,(t,E,M) \to \texttt{TERM}\,(P,A)}{\mathit{activ}\,(\mathit{close}\,(t)\,,E,M) \to \texttt{TERM}\,(P,A)} \tag{A.5}$$

$$\frac{\mathit{activ}\,(t,E,M) \to \texttt{STOP}\,(t',P,A)}{\mathit{activ}\,(\mathit{close}\,(t)\,,E,M) \to \texttt{STOP}\,(t',P,A)} \tag{A.6}$$

$$\frac{\mathit{activ}\,(t,E,M) \to \texttt{SUSP}\,(t',P,A) \qquad P \neq \emptyset \qquad \mathit{activ}\,(\mathit{close}\,(t')\,,E \oplus P,M) \to \texttt{TERM}\,(P',A')}{\mathit{activ}\,(\mathit{close}\,(t)\,,E,M) \to \texttt{TERM}\,(P \oplus P', A \bigcup A')} \tag{A.7}$$

$$\frac{\mathit{activ}\,(t,E,M) \to \texttt{SUSP}\,(t',P,A) \qquad P \neq \emptyset \qquad \mathit{activ}\,(\mathit{close}\,(t')\,,E \oplus P,M) \to \texttt{STOP}\,(t'',P',A')}{\mathit{activ}\,(\mathit{close}\,(t)\,,E,M) \to \texttt{STOP}\,(t'',P \oplus P', A \bigcup A')} \tag{A.8}$$

$$\frac{\mathit{activ}\,(t,E,M) \to \texttt{SUSP}\,(t',P,A) \qquad P = \emptyset}{\mathit{activ}\,(\mathit{close}\,(t)\,,E,M) \to \texttt{STOP}\,(\mathit{eoi}\,(t',E)\,,\emptyset,A)} \tag{A.9}$$

$$\frac{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{SUSP}\left(par\left(l'_{\texttt{SUSP}},l'_{\texttt{STOP}}\right),P,A\right)}{activ\left(par\left(l_{\texttt{SUSP}},l_{\texttt{STOP}}\right),E,M\right)\rightarrow \texttt{SUSP}\left(par\left(l'_{\texttt{SUSP}},l_{\texttt{STOP}}\bullet l'_{\texttt{STOP}}\right),P,A\right.} \quad \text{(A.10)}$$

$$\frac{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{STOP}\left(par\left(nil,l'_{\texttt{STOP}}\right),P,A\right)}{activ\left(par\left(l_{\texttt{SUSP}},l_{\texttt{STOP}}\right),E,M\right)\rightarrow \texttt{STOP}\left(par\left(l_{\texttt{STOP}}\bullet l'_{\texttt{STOP}},nil\right),P,A\right.} \quad \text{(A.11)}$$

$$\frac{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{TERM}\left(P,A\right) \qquad l_{\texttt{STOP}}\neq n}{activ\left(par\left(l_{\texttt{SUSP}},l_{\texttt{STOP}}\right),E,M\right)\rightarrow \texttt{STOP}\left(par\left(l_{\texttt{STOP}},nil\right),P,A\right.} \quad \text{(A.12)}$$

$$\frac{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{TERM}\left(P,A\right) \qquad l_{\texttt{STOP}}= n}{activ\left(par\left(l_{\texttt{SUSP}},l_{\texttt{STOP}}\right),E,M\right)\rightarrow \texttt{TERM}\left(P,A\right)} \quad \text{(A.13)}$$

$$\frac{l_{\texttt{SUSP}}\neq nil \qquad activ\left(head\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{SUSP}\left(t',P,A\right) \qquad activ\left(par\_run\left(tail\left(l_{\texttt{SUSP}}\right)\right),E,M\right)\rightarrow \texttt{SUSP}\left(par\left(l'_{\texttt{SUSP}},l'_{\texttt{STOP}}\right),P',A\right.}{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{SUSP}\left(par\left(\left[t',l'_{\texttt{SUSP}}\right],l'_{\texttt{STOP}}\right),P\oplus P',A\bigcup A'\right)} \quad \text{(A.14)}$$

$$\frac{l_{\texttt{SUSP}}\neq nil \qquad activ\left(head\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{SUSP}\left(t',P,A\right) \qquad activ\left(par\_run\left(tail\left(l_{\texttt{SUSP}}\right)\right),E,M\right)\rightarrow \texttt{STOP}\left(par\left(nil,l'_{\texttt{STOP}}\right),P',A\right.}{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{SUSP}\left(par\left(\left[t',nil\right],l'_{\texttt{STOP}}\right),P\oplus P',A\bigcup A'\right)} \quad \text{(A.15)}$$

$$\frac{l_{\texttt{SUSP}}\neq nil \qquad activ\left(head\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{SUSP}\left(t',P,A\right) \qquad activ\left(par\_run\left(tail\left(l_{\texttt{SUSP}}\right)\right),E,M\right)\rightarrow \texttt{TERM}\left(P',A\right.}{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{SUSP}\left(par\left(\left[t',nil\right],nil\right),P\oplus P',A\bigcup A'\right)} \quad \text{(A.16)}$$

$$\frac{l_{\texttt{SUSP}}\neq nil \qquad activ\left(head\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{STOP}\left(t',P,A\right) \qquad activ\left(par\_run\left(tail\left(l_{\texttt{SUSP}}\right)\right),E,M\right)\rightarrow \texttt{SUSP}\left(par\left(l'_{\texttt{SUSP}},l'_{\texttt{STOP}}\right),P',A\right.}{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{SUSP}\left(par\left(l'_{\texttt{SUSP}},\left[t',l'_{\texttt{STOP}}\right]\right),P\oplus P',A\bigcup A'\right)} \quad \text{(A.17)}$$

$$\frac{l_{\texttt{SUSP}}\neq nil \qquad activ\left(head\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{STOP}\left(t',P,A\right) \qquad activ\left(par\_run\left(tail\left(l_{\texttt{SUSP}}\right)\right),E,M\right)\rightarrow \texttt{STOP}\left(par\left(nil,l'_{\texttt{STOP}}\right),P',A\right.}{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{STOP}\left(par\left(nil,\left[t',l'_{\texttt{STOP}}\right]\right),P\oplus P',A\bigcup A'\right)} \quad \text{(A.18)}$$

$$\frac{l_{\texttt{SUSP}}\neq nil \qquad activ\left(head\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{STOP}\left(t',P,A\right) \qquad activ\left(par\_run\left(tail\left(l_{\texttt{SUSP}}\right)\right),E,M\right)\rightarrow \texttt{TERM}\left(P',A\right.}{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{STOP}\left(par\left(nil,\left[t',nil\right]\right),P\oplus P',A\bigcup A'\right)} \quad \text{(A.19)}$$

$$\frac{l_{\texttt{SUSP}}\neq nil \qquad activ\left(head\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{TERM}\left(P,A\right) \qquad activ\left(par\_run\left(tail\left(l_{\texttt{SUSP}}\right)\right),E,M\right)\rightarrow \texttt{SUSP}\left(par\left(l'_{\texttt{SUSP}},l'_{\texttt{STOP}}\right),P',A\right.}{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{SUSP}\left(par\left(l'_{\texttt{SUSP}},l'_{\texttt{STOP}}\right),P\oplus P',A\bigcup A'\right)} \quad \text{(A.20)}$$

$$\frac{l_{\texttt{SUSP}}\neq nil \qquad activ\left(head\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{TERM}\left(P,A\right) \qquad activ\left(par\_run\left(tail\left(l_{\texttt{SUSP}}\right)\right),E,M\right)\rightarrow \texttt{STOP}\left(par\left(nil,l'_{\texttt{STOP}}\right),P',A\right.}{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{STOP}\left(par\left(nil,l'_{\texttt{STOP}}\right),P\oplus P',A\bigcup A'\right)} \quad \text{(A.21)}$$

$$\frac{l_{\texttt{SUSP}}\neq nil \qquad activ\left(head\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{TERM}\left(P,A\right) \qquad activ\left(par\_run\left(tail\left(l_{\texttt{SUSP}}\right)\right),E,M\right)\rightarrow \texttt{TERM}\left(P',A\right.}{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{TERM}\left(P\oplus P',A\bigcup A'\right)} \quad \text{(A.22)}$$

$$\frac{l_{\texttt{SUSP}}= nil}{activ\left(par\_run\left(l_{\texttt{SUSP}}\right),E,M\right)\rightarrow \texttt{TERM}\left(\emptyset,\emptyset\right.} \quad \text{(A.23)}$$

$$\frac{eoi\left(par\_run\left(l_{\texttt{SUSP}}\right),E\right)\rightarrow par\left(l'_{\texttt{SUSP}},nil\right)}{eoi\left(par\left(l_{\texttt{SUSP}},l_{\texttt{STOP}}\right),E\right)\rightarrow par\left(l'_{\texttt{SUSP}}\bullet l_{\texttt{STOP}},ni\right.} \quad \text{(A.24)}$$

$$\frac{l_{\texttt{SUSP}}\neq nil \qquad eoi\left(par\_run\left(tail\left(l_{\texttt{SUSP}}\right)\right),E\right)\rightarrow par\left(l',ni\right.}{eoi\left(par\_run\left(l_{\texttt{SUSP}}\right),E\right)\rightarrow par\left(\left[eoi\left(head\left(l_{\texttt{SUSP}}\right),E\right),l'\right],ni\right.} \quad \text{(A.25)}$$

$$\frac{l_{\texttt{SUSP}}= nil}{eoi\left(par\_run\left(l_{\texttt{SUSP}}\right),E\right)\rightarrow par\left(nil,ni\right.} \quad \text{(A.26)}$$

;

$$\frac{l \neq nil \qquad activ\left(head\left(l\right), E, M\right) \rightarrow \texttt{SUSP}\left(t', P, A\right)}{activ\left(seq\left(l\right), E, M\right) \rightarrow \texttt{SUSP}\left(seq\left(\left[t', tail\left(l\right)\right]\right), P, A\right)} \tag{A.27}$$

$$\frac{l \neq nil \qquad activ\left(head\left(l\right), E, M\right) \rightarrow \texttt{STOP}\left(t', P, A\right)}{activ\left(seq\left(l\right), E, M\right) \rightarrow \texttt{STOP}\left(seq\left(\left[t', tail\left(l\right)\right]\right), P, A\right)} \tag{A.28}$$

$$\frac{l \neq nil \qquad activ\left(head\left(l\right), E, M\right) \rightarrow \texttt{TERM}\left(P, A\right) \qquad activ\left(seq\left(tail\left(l\right)\right), E \oplus P, M\right) \rightarrow \texttt{SUSP}\left(seq\left(l'\right), P', A'\right)}{activ\left(seq\left(l\right), E, M\right) \rightarrow \texttt{SUSP}\left(seq\left(l'\right), P \oplus P', A \bigcup A'\right)} \tag{A.29}$$

$$\frac{l \neq nil \qquad activ\left(head\left(l\right), E, M\right) \rightarrow \texttt{TERM}\left(P, A\right) \qquad activ\left(seq\left(tail\left(l\right)\right), E \oplus P, M\right) \rightarrow \texttt{STOP}\left(seq\left(l'\right), P', A'\right)}{activ\left(seq\left(l\right), E, M\right) \rightarrow \texttt{STOP}\left(seq\left(l'\right), P \oplus P', A \bigcup A'\right)} \tag{A.30}$$

$$\frac{l \neq nil \qquad activ\left(head\left(l\right), E, M\right) \rightarrow \texttt{TERM}\left(P, A\right) \qquad activ\left(seq\left(tail\left(l\right)\right), E \oplus P, M\right) \rightarrow \texttt{TERM}\left(P', A'\right)}{activ\left(seq\left(l\right), E, M\right) \rightarrow \texttt{TERM}\left(P \oplus P', A \bigcup A'\right)} \tag{A.31}$$

$$\frac{l = nil}{activ\left(seq\left(l\right), E, M\right) \rightarrow \texttt{TERM}\left(\emptyset, \emptyset\right)} \tag{A.32}$$

$$\frac{l \neq nil}{eoi\left(seq\left(l\right), E\right) \rightarrow seq\left(\left[eoi\left(head\left(l\right), E\right), tail\left(l\right)\right]\right)} \tag{A.33}$$

loop

$$\frac{activ\left(par\left(\left[t, \left[pause, nil\right]\right], nil\right), E, M\right) \rightarrow \texttt{SUSP}\left(par\left(\left[t', nil\right], \left[nothing, nil\right]\right), P, A\right)}{activ\left(loop\left(t\right), E, M\right) \rightarrow \texttt{SUSP}\left(seq\left(\left[par\left(\left[t', nil\right], \left[nothing, nil\right]\right), \left[loop\left(t\right), nil\right]\right]\right), P, A\right)} \tag{A.34}$$

$$\frac{activ\left(par\left(\left[t, \left[pause, nil\right]\right], nil\right), E, M\right) \rightarrow \texttt{STOP}\left(par\left(l'_{\texttt{SUSP}}, nil\right), P, A\right)}{activ\left(loop\left(t\right), E, M\right) \rightarrow \texttt{STOP}\left(seq\left(\left[par\left(l'_{\texttt{SUSP}}, nil\right), \left[loop\left(t\right), nil\right]\right]\right), P, A\right)} \tag{A.35}$$

repeat

$$\frac{n > 0 \qquad activ\left(t, E, M\right) \rightarrow \texttt{SUSP}\left(t', P, A\right)}{activ\left(repeat\_n\_times\left(n, t\right), E, M\right) \rightarrow \texttt{SUSP}\left(seq\left(\left[t', \left[repeat\_n\_times\left(n-1, t\right), nil\right]\right]\right), P, A\right)} \tag{A.36}$$

$$\frac{n > 0 \qquad activ\left(t, E, M\right) \rightarrow \texttt{STOP}\left(t', P, A\right)}{activ\left(repeat\_n\_times\left(n, t\right), E, M\right) \rightarrow \texttt{STOP}\left(seq\left(\left[t', \left[repeat\_n\_times\left(n-1, t\right), nil\right]\right]\right), P, A\right)} \tag{A.37}$$

$$\frac{n > 0 \qquad activ\left(t, E, M\right) \rightarrow \texttt{TERM}\left(P, A\right) \qquad activ\left(repeat\_n\_times\left(n-1, t\right), E \oplus P, M\right) \rightarrow \texttt{SUSP}\left(u, P', A'\right)}{activ\left(repeat\_n\_times\left(n, t\right), E, M\right) \rightarrow \texttt{SUSP}\left(u, P \oplus P', A \bigcup A'\right)} \tag{A.38}$$

$$\frac{n > 0 \qquad activ\left(t, E, M\right) \rightarrow \texttt{TERM}\left(P, A\right) \qquad activ\left(repeat\_n\_times\left(n-1, t\right), E \oplus P, M\right) \rightarrow \texttt{STOP}\left(u, P', A'\right)}{activ\left(repeat\_n\_times\left(n, t\right), E, M\right) \rightarrow \texttt{STOP}\left(u, P \oplus P', A \bigcup A'\right)} \tag{A.39}$$

$$\frac{n > 0 \qquad activ\left(t, E, M\right) \rightarrow \texttt{TERM}\left(P, A\right) \qquad activ\left(repeat\_n\_times\left(n-1, t\right), E \oplus P, M\right) \rightarrow \texttt{TERM}\left(P', A'\right)}{activ\left(repeat\_n\_times\left(n, t\right), E, M\right) \rightarrow \texttt{TERM}\left(P \oplus P', A \bigcup A'\right)} \tag{A.40}$$

$$\frac{n = 0}{activ\left(repeat\_n\_times\left(n, t\right), E, M\right) \rightarrow \texttt{TERM}\left(\emptyset, \emptyset\right)} \tag{A.41}$$

control

$$\frac{v \notin E}{\mathit{activ}\left(\mathit{control}\left(v,t\right),E,M\right) \rightarrow \texttt{SUSP}\left(\mathit{control}\left(v,t\right),\emptyset,\emptyset\right)} \tag{A.42}$$

$$\frac{v \in E \qquad \mathit{activ}\left(t,E,M\right) \rightarrow \texttt{SUSP}\left(t',P,A\right)}{\mathit{activ}\left(\mathit{control}\left(v,t\right),E,M\right) \rightarrow \texttt{SUSP}\left(\mathit{control\_run}\left(v,t'\right),P,A\right)} \tag{A.43}$$

$$\frac{v \in E \qquad \mathit{activ}\left(t,E,M\right) \rightarrow \texttt{STOP}\left(t',P,A\right)}{\mathit{activ}\left(\mathit{control}\left(v,t\right),E,M\right) \rightarrow \texttt{STOP}\left(\mathit{control}\left(v,t'\right),P,A\right)} \tag{A.44}$$

$$\frac{\mathit{activ}\left(t,E,M\right) \rightarrow \texttt{SUSP}\left(t',P,A\right)}{\mathit{activ}\left(\mathit{control\_run}\left(v,t\right),E,M\right) \rightarrow \texttt{SUSP}\left(\mathit{control\_run}\left(v,t'\right),P,A\right)} \tag{A.45}$$

$$\frac{\mathit{activ}\left(t,E,M\right) \rightarrow \texttt{STOP}\left(t',P,A\right)}{\mathit{activ}\left(\mathit{control\_run}\left(v,t\right),E,M\right) \rightarrow \texttt{STOP}\left(\mathit{control}\left(v,t'\right),P,A\right)} \tag{A.46}$$

$$\frac{v \in E \qquad \mathit{activ}\left(t,E,M\right) \rightarrow \texttt{TERM}\left(P,A\right)}{\mathit{activ}\left(\mathit{control}\left(v,t\right),E,M\right) \rightarrow \texttt{TERM}\left(P,A\right)} \tag{A.47}$$

$$\frac{\mathit{activ}\left(t,E,M\right) \rightarrow \texttt{TERM}\left(P,A\right)}{\mathit{activ}\left(\mathit{control\_run}\left(v,t\right),E,M\right) \rightarrow \texttt{TERM}\left(P,A\right)} \tag{A.48}$$

$$\mathit{eoi}\left(\mathit{control\_run}\left(v,t\right),E\right) \rightarrow \mathit{control}\left(v,\mathit{eoi}\left(t,E\right)\right) \tag{A.49}$$

$$\mathit{eoi}\left(\mathit{control}\left(v,t\right),E\right) \rightarrow \mathit{control}\left(v,t\right) \tag{A.50}$$

kill

$$\frac{\mathit{activ}\left(t,E,M\right) \rightarrow \texttt{SUSP}\left(t',P,A\right)}{\mathit{activ}\left(\mathit{kill}\left(v,t,u\right),E,M\right) \rightarrow \texttt{SUSP}\left(\mathit{kill}\left(v,t',u\right),P,A\right)} \tag{A.51}$$

$$\frac{\mathit{activ}\left(t,E,M\right) \rightarrow \texttt{TERM}\left(P,A\right)}{\mathit{activ}\left(\mathit{kill}\left(v,t,u\right),E,M\right) \rightarrow \texttt{TERM}\left(P,A\right)} \tag{A.52}$$

$$\frac{\mathit{activ}\left(t,E,M\right) \rightarrow \texttt{STOP}\left(t',P,A\right)}{\mathit{activ}\left(\mathit{kill}\left(v,t,u\right),E,M\right) \rightarrow \texttt{SUSP}\left(\mathit{kill\_stop}\left(v,t',u\right),P,A\right)} \tag{A.53}$$

$$\mathit{activ}\left(\mathit{kill\_stop}\left(v,t,u\right),E,M\right) \rightarrow \texttt{SUSP}\left(\mathit{kill\_stop}\left(v,t,u\right),\emptyset,\emptyset\right) \tag{A.54}$$

$$\frac{v \in E}{\mathit{eoi}\left(\mathit{kill}\left(v,t,u\right),E\right) \rightarrow u} \tag{A.55}$$

$$\frac{v \notin E}{\mathit{eoi}\left(\mathit{kill}\left(v,t,u\right),E\right) \rightarrow \mathit{kill}\left(v,\mathit{eoi}\left(t,E\right),u\right)} \tag{A.56}$$

$$\frac{v \in E}{\mathit{eoi}\left(\mathit{kill\_stop}\left(v,t,u\right),E\right) \rightarrow u} \tag{A.57}$$

$$\frac{v \notin E}{\mathit{eoi}\left(\mathit{kill\_stop}\left(v,t,u\right),E\right) \rightarrow \mathit{kill}\left(v,t,u\right)} \tag{A.58}$$

when

$$\frac{v \in E \qquad \mathit{activ}\left(t,E,M\right) \rightarrow \texttt{SUSP}\left(t',P,A\right)}{\mathit{activ}\left(\mathit{when}\left(v,t,u\right),E,M\right) \rightarrow \texttt{SUSP}\left(t',P,A\right)} \tag{A.59}$$

$$\frac{v \in E \qquad \mathit{activ}\left(t,E,M\right) \rightarrow \texttt{STOP}\left(t',P,A\right)}{\mathit{activ}\left(\mathit{when}\left(v,t,u\right),E,M\right) \rightarrow \texttt{STOP}\left(t',P,A\right)} \tag{A.60}$$

$$\frac{v \in E \qquad \mathit{activ}\left(t,E,M\right) \rightarrow \texttt{TERM}\left(P,A\right)}{\mathit{activ}\left(\mathit{when}\left(v,t,u\right),E,M\right) \rightarrow \texttt{TERM}\left(P,A\right)} \tag{A.61}$$

$$\frac{v \notin E}{\mathit{activ}\left(\mathit{when}\left(v,t,u\right),E,M\right) \rightarrow \texttt{SUSP}\left(\mathit{when}\left(v,t,u\right),\emptyset,\emptyset\right)} \tag{A.62}$$

$$\mathit{eoi}\left(\mathit{when}\left(v,t,u\right),E\right) \rightarrow u \tag{A.63}$$

await

$$\frac{v \in E}{\mathit{activ}\left(\mathit{await}\left(v\right), E, M\right) \rightarrow \texttt{TERM}\left(\emptyset, \emptyset\right)} \tag{A.64}$$

$$\frac{v \notin E}{\mathit{activ}\left(\mathit{await}\left(v\right), E, M\right) \rightarrow \texttt{SUSP}\left(\mathit{await}\left(v\right), \emptyset, \emptyset\right)} \tag{A.65}$$

$$\mathit{eoi}\left(\mathit{await}\left(v\right), E\right) \rightarrow \mathit{await}\left(v\right) \tag{A.66}$$

pause for ever

$$\mathit{activ}\left(\mathit{pause\_for\_ever}, E, M\right) \rightarrow \texttt{STOP}\left(\mathit{pause\_for\_ever}, \emptyset, \emptyset\right) \tag{A.67}$$

pause n times

$$\frac{n > 0}{\mathit{activ}\left(\mathit{pause\_n\_times}\left(n\right), E, M\right) \rightarrow \texttt{STOP}\left(\mathit{pause\_n\_times}\left(n-1\right), \emptyset, \emptyset\right)} \tag{A.68}$$

$$\frac{n = 0}{\mathit{activ}\left(\mathit{pause\_n\_times}\left(n\right), E, M\right) \rightarrow \texttt{TERM}\left(\emptyset, \emptyset\right)} \tag{A.69}$$

pause

$$\mathit{activ}\left(\mathit{pause}, E, M\right) \rightarrow \texttt{STOP}\left(\mathit{nothing}, \emptyset, \emptyset\right) \tag{A.70}$$

action

$$\mathit{activ}\left(\mathit{action}\left(v\right), E, M\right) \rightarrow \texttt{TERM}\left(\emptyset, \{v\}\right) \tag{A.71}$$

generate

$$\mathit{activ}\left(\mathit{generate}\left(v\right), E, M\right) \rightarrow \texttt{TERM}\left(\{v\}, \emptyset\right) \tag{A.72}$$

nothing

$$\mathit{activ}\left(\mathit{nothing}, E, M\right) \rightarrow \texttt{TERM}\left(\emptyset, \emptyset\right) \tag{A.73}$$

B Annexe B

```
#Set_sat_solver "minisat";
#Set_nb_proc "4";

let F = signature "true:0; false:0; nil:0; nop:0; void:0; activ:3; SUSP:3; STOP:3;
    TERM:2; eoi:2; peoi:2; swap:1; exec:2; nothing:0; close:1; instant:3; prod:3; End
    :2; pause:0; pause_n_times:1; pause_for_ever:0; repeat_n_times:2; seq:1; par:2;
    par_run:1; action:1; cons:2; await:1; control_run:2; generate:1; loop:1; Run:2;
    notin:2; in:2; eq:1; when:3; kill:3; control:2; kill_stop:3; union:2; succ:1;
    isVoid:1; isEmpty:1; e:0; f:0; g:0; a:0; b:0; c:0; cat:2; zero:0; notNil:1; head:1;
    tail:1; is:2; union_1:3; in_1:3; gt0:1; U_activ_await_1:4; U_activ_close_1:4;
    U_activ_close_2:5; U_activ_close_3:6; U_activ_control_1:4; U_activ_control_2
    :5; U_activ_control_run_1:4; U_activ_instant_1:4; U_activ_kill_1:4;
    U_activ_loop_1:4; U_activ_par_1:4; U_activ_par_2:5; U_activ_par_run_1:4;
    U_activ_par_run_2:5; U_activ_par_run_3:6; U_activ_pause_n_times_1:4;
    U_activ_repeat_n_times_1:4; U_activ_repeat_n_times_2:5;
    U_activ_repeat_n_times_3:6; U_activ_seq_1:4; U_activ_seq_2:5; U_activ_seq_3
    :6; U_activ_when_1:4; U_activ_when_2:5; U_eoi_kill_1:3; U_eoi_kill_stop_1:3;
    U_eoi_par_1:3; U_eoi_par_run_1:3; U_eoi_par_run_2:4; U_eoi_seq_1:3;";
let X = variables "A,M,P,E,alpha,t,u,v,A_,M_,P_,A__,M__,t_,u_,t__,l,n,l_,
    l_SUSP,l_STOP,l__SUSP,l__STOP";
let T = algebra F;

let RJFS = ctrs T "
  cat(cons(t, l), l_)-> cat(l, cons(t, l_));
  cat(nil, l)-> l;
  union(cons(v, E), P) -> union_1(in(v, P), cons(v, E), P);
  union_1(false, cons(v, E), P)-> union (E, cons(v, P));
  union_1(true, cons(v, E), P)-> union (E, P);
  union(void, E) -> E;

  in(v, cons(u, l))-> in_1(is(v, u),v, cons(u, l));
  in_1(false, v, cons(u, l)) -> in(v, l);
  in_1(true, v, cons(u, l)) -> true;
  in(v, void)-> false;

  is(succ(u), succ(v))->is(u,v);
  is(zero, succ(u))-> false;
  is(succ(u),zero)-> false;
  is(zero,zero)-> true;

  swap(exec(void, M)) -> M;
  isVoid(void)->true;
  isVoid(cons(v, E))->false;
  notNil(cons(t, l)) -> true;
  notNil(nil) -> false;
  head(cons(t, l))->t;
  tail(cons(t, l))->l;
  gt0(succ(n))->true;
  gt0(zero)-> false;

(* Instruction clock *)
    U_activ_instant_1(TERM(P, A), t, E, M) -> End( union(E, P), swap(exec(A,
        M)));
  instant(t, E, M) -> U_activ_instant_1(activ(close(t), E, M), t, E, M);
    U_activ_instant_1(STOP(t_, P, A), t, E, M) -> prod(t_, union(E, P), swap
        (exec(A, M)));
```

```
        U_activ_close_3(SUSP(t_, P, A), false, STOP(t__, P_, A_), close(t), E
            , M) -> STOP(t__,  union(P, P_), union(A, A_));
        U_activ_close_3(SUSP(t_, P, A), false, TERM(P_, A_), close(t), E, M)
            -> TERM( union(P, P_), union(A, A_));
      U_activ_close_2(SUSP(t_, P, A), false, close(t), E, M) ->
          U_activ_close_3(SUSP(t_, P, A), false, activ(close(t_),  union(E, P)
          , M), close(t), E, M);
    U_activ_close_1(SUSP(t_, P, A), close(t), E, M) -> U_activ_close_2(SUSP(
        t_, P, A), isVoid(P), close(t), E, M);
  activ(close(t), E, M) -> U_activ_close_1(activ(t, E, M), close(t), E, M);
    U_activ_close_1(STOP(t_, P, A), close(t), E, M) -> STOP(t_, P, A);
    U_activ_close_1(TERM(P, A), close(t), E, M) -> TERM(P, A);
      U_activ_close_2(SUSP(t_, P, A), true, close(t), E, M) -> STOP(eoi(t_, E
          ), void, A);


(* Instruction par *)
  activ(par(l_SUSP, l_STOP), E, M) -> U_activ_par_1(activ(par_run(l_SUSP), E,
      M), par(l_SUSP, l_STOP), E, M);
      U_activ_par_2(TERM(P, A), false, par(l_SUSP, l_STOP), E, M) -> TERM(P,
          A);
  eoi(par(l_SUSP, l_STOP), E) -> U_eoi_par_1(eoi(par_run(l_SUSP), E), par(
      l_SUSP, l_STOP), E);
      U_activ_par_2(TERM(P, A), true, par(l_SUSP, l_STOP), E, M) -> STOP(par(
          l_STOP, nil), P, A);
    U_activ_par_1(TERM(P, A), par(l_SUSP, l_STOP), E, M) -> U_activ_par_2(
        TERM(P, A), notNil(l_STOP), par(l_SUSP, l_STOP), E, M);
    U_activ_par_1(SUSP(par(l__SUSP, l__STOP), P, A), par(l_SUSP, l_STOP), E,
        M) -> SUSP(par(l__SUSP,  cat(l_STOP, l__STOP)), P, A);
  U_eoi_par_1(par(l__SUSP, nil), par(l_SUSP, l_STOP), E) -> par( cat(l__SUSP,
      l_STOP), nil);
    U_activ_par_1(STOP(par(nil, l__STOP), P, A), par(l_SUSP, l_STOP), E, M)
        -> STOP(par( cat(l_STOP, l__STOP), nil), P, A);

        U_activ_par_run_3(true, TERM(P, A), TERM(P_, A_), par_run(l_SUSP), E,
            M) -> TERM( union(P, P_), union(A, A_));
        U_activ_par_run_3(true, STOP(t_, P, A), TERM(P_, A_), par_run(l_SUSP)
            , E, M) -> STOP(par(nil, cons(t_, nil)),  union(P, P_), union(A,
            A_));
        U_activ_par_run_3(true, SUSP(t_, P, A), SUSP(par(l__SUSP, l__STOP),
            P_, A_), par_run(l_SUSP), E, M) -> SUSP(par(cons(t_, l__SUSP),
            l__STOP),  union(P, P_), union(A, A_));
    U_activ_par_run_1(true, par_run(l_SUSP), E, M) -> U_activ_par_run_2(true,
        activ(head(l_SUSP), E, M), par_run(l_SUSP), E, M);
      U_activ_par_run_2(true, TERM(P, A), par_run(l_SUSP), E, M) ->
          U_activ_par_run_3(true, TERM(P, A), activ(par_run(tail(l_SUSP)), E,
          M), par_run(l_SUSP), E, M);
        U_activ_par_run_3(true, TERM(P, A), STOP(par(nil, l__STOP), P_, A_),
            par_run(l_SUSP), E, M) -> STOP(par(nil, l__STOP),  union(P, P_),
            union(A, A_));
        U_activ_par_run_3(true, STOP(t_, P, A), SUSP(par(l__SUSP, l__STOP),
            P_, A_), par_run(l_SUSP), E, M) -> SUSP(par(l__SUSP, cons(t_,
            l__STOP)),  union(P, P_), union(A, A_));
  activ(par_run(l_SUSP), E, M) -> U_activ_par_run_1(notNil(l_SUSP), par_run(
      l_SUSP), E, M);
        U_activ_par_run_3(true, SUSP(t_, P, A), TERM(P_, A_), par_run(l_SUSP)
            , E, M) -> SUSP(par(cons(t_, nil), nil),  union(P, P_), union(A,
            A_));
```

```
  U_eoi_par_run_2(true, par(l_, nil), par_run(l_SUSP), E) -> par(cons(eoi(
      head(l_SUSP), E), l_), nil);
  eoi(par_run(l_SUSP), E) -> U_eoi_par_run_1(notNil(l_SUSP), par_run(l_SUSP),
       E);
  U_eoi_par_run_1(true, par_run(l_SUSP), E) -> U_eoi_par_run_2(true, eoi(
      par_run(tail(l_SUSP)), E), par_run(l_SUSP), E);
        U_activ_par_run_3(true, TERM(P, A), SUSP(par(l__SUSP, l__STOP), P_,
            A_), par_run(l_SUSP), E, M) -> SUSP(par(l__SUSP, l__STOP), union(
            P, P_), union(A, A_));
      U_activ_par_run_2(true, STOP(t_, P, A), par_run(l_SUSP), E, M) ->
          U_activ_par_run_3(true, STOP(t_, P, A), activ(par_run(tail(l_SUSP)),
           E, M), par_run(l_SUSP), E, M);
    U_activ_par_run_1(false, par_run(l_SUSP), E, M) -> TERM(void, void);
        U_activ_par_run_3(true, SUSP(t_, P, A), STOP(par(nil, l__STOP), P_,
            A_), par_run(l_SUSP), E, M) -> SUSP(par(cons(t_, nil), l__STOP),
            union(P, P_), union(A, A_));
        U_activ_par_run_3(true, STOP(t_, P, A), STOP(par(nil, l__STOP), P_,
            A_), par_run(l_SUSP), E, M) -> STOP(par(nil, cons(t_, l__STOP)),
            union(P, P_), union(A, A_));
      U_activ_par_run_2(true, SUSP(t_, P, A), par_run(l_SUSP), E, M) ->
          U_activ_par_run_3(true, SUSP(t_, P, A), activ(par_run(tail(l_SUSP)),
           E, M), par_run(l_SUSP), E, M);
  U_eoi_par_run_1(false, par_run(l_SUSP), E) -> par(nil, nil);


(* Instruction seq *)
      U_activ_seq_2(true, TERM(P, A), seq(l), E, M) -> U_activ_seq_3(true,
          TERM(P, A), activ(seq(tail(l)), union(E, P), M), seq(l), E, M);
        U_activ_seq_3(true, TERM(P, A), SUSP(seq(l_), P_, A_), seq(l), E, M)
            -> SUSP(seq(l_), union(P, P_), union(A, A_));
      U_activ_seq_2(true, STOP(t_, P, A), seq(l), E, M) -> STOP(seq(cons(t_,
          tail(l))), P, A);
    U_activ_seq_1(false, seq(l), E, M) -> TERM(void, void);
    U_activ_seq_1(true, seq(l), E, M) -> U_activ_seq_2(true, activ(head(l), E
        , M), seq(l), E, M);
  activ(seq(l), E, M) -> U_activ_seq_1(notNil(l), seq(l), E, M);
        U_activ_seq_3(true, TERM(P, A), TERM(P_, A_), seq(l), E, M) -> TERM(
            union(P, P_), union(A, A_));
      U_activ_seq_2(true, SUSP(t_, P, A), seq(l), E, M) -> SUSP(seq(cons(t_,
          tail(l))), P, A);
  U_eoi_seq_1(true, seq(l), E) -> seq(cons(eoi(head(l), E), tail(l)));
  eoi(seq(l), E) -> U_eoi_seq_1(notNil(l), seq(l), E);
        U_activ_seq_3(true, TERM(P, A), STOP(seq(l_), P_, A_), seq(l), E, M)
            -> STOP(seq(l_), union(P, P_), union(A, A_));


(* Instruction loop *)
  activ(loop(t), E, M) -> U_activ_loop_1(activ(par(cons(t, cons(pause, nil)),
       nil), E, M), loop(t), E, M);
    U_activ_loop_1(SUSP(par(cons(t_, nil), cons(nothing, nil)), P, A), loop(t
        ), E, M) -> SUSP(seq(cons(par(cons(t_, nil), cons(nothing, nil)), cons
        (loop(t), nil))), P, A);
    U_activ_loop_1(STOP(par(l__SUSP, nil), P, A), loop(t), E, M) -> STOP(seq(
        cons(par(l__SUSP, nil), cons(loop(t), nil))), P, A);


(* Instruction repeat_n_times *)
        U_activ_repeat_n_times_3(true, TERM(P, A), SUSP(u, P_, A_),
            repeat_n_times(succ(n), t), E, M) -> SUSP(u, union(P, P_), union
```

```
            (A, A_));
        U_activ_repeat_n_times_3(true, TERM(P, A), STOP(u, P_, A_),
            repeat_n_times(succ(n), t), E, M) -> STOP(u, union(P, P_), union
            (A, A_));
      U_activ_repeat_n_times_2(true, TERM(P, A), repeat_n_times(succ(n), t),
          E, M) -> U_activ_repeat_n_times_3(true, TERM(P, A), activ(
          repeat_n_times(n, t), union(E, P), M), repeat_n_times(succ(n), t),
          E, M);
        U_activ_repeat_n_times_3(true, TERM(P, A), TERM(P_, A_),
            repeat_n_times(succ(n), t), E, M) -> TERM(union(P, P_), union(A,
            A_));
    U_activ_repeat_n_times_1(true, repeat_n_times(succ(n), t), E, M) ->
        U_activ_repeat_n_times_2(true, activ(t, E, M), repeat_n_times(succ(n),
        t), E, M);
      U_activ_repeat_n_times_2(true, SUSP(t_, P, A), repeat_n_times(succ(n),
          t), E, M) -> SUSP(seq(cons(t_, cons(repeat_n_times(n, t), nil))), P,
          A);
      U_activ_repeat_n_times_2(true, STOP(t_, P, A), repeat_n_times(succ(n),
          t), E, M) -> STOP(seq(cons(t_, cons(repeat_n_times(n, t), nil))), P,
          A);
    U_activ_repeat_n_times_1(false, repeat_n_times(succ(n), t), E, M) -> TERM
        (void, void);
  activ(repeat_n_times(succ(n), t), E, M) -> U_activ_repeat_n_times_1(gt0(n),
      repeat_n_times(succ(n), t), E, M);


(* Instruction control *)
      U_activ_control_2(true, STOP(t_, P, A), control(v, t), E, M) -> STOP(
          control(v, t_), P, A);
      U_activ_control_2(true, TERM(P, A), control(v, t), E, M) -> TERM(P, A);
  activ(control(v, t), E, M) -> U_activ_control_1(in(v, E), control(v, t), E,
      M);
    U_activ_control_1(true, control(v, t), E, M) -> U_activ_control_2(true,
        activ(t, E, M), control(v, t), E, M);
      U_activ_control_2(true, SUSP(t_, P, A), control(v, t), E, M) -> SUSP(
          control_run(v, t_), P, A);
  eoi(control(v, t), E) -> control(v, t);
    U_activ_control_1(false, control(v, t), E, M) -> SUSP(control(v, t), void
        , void);

    U_activ_control_run_1(TERM(P, A), control_run(v, t), E, M) -> TERM(P, A);
    U_activ_control_run_1(STOP(t_, P, A), control_run(v, t), E, M) -> STOP(
        control(v, t_), P, A);
    U_activ_control_run_1(SUSP(t_, P, A), control_run(v, t), E, M) -> SUSP(
        control_run(v, t_), P, A);
  activ(control_run(v, t), E, M) -> U_activ_control_run_1(activ(t, E, M),
      control_run(v, t), E, M);
  eoi(control_run(v, t), E) -> control(v, eoi(t, E));


(* Instruction kill *)
    U_activ_kill_1(SUSP(t_, P, A), kill(v, t, u), E, M) -> SUSP(kill(v, t_, u
        ), P, A);
    U_activ_kill_1(TERM(P, A), kill(v, t, u), E, M) -> TERM(P, A);
  U_eoi_kill_1(true, kill(v, t, u), E) -> u;
  U_eoi_kill_1(false, kill(v, t, u), E) -> kill(v, eoi(t, E), u);
  eoi(kill(v, t, u), E) -> U_eoi_kill_1(in(v, E), kill(v, t, u), E);
    U_activ_kill_1(STOP(t_, P, A), kill(v, t, u), E, M) -> SUSP(kill_stop(v,
        t_, u), P, A);
```

```
  activ(kill(v, t, u), E, M) -> U_activ_kill_1(activ(t, E, M), kill(v, t, u),
       E, M);

  activ(kill_stop(v, t, u), E, M) -> SUSP(kill_stop(v, t, u), void, void);
  eoi(kill_stop(v, t, u), E) -> U_eoi_kill_stop_1(in(v, E), kill_stop(v, t, u
      ), E);
  U_eoi_kill_stop_1(false, kill_stop(v, t, u), E) -> kill(v, t, u);
  U_eoi_kill_stop_1(true, kill_stop(v, t, u), E) -> u;


(* Instruction when *)
      U_activ_when_2(true, TERM(P, A), when(v, t, u), E, M) -> TERM(P, A);
  eoi(when(v, t, u), E) -> u;
    U_activ_when_1(true, when(v, t, u), E, M) -> U_activ_when_2(true, activ(t
        , E, M), when(v, t, u), E, M);
    U_activ_when_1(false, when(v, t, u), E, M) -> SUSP(when(v, t, u), void,
        void);
      U_activ_when_2(true, SUSP(t_, P, A), when(v, t, u), E, M) -> SUSP(t_, P
          , A);
  activ(when(v, t, u), E, M) -> U_activ_when_1(in(v, E), when(v, t, u), E, M)
      ;
      U_activ_when_2(true, STOP(t_, P, A), when(v, t, u), E, M) -> STOP(t_, P
          , A);


(* Instruction await *)
    U_activ_await_1(true, await(v), E, M) -> TERM(void, void);
  eoi(await(v), E) -> await(v);
    U_activ_await_1(false, await(v), E, M) -> SUSP(await(v), void, void);
  activ(await(v), E, M) -> U_activ_await_1(in(v, E), await(v), E, M);


(* Instruction pause_for_ever *)
  activ(pause_for_ever, E, M) -> STOP(pause_for_ever, void, void);


(* Instruction pause_n_times *)
  activ(pause_n_times(succ(n)), E, M) -> U_activ_pause_n_times_1(gt0(n),
      pause_n_times(succ(n)), E, M);
    U_activ_pause_n_times_1(false, pause_n_times(succ(n)), E, M) -> TERM(void
        , void);
    U_activ_pause_n_times_1(true, pause_n_times(succ(n)), E, M) -> STOP(
        pause_n_times(n), void, void);


(* Instruction pause *)
  activ(pause, E, M) -> STOP(nothing, void, void);


(* Instruction action *)
  activ(action(v), E, M) -> TERM(void, cons(v, void));


(* Instruction generate *)
  activ(generate(v), E, M) -> TERM(cons(v, void), void);


(* Instruction nothing *)
  activ(nothing, E, M) -> TERM(void, void);
```

```
" ;
```

C Éléments historiques

1. 2/07/2010 : première version de la sémantique sans la formalisation des opérations atomiques. Expression informelle des propriétés à vérifier. L'objectif de cette première version est avant tout de fournir une première description formelle du noyau réactif suffisamment simple pour évaluer la faisabilité d'une preuve automatique en utilisant l'outil CIME.
2. 18/08/2012 : Une refonte complète du document qui est intégré au «repository» des **SugarCubes** pour être automatiquement synchronisé avec les évolutions de la spécification et de l'implantation des **SugarCubes**.

Références

[1] F. BOUSSINOT, “Rc semantics using rewriting rules,” 1992.

[2] F. BOUSSINOT, “Reseaux de processus reactifs,” Rapport de recherche RR-1588, INRIA, 1992.

[3] F. BOUSSINOT, *La programmation réactive - Application aux systèmes communicants, Collection technique et scientifique des télécommunications*. MASSON, 1996.

[4] F. BOUSSINOT and J.-F. SUSINI, “The sugarcubes toolbox : A reactive java framework,” *SOFTWARE Practice and Experience*, vol. 28, pp. 1531–1550, december 1998.

[5] G. Berry and L. Cosserat, “The esterel synchronous programming language and its mathematical semantics,” in *Seminar on Concurrency, Carnegie-Mellon University*, (London, UK, UK), pp. 389–448, Springer-Verlag, 1985.

[6] F. BOUSSINOT and R. DE SIMONE, “The sl synchronous language,” *IEEE Trans. Softw. Eng.*, vol. 22, pp. 256–266, Apr. 1996.

[7] J.-F. SUSINI, L. HAZARD, and F. BOUSSINOT, “Distributed reactive machines,” in *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, RTCSA ’98, (Washington, DC, USA), pp. 267–, IEEE Computer Society, 1998.

[8] F. BOUSSINOT, “Java Fair Threads,” Tech. Rep. RR-4139, INRIA, Feb. 2001.

[9] J.-F. SUSINI, “The reactive programming approach on top of java/j2me,” in *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES ’06, (New York, NY, USA), pp. 227–236, ACM, 2006.

[10] D. Harel and A. Pnueli, “Logics and models of concurrent systems,” ch. On the development of reactive systems, pp. 477–498, New York, NY, USA : Springer-Verlag New York, Inc., 1985.

[11] J. L. Lawall, G. Muller, and L. P. Barreto, “Capturing os expertise in an event type system : the bossa experience,” in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, (New York, NY, USA), pp. 54–61, ACM, 2002.

[12] “Bossa - a framework for scheduler development,” 2001. http ://bossa.lip6.fr/ (last check 2012).

[13] F. BOUSSINOT and L. HAZARD, “Reactive scripts,” in *Proceedings of the Third International Workshop on Real-Time Computing Systems Application*, RTCSA ’96, (Washington, DC, USA), pp. 270–, IEEE Computer Society, 1996.

Table des matières